Original published: M. Hofmann and G. Rünger. In-place algorithms for the symmetric all-to-all exchange with MPI. In J. G. Blas, J. Carretero, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface: 20th European MPI Users' Group Meeting (EuroMPI 2013)*, pages 105–110. ACM, 2013. Online available at `http://dx.doi.org/10.1145/2488551.2488568`.

In-place Algorithms for the Symmetric All-to-all Exchange with MPI

Michael Hofmann* and Gudula Rünger
Department of Computer Science
Chemnitz University of Technology
Chemnitz, Germany
{mhofma,ruenger}@cs.tu-chemnitz.de

ABSTRACT

The article presents two efficient in-place algorithms for the symmetric all-to-all exchange of the `MPI_Alltoallv` operation. The first algorithm performs a series of pairwise data exchanges similar to the existing algorithm used by MPICH, but with fewer consecutive communication steps and idle processes. The second algorithm uses hierarchical sets of processes that lead to a better locality of communication. Exploiting additionally available memory for performance improvements is described. Performance results for an InfiniBand cluster and an IBM Blue Gene/Q system demonstrate the performance benefits of the algorithms within a generic benchmark program and an FFT application.

Keywords

symmetric all-to-all, collective communication, in-place, limited memory, MPI

1. INTRODUCTION

The main memory of a parallel computer represents a limited resource that often decides which maximum problem size of an application can still be processed. For distributed memory parallel programming, the utilized communication operations of MPI can significantly increase the memory requirements: Data to be sent and data to be received has to be stored at separate locations in memory, and thus, the data exists twice during the data exchange (i.e., on the sender process and the receiver process). In the worst case, these doubled memory requirements occur only during data redistribution steps and lead to large amounts of unused memory during all other steps of an application.

In-place communication was introduced in MPI to reduce the memory requirements of communication operations. For collective communication operations, such as reduce, gather, scatter, or all-to-all, the key word `MPI_IN_PLACE` can be specified instead of a send buffer. In this case, the data to be sent is read from the given receive buffer before the incoming data is stored. Using the `MPI_IN_PLACE` key word for the `MPI_Alltoallv` operation restricts the collective data exchange to a so-called symmetric all-to-all exchange: Between each pair of processes, the same amount of data is exchanged (i.e., send and receive counts are equal) and the data is read and written at the same locations in the given receive buffer (i.e., send and receive displacements are equal). The symmetric all-to-all exchange represents a strong restriction of the general all-to-all exchange with arbitrary counts and displacements. However, the symmetric all-to-all exchange is the only in-place communication with the `MPI_Alltoallv` operation supported by the current MPI standard.

In this article, two efficient in-place algorithms for the symmetric all-to-all exchange with the `MPI_Alltoallv` operation are presented. Both algorithms represent a significant improvement over the algorithm used in MPICH [1]. With p parallel processes, the MPICH algorithm has a complexity of $\mathcal{O}(p^2)$ and leads to communication steps with large numbers of idle processes. The two improved algorithms have a complexity of $\mathcal{O}(p)$ and fewer idle processes. The first algorithm performs p consecutive communication steps with at most one process being idle in each step. The second algorithm performs at most $p + \lceil \log_2 p \rceil - 2$ consecutive communication steps, but uses hierarchical sets of processes that lead to a better locality of communication. Exploiting additionally available memory for performance improvements is described. The presented algorithms are not limited to the `MPI_Alltoallv` operation, but can also be used for implementing the in-place communication of the `MPI_Alltoall` and `MPI_Alltoallw` operation. All algorithms are implemented and used within a generic benchmark program and within the in-place FFT provided by the FFTW software library [6]. Performance results for an InfiniBand cluster and an IBM Blue Gene/Q system are shown.

The rest of this article is organized as follows: Section 2 presents related work. Section 3 describes the in-place algorithms for the symmetric all-to-all exchange. Section 4 shows performance results. Section 5 concludes the article.

