Original published: M. Hofmann and G. Rünger. Fine-grained data distribution operations for particle codes. In M. Ropo, J. Westerholm, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users Group Meeting*, volume 5759 of *LNCS*, pages 54–63. Springer, 2009. Online available at `http://dx.doi.org/10.1007/978-3-642-03770-2_12`.

Fine-grained Data Distribution Operations for Particle Codes

Michael Hofmann[⋆] and Gudula Rünger

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

Abstract This paper proposes a new fine-grained data distribution operation `MPI_Alltoall_specific` that allows an element-wise distribution of data elements to specific target processes. This operation can be used to implement irregular data distribution operations that are required, for example, in particle codes. We present different implementation variants for `MPI_Alltoall_specific` which are based on collective MPI operations, on point-to-point communication operations, or on parallel sorting. The properties of the implementation variants are discussed and performance results with different data sets are presented. For the performance results two high scaling hardware platforms, including a Blue Gene/P system, are used.

Keywords: MPI, all-to-all communication, irregular communication, personalized communication, particle codes

1 Introduction

Implementing efficient data distribution operations is a crucial task for achieving performance in parallel scientific applications, especially on high scaling supercomputer systems. Often MPI communication operations are used to implement data distribution operations. MPI provides point-to-point operations and collective operations that implement classical communication patterns like *one-to-all*, *all-to-one* and *all-to-all*. Vector variants of collective operations like `MPI_Scatterv`, `MPI_Gatherv`, and `MPI_Alltoallv` provide a way to use the operations in applications with irregular communication patterns, too. All these communication operations require that the data elements to be distributed are organized in contiguous blocks. This can require additional efforts for preparing the data for the communication operations, especially when the data is organized in application specific data structures.

In this paper we introduce a new fine-grained data distribution operation `MPI_Alltoall_specific` that allows an element-wise distribution of data elements to specific target processes instead of the block-wise distribution currently

[⋆] Supported by Deutsche Forschungsgemeinschaft (DFG).

provided by MPI. The `MPI_Alltoall_specific` operation requires that the information about the target process of an element is located inside the element itself. This kind of data structures can be found, for example, in particles codes where the data is distributed according to domain decomposition schemes and the information about the data partitioning is associated with the single particles. We introduce several alternative implementations for the `MPI_Alltoall_specific` operation and discuss their properties. To investigate the performance of the different implementations, we present performance results on an SMP cluster using up to 256 processes and on a Blue Gene/P system using up to 4096 processes.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 proposes the new operation `MPI_Alltoall_specific` for the fine-grained data distribution. Section 4 presents different implementation variants of this operation. Section 5 shows performance results and Section 6 concludes the paper.

2 Related Work

Spatial decomposition techniques and load balancing techniques used in particles codes for molecular dynamics or astrophysical simulations often require many-to-many personalized communication (e.g., [1]). The resulting irregular communication pattern can require specialized algorithms for achieving good performance on different hardware platforms [2,3]. Also, there have been efforts to perform these data distribution operations with limited additional memory [4,5]. Sparse collective operations provide an approach to communicate only with a subset of neighboring processes, thus making the interfaces of the collective operations independent from the total number of nodes [6].

All these communication operations as well as their underlying algorithms require that the data to be distributed consists of contiguous blocks of elements. Non-contiguous blocks of elements can be distributed in MPI by creating specific MPI data types (see [7], examples 4.17 and 4.18). Because of the additional costs and resources required for the creation of these data types, the feasibility of this approach strongly depends on the number of blocks and the reuse of the data types. In [8], Bader et al. present parallel algorithms for h-relation personal communication. This communication operation sends elements to specific target locations only assuming that no process receives more than h elements. The proposed architecture-independent algorithm can be adapted for an implementation of the `MPI_Alltoall_specific` operation.

3 Fine-grained Data Distribution Operations

Message passing operations in MPI can be used to send blocks of data to target processes. The blocks consist of consecutive data elements, specified by the number of elements, the MPI data type of a single element and the starting address. Collective operations can be used to send multiple blocks to distinct processes in one pass. This coarse-grained nature (*one block for one process*) is inherent to all operations in MPI. To overcome this limitation, we propose a new fine-grained data distribution operation by specifying target processes on a per element basis.