2. RELATED WORK

The MPI specifications as well as single MPI implementations have been investigated with respect to their memory requirements [11, 5, 7]. These analyses consider the scalability of specific MPI functionalities, the sizes of internal data structures, or the management of intra- and inter-node connections for very high numbers of MPI processes. Improving the memory requirements of collective communication oper-

*Supported by the Cluster of Excellence MERGE funded by DFG.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroMPI '13, September 15 - 18 2013, Madrid, Spain
Copyright 2013 ACM 978-1-4503-1903-4/13/09 ...$15.00.

ations is limited to their interfaces. Examples are the neighborhood collective communication operations introduced in MPI version 3.0 [4], the integration of data size and displacement parameters into derived MPI datatypes [14], or the specification of data redistribution patterns based on data elements instead of MPI processes [9].

In-place buffers for collective communication operations were introduced with MPI version 2.0 and are supported by MPI implementations, such as MPICH [1] and OpenMPI [2]. Only MPICH supports in-place buffers for the `MPI_Alltoall` operation and for the symmetric all-to-all exchange with the `MPI_Alltoallv` and `MPI_Alltoallw` operation. The utilized algorithm is described in Sect. 3.1. An in-place algorithm for the general (not-symmetric) all-to-all exchange with the `MPI_Alltoallv` operation is presented in [10]. The algorithm requires additional memory as temporary buffer and leads to a complex implementation that respects arbitrary dependencies between sending and receiving (i.e., overwriting) data. The presented algorithms do not require temporary buffers and are specifically designed for the dependencies of the symmetric all-to-all exchange. The MADRE library provides memory efficient operations for redistributing equal sized data blocks [13]. Exchanging the arbitrary sized data of the `MPI_Alltoallv` operation requires to treat each data element (specified by the MPI datatype) as a separate data block, thus leading to increased memory requirements that are proportional to the size of the data to be exchanged.

Platform-specific optimizations and library-internal details of MPI implementations have been a frequent subject of research that can also affect in-place communication operations. For example, the hierarchical factor algorithm can provide series of pairwise data exchanges that are suitable for hierarchical systems, such as SMP clusters [12].

3. IN-PLACE ALGORITHMS FOR THE SYMMETRIC ALL-TO-ALL EXCHANGE

The symmetric all-to-all exchange for the in-place communication of the `MPI_Alltoallv` operation can be performed with a series of pairwise data exchanges. Each pairwise data exchange involves two processes that send their data to each other and overwrite their data with the incoming data. This functionality is provided by the `MPI_Sendrecv_replace` operation. In the following, three algorithms for performing the pairwise data exchanges are presented.

3.1 MPICH Algorithm

An implementation of the symmetric all-to-all exchange is part of the portable MPI implementation MPICH [1]. The pseudocode of the utilized algorithm is shown in Fig. 1. The algorithm is executed by all parallel processes in an SPMD way. Two loops are used to iterate over all pairs of processes. In the inner loop, the pairwise data exchange between the processes i and j is performed while all other processes continue the loops. The two loops lead to a complexity of $\mathcal{O}(p^2)$, which is inappropriate for large numbers of processes p. Furthermore, the algorithm performs data exchanges for each process with itself, which is unnecessary.

Figure 2 (left) illustrates the resulting order of the pairwise data exchanges for an example with $p = 7$ processes. The loop overhead is neglected and it is assumed that each process can perform a single data exchange in each communication step. Each entry $(i, j), 0 \leq i, j < 7$, of the 7×7 matrix represents a data transfer from process i to process j, thus entries (i, j) and (j, i) represent one pairwise data exchange. The number of each entry represents the communication step in which the data exchange is performed. For example, process 0 performs the pairwise data exchange with itself in step 1 while all other processes are idle. In step 2, processes 0 and 1 exchange data with each other while all other processes are idle. In step 3, processes 0 and 2 exchange data with each other while process 1 continues all iterations of the inner loop and performs the pairwise data exchange with itself. The number of idle processes decreases until step 7. After step 7, the number of idle processes increases and in step 13 only process 6 performs the last data exchange with itself. For p parallel processes, the MPICH algorithm performs $2p-1$ consecutive communication steps.

```
1 Let rank be the local process rank
2 for i = 0 to p − 1 do
3     for j = i to p − 1 do
4         if i = rank then
5             Exchange data with process j
6         else if j = rank then
7             Exchange data with process i
```

Figure 1: Pseudocode of the MPICH algorithm for the symmetric all-to-all exchange. The algorithm is executed by p parallel processes in an SPMD way.

3.2 Linear Shift Algorithm

The linear shift algorithm eliminates the high complexity and the large number of idle processes of the MPICH algorithm. Figure 3 shows the pseudocode of the linear shift algorithm that is executed by all parallel processes in an SPMD way. A single loop iterates over the consecutive communication steps. In each iteration, the calculated value j represents the rank of the target process for the pairwise data exchange. This data exchange is only performed if the target process is not the local process itself. For p parallel processes, the linear shift algorithm has a complexity of $\mathcal{O}(p)$ and performs at most p consecutive communication steps.

The calculation of the target process in line 3 corresponds to a linear processing of all processes according to their ranks. However, the linear processing is shifted by the rank value of the local process, thus leading to an individual order of pairwise data exchanges on each process. Figure 2 (middle) illustrates the resulting order of the pairwise data exchanges for an example with $p = 7$ processes. Apart from the skipped data exchanges in the main diagonal, the upper left part of the matrix is equal to the matrix of the MPICH algorithm in Fig. 2 (left). The lower right part of the matrix is different and shows that these pairwise data exchanges are also performed during steps 1–7. The i-th row of the matrix ($0 \leq i < 7$) corresponds to the 0-th row shifted i entries to the left and with the dropped entries reinserted on the right.

3.3 Hierarchical Sets Algorithm

The hierarchical sets algorithm divides the symmetric all-to-all exchange between all processes into a series of exchanges between disjoint sets of processes. The algorithm starts with an initial set containing all processes. This initial

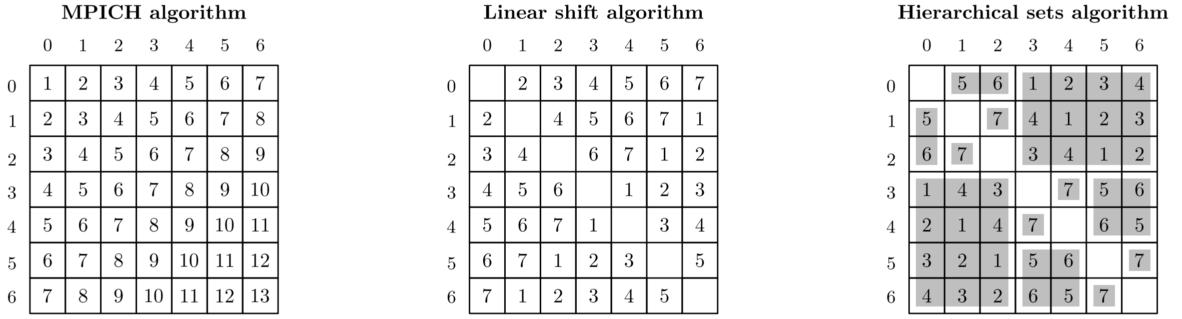


Figure 2: Illustration of the communication steps performed by the MPICH algorithm (left), the linear shift algorithm (middle), and the hierarchical sets algorithm (right) for $p = 7$ processes. Each entry $(i, j), 0 \leq i, j < 7$, of the matrices denotes the communication step in which the data transfer from process i to process j is performed. Gray rectangles for the hierarchical sets algorithm represent data exchanges that are performed between disjoint subsets of processes.

```
1 Let rank be the local process rank
2 for i = 0 to p − 1 do
3     j = i − rank + p  mod p
4     if j ≠ rank then
5         Exchange data with process j
```

Figure 3: Pseudocode of the linear shift algorithm for the symmetric all-to-all exchange. The algorithm is executed by p parallel processes in an SPMD way.

set is equally divided into two disjoint subsets of processes, each containing processes with consecutive ranks. Between the two subsets of processes, the exchange is performed with an algorithm similar to the linear shift algorithm: A loop iterates over the communication steps and each process calculates its target processes for the pairwise data exchanges according to a linear processing that is shifted by the rank value. After the exchange between the two subsets, the exchange within each subset is performed independently from each other in the same way as for the initial set.