Every process j with $0 \leq j < p$ participating in the fine-grained distribution operation contributes a list of n_j elements $e_{j0}, \ldots, e_{jn_j-1}$ and receives $\hat{n}_j$ elements $\hat{e}_{j0}, \ldots, \hat{e}_{j\hat{n}_j-1}$. Let $tproc(e)$ denote the target process of the distribution for an element e. After performing the distribution operation, every process j has received exactly those elements $\hat{e}_{ji}$ with $0 \leq i < \hat{n}_j$ and $tproc(\hat{e}_{ji}) = j$. The function *tproc* specifies only the target process, but not the resulting order of the elements on a target process. We adopt the definition of stability from sorting algorithms (see [9]) to define a special ordering of the elements after the distribution. Let $sproc(e)$ and $spos(e)$ denote the source process of an element e and its original position on the source process, respectively. We call the fine-grained distribution *stable* if for any two elements $\hat{e}_{ji}$ and $\hat{e}_{j'i'}$ with $j = j'$ and $i < i'$ either $sproc(\hat{e}_{ji}) < sproc(\hat{e}_{j'i'})$ or $sproc(\hat{e}_{ji}) = sproc(\hat{e}_{j'i'}) \wedge spos(\hat{e}_{ji}) < spos(\hat{e}_{j'i'})$. We call an implementation of the fine-grained data distribution operation stable if it always performs a stable distribution. Otherwise, the implementation is called *unstable*.

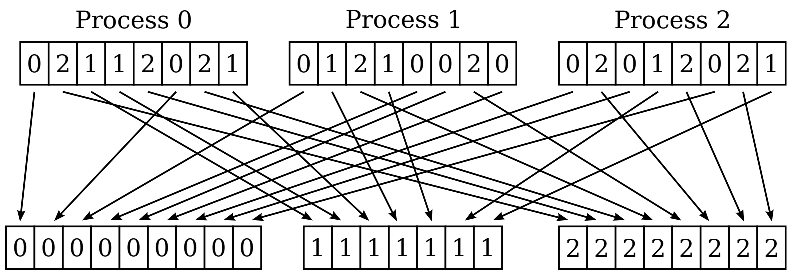


Figure 1. Fine-grained data re-distribution example with three processes.

Figure 1 shows an example with three processes for the situation before and after such a fine-grained re-distribution. Initially, the elements are arbitrarily placed on the processes. Every element holds a process rank that specifies the target process for the distribution. After the distribution all elements with target process rank 0 are located on process 0, etc. The example in Figure 1 shows a stable distribution since the original order of the elements on a single target process is maintained.

The fine-grained data distribution operation is customized for the needs of irregular applications like particle codes. In these codes, domain decomposition or partition techniques (e.g., based on space-filling curves) are used to assign single data elements to individual target processes. At least two different kinds of data distribution can be distinguished in common scientific applications. (1) Distribution of initial data (before any locality preserving scheme is applied) and (2) distribution of (slightly) altered data, e.g. after certain time steps in molecular dynamics or astrophysical simulations. In the first case, the elements are uniformly distributed to all processes and the amount of communication between all processes is almost equal. In the second case, each process exchanges elements only with a small number of neighboring processes, which leads to an unequal amount of communication load between the processes.

We propose a new collective operation MPI_Alltoall_specific to implement the fine-grained data distribution operation. It has the following prototype:

```
int MPI_Alltoall_specific(
  void *sendbuf, int sendcount, MPI_Datatype sendtype,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
  int tproc_pos, int *received, MPI_Comm comm);
```

The operation uses a send buffer (sendbuf, sendcount, sendtype) to specify the local contribution of elements and a receive buffer (recvbuf, recvcount, recvtype) to store incoming elements. Because recvcount specifies only the maximum size of the receive buffer, the actual number of elements received is returned in the output value received. The individual target processes of the elements (previously given by $tproc(e)$ for an element e) have to be part of the elements. tproc_pos specifies the displacement in bytes (relative to the base address of an element) where the information about the target process is located inside an element. This current approach requires to use a single integer value of fixed size at a fixed position and with a fixed range of values. A more flexible approach is to use a user-defined callback function that maps a single element to a target process. However, using a callback function can result in a loss of performance in comparison to having direct access to the target process values. Using callback functions also imposes a read-only access to the target process values that makes it impossible to apply modifications to these values (e.g. to gain stability, see Section 4). To support both, performance and flexibility, an alternative would be to support a set of pre-defined (parametrized) mapping functions as well as user-defined mapping functions (similar to pre-defined and user-defined reduction operations in MPI).