Figure 4 shows the pseudocode of the hierarchical sets algorithm that is executed by all parallel processes in an SPMD way. Variables *low* and *high* denote the current set of processes to which the local process belongs to. A while-loop is performed as long as the current set contains more than one process (line 3). In each iteration of the while-loop, the current set is divided into two subsets by calculating the middle value *mid* between *low* and *high* (line 4). If the local process belongs to the subset of processes with lower rank values, then pairwise data exchanges are performed with processes with higher rank values (lines 5–10). Otherwise, pairwise data exchanges are performed with processes with the lower rank values (lines 12–17). The new current set of processes for the next iteration of the while-loop is selected by setting *high* (line 11) or *low* (line 18) to the middle value.

Figure 2 (right) illustrates the resulting order of the pairwise data exchanges for an example with $p = 7$ processes. The gray rectangles represent data exchanges that are performed between disjoint subsets of processes. For example, the two biggest gray rectangles represent the data exchanges

```
1  Let rank be the local process rank
2  Set low = 0 and high = p
3  while low + 1 < high do
4      mid = ⌊(low+high)/2⌋
5      if rank < mid then
6          n = high − mid
7          for i = 0 to n − 1 do
8              shift = min(rank − low, n − 1)
9              j = mid + i + shift  mod n
10             Exchange data with process j
11         high = mid
12     else
13         n = mid − low
14         for i = 0 to n − 1 do
15             shift = min(rank − mid, n − 1)
16             j = low + shift − i + n  mod n
17             Exchange data with process j
18         low = mid
```

Figure 4: Pseudocode of the hierarchical sets algorithm for the symmetric all-to-all exchange. The algorithm is executed by p parallel processes in an SPMD way.

between the subsets of processes 0–2 and 3–7 that are performed in steps 1–4. Further data exchanges between subsets of processes are performed in steps 5–6 and step 7.

The number of communication steps required for data exchanges between two disjoint subsets of processes is equal to the size of the larger subset. If the number of parallel processes p is a power of 2, then all occurring sets of processes are of even size and can always be divided into equal sized subsets. In this case, the hierarchical sets algorithm performs the optimal number of $p-1$ consecutive communication steps. Otherwise, sets of uneven size occur and each of the $\lceil \log_2 p \rceil - 1$ iterations of the while-loop can increase the optimal number of consecutive communication steps by one. Thus, the hierarchical sets algorithm performs at most $p + \lceil \log_2 p \rceil - 2$ consecutive communication steps.

3.4 Usage of Additional Memory

The algorithms described in the previous subsections use the `MPI_Sendrecv_replace` operation for pairwise data exchanges. Since the `MPI_Sendrecv_replace` operation is blocking, only a single pairwise data exchange can be performed at the same time. Furthermore, additionally available memory can not be used for improving the data buffering. To exploit additionally available memory, each process uses an auxiliary buffer as temporary storage for outgoing data.

As long as the empty space within the auxiliary buffer is large enough, the outgoing data of a pairwise data exchange is copied from the given memory buffer to the auxiliary buffer. In this case, non-blocking point-to-point communication operations are used to send the outgoing data from the auxiliary buffer and to receive the incoming data in the given memory buffer. Each copy of the outgoing data reduces the empty space within the auxiliary buffer. If the empty space is too small for a copy of the outgoing data, then the `MPI_Sendrecv_replace` operation is used. Usually, each process will first perform data exchanges with non-blocking communication operations until the empty space within the auxiliary buffer is almost exhausted. After that, the remaining data exchanges will be performed with the blocking `MPI_Sendrecv_replace` operation. Each process can individually decide about the usage of blocking or non-blocking operations, because both kinds of communication operations can be matched with each other. After all data exchanges are performed, each process waits for the completion of the non-blocking communication operations.

4. PERFORMANCE RESULTS

The algorithms described in Sect. 3 are implemented with MPI point-to-point communication operations. Performance results are shown to compare the algorithms with each other and with other existing implementations.

4.1 Experimental setup

Two supercomputer systems called JuRoPA and Juqueen have been used to obtain the performance results[1]. The JuRoPA system is a parallel computing cluster consisting of 2208 compute nodes connected with a QDR InfiniBand network. Each compute node contains two quad-core Intel Xeon processors with 2.93 GHz and 24 GiB main memory. The MPICH based ParaStation MPI library from ParTec is used. All results for the JuRoPA system were obtained executing 8 processes on each compute node.

The Juqueen system is an IBM Blue Gene/Q system with 28 672 compute nodes. Each compute node consists of a PowerPC A2 1.6 GHz processor with 16 compute cores and 16 GiB main memory [8]. An MPICH based MPI implementation specially adapted to the high-performance networks of the Blue Gene/Q platform is used. All results for the Juqueen system were obtained executing 16 processes on each compute node.

For the following generic benchmark runs, 64-bit integers are used as data elements for the symmetric all-to-all exchange. The size of the data each pair of processes exchanges with each other is chosen uniformly random. The total size of the data of each process for one symmetric all-to-all exchange is the same for all processes: 1 GB for the JuRoPA system and 300 MB for the Juqueen system.

[1]The measurements were performed at the John von Neumann Institute for Computing, Forschungszentrum Jülich, Germany.

4.2 Symmetric All-to-all Exchange

The implementations of the three algorithms for the symmetric all-to-all exchange described in Sect. 3 are compared with each other and with two implementations of the general (not-symmetric) `MPI_Alltoallv` operation, i.e., the (not-in-place) operation of the system-specific MPI library and the general in-place algorithm presented in [10]. The implementation of the general in-place algorithm is available as a software library [3] and uses 1 MB additional memory as temporary buffer on each process. Figure 5 shows runtimes of the system-specific `MPI_Alltoallv` operation, the general in-place algorithm, the MPICH algorithm, the linear shift algorithm (LSA), and the hierarchical sets algorithm (HSA) depending on the number of processes for the JuRoPA system (left) and the Juqueen system (right).

On the JuRoPA system, all implementations lead to increasing runtimes for increasing numbers of processes. The MPICH algorithm is always about a factor of two slower than the best of the other implementations. Up to 32 processes, the system-specific `MPI_Alltoallv` operation represents the fastest implementation. With more than 32 processes, the system-specific `MPI_Alltoallv` operation is significantly slower than LSA and HSA. This behavior of the system-specific `MPI_Alltoallv` operation is usually not expected and indicates the absence of a suitable optimization. The differences between LSA and HSA are only small, but HSA is faster in almost all cases (about 10 % in the best case). The behavior of the general in-place algorithm is similar to the system-specific `MPI_Alltoallv` operation. Thus, especially for large numbers of processes, the algorithms presented for the symmetric all-to-all exchange can lead to significant performance benefits.

On the Juqueen system, all implementations except the system-specific `MPI_Alltoallv` operation lead to increasing runtimes for increasing numbers of processes. The system-specific `MPI_Alltoallv` operation shows strongly increasing runtimes for up to 16 processes (i.e., all processes are executed on a single compute node). Further increasing the number of processes leads to almost constant or decreasing runtimes, thus demonstrating the good performance of the optimized collective communication operation of the Blue Gene/Q system. The MPICH algorithm has significantly higher runtimes in most cases, including a strongly increased runtime with 512 processes that is about a factor of three higher than the other in-place implementations. The runtimes of LSA and HSA differ up to 25 %, with HSA being faster in most cases. With the torus network of the Juqueen system, the hierarchical processing of HSA leads to more stable results for varying numbers of processes than the linear processing of the MPICH algorithm and LSA. The general in-place algorithm also leads to stable results for varying numbers of processes, but is significantly slower than HSA which is specialized for the symmetric all-to-all exchange.