In comparison to the standard all-to-all communication operations, the interface of the MPI_Alltoall_specific operation is independent from the number of processes p. It depends only on the local number of elements n_j to be distributed by process j. Specifying individual target processes per element usually requires more memory than specifying only counts and displacements for each process. But, in many situations the additional storage location for the target process is already part of the application data, whereas the memory for send and receive counts and displacements (e.g., required by MPI_Alltoallv) needs to be allocated additionally.

Several MPI operations can work in-place by providing the MPI_IN_PLACE argument instead of the send or receive buffer. For in-place operations, the output buffer is identical to the input buffer and the input data is replaced by the output data of the communication operation. We call operations with separate input and output buffer *out-of-place* operations.

We give three examples for using the `MPI_Alltoall_specific` operation with different sample data sets. The examples are later used for the performance measurements in Section 5.

Example A As a first example, we consider p processes and an array of integers with values in the range of $0 \dots p-1$ as send buffer for each process (Figure 1 can be seen as an example with $p = 3$). The integers are the elements to be distributed and their values are used as target process values by specifying `tproc_pos` $= 0$. After a call to `MPI_Alltoall_specific` all integers with value j are located on process j. We use random values (in the range of $0 \dots p-1$) for the integer values.

Example B Another example can be found in particle codes, where the elements contain more complex information describing the properties of the particles. We use arrays of the following example data structure for the particles:

```
typedef struct {
  double pos[3];  /* position */
  double q;       /* charge or mass */
  double vel[3];  /* velocity */
  int tproc;      /* target process */
} particle_data_t;
```

The `tproc` component specifies the target process of the particle. The offset of the `tproc` component inside the data structure has to be used for the `tproc_pos` value and can be determined, for example, with `MPI_Get_address`.

We determine the target processes of the particles by adopting a data partitioning scheme based on space filling curves [10]. To get a distribution similar to initial data before any locality preserving scheme is applied, the particles are randomly placed inside a unit cube. This cube is equally divided into 8^6 smaller sub-cubes. The sub-cubes are numbered according to a Z-order space filling curve and then equally distributed to all processes. Each particle is located in one of the sub-cubes and gets its target process from the distribution of the sub-cubes to the processes. Because of the initial random placement of the particles, almost all particles need to be distributed to different processes.

Example C The last example uses the output data from Example B and additionally moves the particles randomly. Each particle is allowed to do a step in each of the three dimensions. The step size is limited to the half width of a sub-cube. After the movement, the particles get new target processes corresponding to the sub-cubes they have moved to. Due to the limited movement, only a small number of particles move to neighboring sub-cubes that are assigned to other processes. This leads to an unequal amount of communication load between the different processes.

4 Implementation Variants for `MPI_Alltoall_specific`

In the following, we present four implementation variants called alltoallw, alltoallv, sendrecv and sort for the `MPI_Alltoall_specific` operation and discuss their properties.