4.3 Usage of Additional Memory

The implementation of the hierarchical sets algorithm is extended to make use of additionally available memory as described in Sect. 3.4. Figure 6 (left) shows runtimes of the system-specific `MPI_Alltoallv` operation and the hierarchical sets algorithm (HSA) with different amounts of ad-

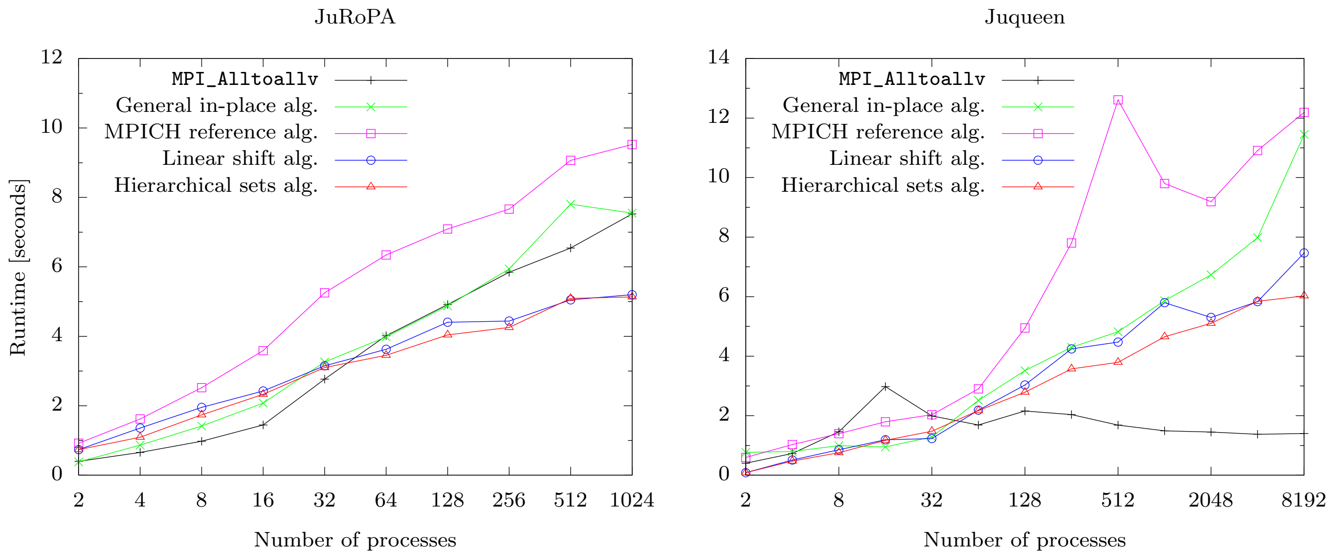


Figure 5: Runtimes of the system-specific MPI_Alltoallv operation, the general in-place algorithm, the MPICH algorithm, the linear shift algorithm, and the hierarchical sets algorithm depending on the number of processes for the JuRoPA system (left) and the Juqueen system (right).

ditional memory depending on the number of processes for the JuRoPA system.

The usage of additional memory leads to varying results that reflect the behavior seen in Fig. 5 (left). With 1000 MB additional memory, HSA performs the entire symmetric all-to-all exchange with simultaneously executed non-blocking point-to-point communication operations. In this case, HSA shows the same behavior for increasing numbers of processes as the system-specific `MPI_Alltoallv` operation, but with the latter always being 5–30 % faster. This difference can be attributed to the extra memory copies that are required by HSA for creating temporary copies of the outgoing data. Up to 32 processes, both HSA with 1000 MB additional memory and the system-specific `MPI_Alltoallv` operation lead to better results than HSA with lesser or without additional memory. With more than 32 processes, the usage of additional memory increases the runtime of HSA such that the best results are achieved without additional memory. Thus, on the JuRoPA system, it is sufficient to perform the symmetric all-to-all exchange with a series of consecutive blocking communication operations. Overlapping these blocking communication operations with additional non-blocking communication operations continuously reduces the performance for increasing numbers of processes.