alltoallw The alltoallw implementation uses the MPI operation `MPI_Alltoallw`. `MPI_Alltoallw` is the most flexible all-to-all communication operation in MPI and allows the specification of lists with separate send and receive data types for every process. To implement `MPI_Alltoall_specific` using `MPI_Alltoallw`, each process creates p different send types with `MPI_Type_create_indexed_block` that cover exactly the fine-grained structure of the data. On process j, the k-th send type with $0 \leq k < p$ uses all indices i with $0 \leq i < n_j$ and $tproc(e_{ji}) = k$. The receive types are contiguous types, whose counts and displacements are determined in advance (this requires an additional call to `MPI_Alltoall`). Using these MPI data types, the `MPI_Alltoall_specific` operation can be implemented with a call to `MPI_Alltoallw`. An advantage of this implementation is that the original data of the input buffer remains unchanged. Potential disadvantages are high costs for creating the appropriate MPI data types. These data types can not be reused and need to be created for each call to `MPI_Alltoall_specific`. The alltoallw implementation is stable if the indexed data types and the receive displacements are created in an appropriate (straight forward) way. The alltoallw implementation can not work in-place, because the `MPI_IN_PLACE` option is currently not supported by `MPI_Alltoallw`.

alltoallv This implementation first sorts the local elements of each process according to the target process ranks. The standard operation `MPI_Alltoallv` can then be used to distribute the sorted elements as contiguous blocks. We implement the local sort, by copying the elements from the send buffer to the receive buffer and then sorting the elements from the receive buffer to the send buffer using a (stable) radix sort algorithm. Unless an additional buffer (equal to the size of the input data) is used to store the sorted elements, the local sort changes the original data of the input buffer and requires that the receive buffer is as large as the send buffer (this can be avoided using an in-place local sort). The alltoallv implementation is stable if the local sort is stable. The alltoallv implementation can not work in-place, because the `MPI_IN_PLACE` option is currently not supported by `MPI_Alltoallv`.

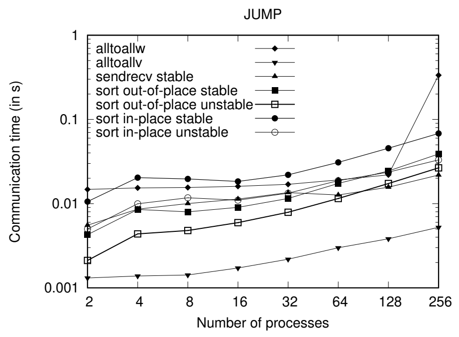
sendrecv The sendrecv implementation iterates over all elements and sends them to the target processes using point-to-point communication operations. Because transferring single elements is very inefficient, we use auxiliary buffers (one per target process) to aggregate larger messages. We use buffers equal to the size of 128 elements for each target process. `MPI_Isend` operations are used to send the messages, while at the same time every process receives messages with MPI_Irecv. The data of the input buffer remains unchanged. Additionally, this implementation variant may be advantageous if only a small number of elements (that fit into the auxiliary buffers) need to be exchanged. The sendrecv implementation performs a stable distribution if the iteration over the elements starts at the first element and if the exact receive displacements are determined in advance, so that they can be used to place received elements. An unstable variant of the sendrecv implementation can neglect the distinct receive displacements for every source process.

sort This implementation uses a parallel sorting algorithm to sort all elements according to the target process ranks. A final balancing step after the parallel sorting is used to make sure that every process receives its part of the sorted list of elements. We use the merge-based parallel sorting algorithm introduced in [11] for this implementation variant of the MPI_Alltoall_specific operation. The sort implementation performs an unstable distribution, because our underlying merge-based parallel sorting algorithm is unstable, too. The sort implementation can be made stable, by replacing the (non-distinct) target process values with new distinct keys computed from the source process rank and the original position of the elements. The original target process values can be restored afterwards. The parallel sorting algorithm works in-place, but is also able to use additional memory to improve its performance. We use the send buffer as additional memory for the out-of-place variant of the sort implementation. Consequently, the original data in the send buffer is changed. For the in-place variant, we use additional memory of fixed size (1024 elements).

5 Performance Results

We use the three examples presented in Section 3 to demonstrate the performance of the different implementations of the MPI_Alltoall_specific operation. The performance experiments are performed on the supercomputer systems JUMP and JUGENE. The IBM Power 575 system JUMP consists of 14 SMP nodes (32 Power6 4.7 GHz processors and 128 GB main memory per node) with InfiniBand interconnects for MPI communication. A vendor specific MPI implementation is used (PE MPI, part of *IBM Parallel Environment*). JUGENE is an IBM Blue Gene/P system with a total number of 16384 compute nodes (4-way SMP processor and 2 GB main memory per node) and several specific networks (3D torus network, collective network, barrier network) for MPI communication. An MPICH2 based MPI implementation optimized for the Blue Gene architecture is used [3].