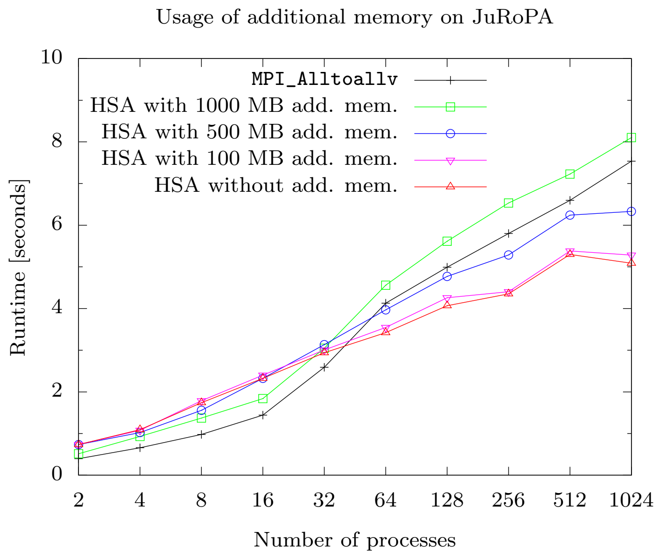
4.4 In-place FFT

The parallel fast Fourier transforms provided by the FFTW software library [6] rely on so-called global transposition steps that perform an all-to-all exchange between all processes. Depending on the block sizes that result from the FFT size and the number of processes, the all-to-all exchange can either be implemented with the `MPI_Alltoall` or the `MPI_Alltoallv` operation. If the FFT operates in-place (i.e., the input data replaces the output data), then the all-to-all exchange has also to be performed in-place. In this case, the current implementation of the FFTW (version 3.3.3) allocates a temporary buffer and performs the all-to-all exchange with point-to-point communication operations. This implementation was modified to perform the all-to-all exchange with the in-place algorithms presented in this article. The parallel benchmark program of the FFTW is used to perform three-dimensional real-valued forward FFTs that operate in-place. The FFT sizes are chosen such that only equal block sizes occur and only an in-place version of the `MPI_Alltoall` operation is required: 512x512x512 for 2 processes, 1024x1024x1024 for 16 processes, 2048x2048x2048 for 128 processes, and 4096x4096x4096 for 1024 processes. With these FFT sizes, the total size of the data of each process for the all-to-all exchange is about 512 MiB.

Figure 6 (right) shows runtimes of the all-to-all exchange and the FFT using the MPICH algorithm and the hierarchical sets algorithm (HSA) depending on the number of processes for the Juqueen system. The results represent the runtime with the modified all-to-all exchange relative to the unmodified FFTW. Using the MPICH algorithm leads to a strong increase of the runtime of the all-to-all exchange. The effect on the total runtime of the FFT is smaller and depends on the number of processes. Especially for large numbers of processes, the usage of the MPICH algorithm leads to a strong increase of the total runtime up to a factor of about 1.6. With HSA, a significant increase of the runtime is only observed with 16 processes. However, the corresponding runtimes of HSA conform to the results obtained in the previous subsections. For all other numbers of processes, there exists either no significant increase of the runtime or even a small decrease of the runtime. Even though the overall effect on the total runtime of the FFT is only small, the advantage of HSA is that no memory intensive allocations of temporary buffers are required.

5. SUMMARY

In this article, two in-place algorithms for the symmetric all-to-all data exchange of the `MPI_Alltoallv` operation have been presented. Both algorithms perform a series of