Figure 2 shows the communication times for the different implementation variants of MPI_Alltoall_specific depending on the number of processes. Each process contributes 100.000 elements. The input data corresponds to Example A. The results for the unstable sendrecv implementation are not shown, because the differences to the stable variant were too small.



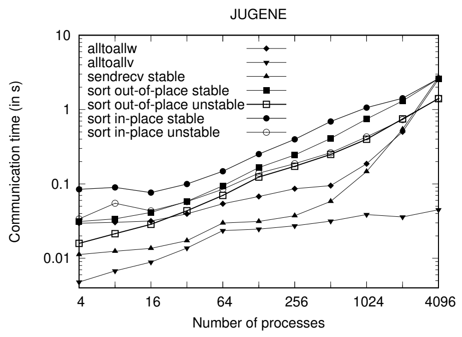
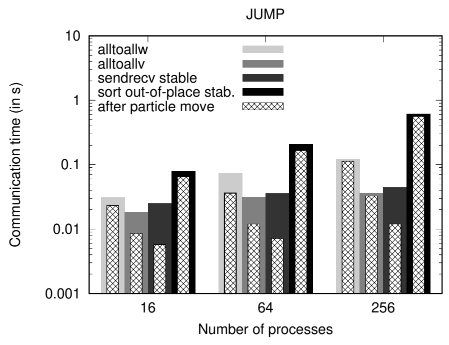


Figure 2. Communication times for the implementation variants of `MPI_Alltoall_specific` with data corresponding to Example A on the JUMP (left) and the JUGENE (right) system.

The results on both hardware platforms show that the alltoallv implementation is always the best while all other implementations are significantly slower. Comparing only the two implementation variants alltoallw and sendrecv that do not change the input data, sendrecv achieves better results. The performance of alltoallw suffers from the expensive creation of the indexed data types. On JUGENE with more than 1024 processes, the differences between these two variants become rather small. The sort implementations show differences in performance between the stable and unstable variants. This is caused by the additional work required for the key modification of the stable variant. Working in-place reduces the performance of the sort implementation on both hardware platforms, too. On JUGENE, the differences between the in-place and the out-of-place sort implementation variants become smaller for increasing numbers of processes. The communication times of all implementation variants increase for increasing numbers of processes. This is caused by the growing amount of data and the increasing amount of communication required for the distribution of the data. With p processes, the probability for a single element (with a random target process) to stay on the initial process is only about $\frac{1}{p}$.

Figure 3 shows communication times for the stable out-of-place implementation variants of `MPI_Alltoall_specific` depending on the number of processes. Each process contributes 100.000 elements. The results shown with the solid bars use the particle data of Example B. The (smaller) patterned bars show results after the particles have moved corresponding to Example C.

The results of Examples B and C confirm the good performance of the alltoallv implementation variant on both hardware platforms. The performance of the sendrecv implementation with particle data has improved and is close to alltoallv. The alltoallw and the sort implementation are significantly slower on JUMP, especially for larger numbers of processes. On JUGENE, only the sort implementation has significantly higher communication times.



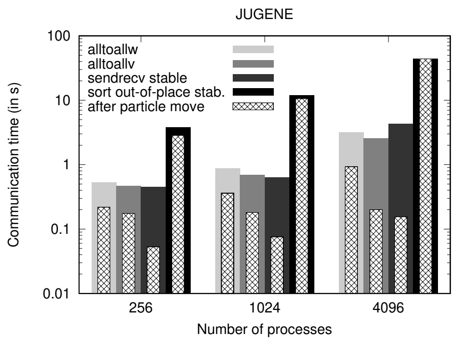


Figure 3. Communication times for the stable out-of-place implementation variants of `MPI_Alltoall_specific` with particle data from Example B (solid bars) and Example C (patterned bars) on the JUMP (left) and the JUGENE (right) system.

Depending on the number of processes, about 3-17 % of the particles need to be distributed to other processes when the particles have moved corresponding to Example C. In comparison to the results with Example B, almost all implementation variants benefit from the reduced number of elements that need to be distributed. The highest reduction in communication time is achieved by the sendrecv implementation, which becomes the best implementation variant of `MPI_Alltoall_specific` for this kind of unbalanced communication.

The overall results show that the alltoallv implementation achieves always good performance. This can be attributed to the separation between rearranging the data and sending the data to the target processes with collective communication operations. Therefore, the alltoallv implementation benefits from the performance of optimized `MPI_Alltoallv` operations. At least on Blue Gene systems, this operation is known to be very efficient [3]. The sendrecv implementation shows good results too, especially in situations where only a small portion of the data needs to be distributed. The organization of the communication of the sendrecv implementation depends only on the data to be distributed. With large numbers of processes and balanced amounts of communication between the processes, the performance is reduced by the missing adaptation to the hardware platform. The alltoallw implementation suffers from the expensive creation of the indexed data types. Additionally, the underlying `MPI_Alltoallw` operation appears to be less optimized on the chosen hardware platforms. The low performance of the sort implementation is caused by the merge-based parallel sorting algorithm. The algorithm uses multiple steps to send the elements (across intermediate processes) to the target processes. This increases the amount of communication in comparison to the other implementation variants, which send the elements directly to the target processes. The main advantage of the sort algorithm is that it can work in-place.

6 Summary

In this paper, we have proposed a new fine-grained data distribution operation called `MPI_Alltoall_specific` together with several different implementation variants. In comparison to standard MPI all-to-all communication operations, the new operation is more flexible and appropriate for applications like particle codes that require irregular or unbalanced distribution operations. The presented implementation variants possess different capabilities, e.g. in terms of guaranteeing a specific ordering of the output data, preserving the input data, or being able to work in-place by replacing the input data with the output data. We have shown performance results on two supercomputer systems with different sample data sets including particle data. The results demonstrated the advantages and disadvantages of the different implementation variants.

Acknowledgment

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

References

1. Fitch, B., Rayshubskiy, A., M., E., Ward, T., Giampapa, M., Zhestkov, Y., Pitman, M., Suits, F., Grossfield, A., Pitera, J., Swope, W., Zhou, R., Feller, S., Germain, R.: Blue Matter: Strong Scaling of Molecular Dynamics on Blue Gene/L. In: Proc. of the Int. Conf. on Computational Science 2006, Part II. (2006) 846–854
2. Liu, W., Wang, C.L., Prasanna, V.: Portable and scalable algorithm for irregular all-to-all communication. J. Parallel Distrib. Comput. **62**(10) (2002) 1493–1526
3. Almási, G., Heidelberger, P., Archer, C., Martorell, X., Erway, C., Moreira, J., 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
4. Pinar, A., Hendrickson, B.: Interprocessor Communication with Limited Memory. IEEE Trans. Parallel Distrib. Syst. **15**(7) (2004) 606–616
5. Siegel, S., Siegel, A.: MADRE: The Memory-Aware Data Redistribution Engine. In: Proc. of the 15th EuroPVM/MPI Conf., Springer (2008) 218–226
6. Höfler, T., Träff, J.: Sparse Collective Operations for MPI. In: Proc. of the 23rd Int. Parallel & Distributed Processing Symposium, HIPS Workshop. (2009) 1–8
7. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 2.1. (2008)
8. Bader, D., Helman, D., JáJá, J.: Practical parallel algorithms for personalized communication and integer sorting. J. Exp. Algorithmics **1** (1996) Article No. 3
9. Knuth, D.: The Art of Computer Programming, Volume III: Sorting and Searching. 2nd edn. Addison-Wesley (1998)
10. Aluru, S., Sevilgen, F.: Parallel Domain Decomposition and Load Balancing Using Space-Filling Curves. In: Proc. of the 4th Int. Conf. on High-Performance Computing, IEEE (1997) 230–235
11. Dachsel, H., Hofmann, M., Rünger, G.: Library Support for Parallel Sorting in Scientific Computations. In: Proc. of the 13th Int. Euro-Par Conf., Springer (2007) 695–704