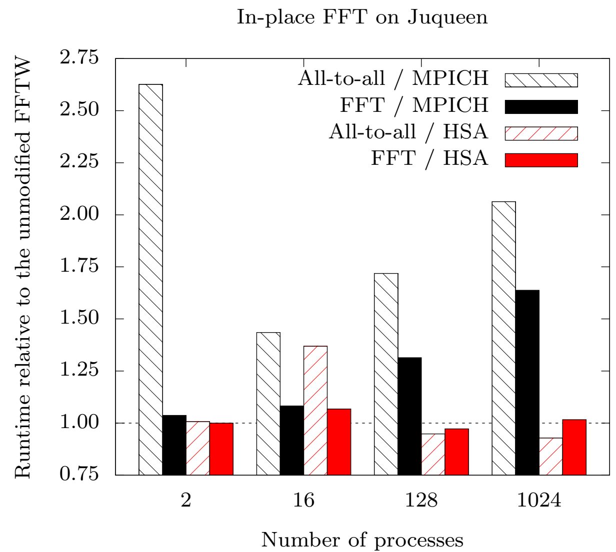


Figure 6: Left: Runtimes of the system-specific `MPI_Alltoallv` operation and the hierarchical sets algorithm with different amounts of additional memory depending on the number of processes for the JuRoPA system. Right: Runtimes of the in-place all-to-all exchange and the in-place FFT using the MPICH algorithm and the hierarchical sets algorithm depending on the number of processes for the Juqueen system.

pairwise data exchanges, but differ in the order in which these pairwise data exchanges are performed. The linear shift algorithm performs the pairwise data exchanges in order of the ranks of the processes. The hierarchical sets algorithm divides the processes hierarchically into subsets, thus leading to a better locality of communication. Performance results for an InfiniBand cluster and an IBM Blue Gene/Q system have shown that both presented algorithms perform significantly better than the existing algorithm of MPICH. The hierarchical sets algorithm usually leads to better results than the linear shift algorithm. On the InfiniBand cluster, the differences between both algorithms are only small. However, on the Blue Gene/Q system, the hierarchical sets algorithms is faster in most cases and leads to more stable results for varying numbers of processes. Exploiting additionally available memory has been described, but performance improvements are only achieved in few cases with small numbers of processes. Performance results for parallel FFTs within the FFTW software library have shown that optimized in-place algorithms can reduce the memory requirements without causing significant performance loss.

6. REFERENCES

[1] MPICH – High-Performance Portable MPI, Ver. 3.0.3. `http://www.mpich.org/`.

[2] Open MPI: Open Source High Performance Computing, Ver. 1.6.4. `http://www.open-mpi.org/`.

[3] ZMPI All-to-all In-place Library, Ver. 1.0.2. `http://www.tu-chemnitz.de/cs/PI/dl/software/`.

[4] *MPI: A Message-Passing Interface Standard Version 3.0*, 2012. `http://www.mpi-forum.org/`.

[5] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. Träff. MPI on Millions of Cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[6] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[7] D. Goodell, W. Gropp, X. Zhao, and R. Thakur. Scalable Memory Use in MPI: A Case Study with MPICH2. In *Proc. of the 18th European MPI Users' Group Meeting Conf.*, pages 140–149. Springer, 2011.

[8] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, 2012.

[9] M. Hofmann and G. Rünger. Fine-Grained Data Distribution Operations for Particle Codes. In *Proc. of the 16th European PVM/MPI Users' Group Meeting Conf.*, pages 54–63. Springer, 2009.

[10] M. Hofmann and G. Rünger. An In-place Algorithm for Irregular All-to-All Communication with Limited Memory. In *Proc. of the 17th European MPI Users' Group Meeting Conf.*, pages 113–121. Springer, 2010.

[11] M. Pérache, P. Carribault, and H. Jourdren. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In *Proc. of the 16th European PVM/MPI Users' Group Meeting Conf.*, pages 94–103. Springer, 2009.

[12] P. Sanders and J. Träff. The Hierarchical Factor Algorithm for All-to-All Communication. In *Proc. of the 8th Int. Euro-Par Conf. on Parallel Processing*, pages 799–804. Springer, 2002.

[13] S. Siegel and A. Siegel. MADRE: The Memory-Aware Data Redistribution Engine. *Int. J. High Performance Computing Applications*, 24:93–104, 2010.

[14] J. Träff. Alternative, uniformly expressive and more scalable interfaces for collective communication in MPI. *Parallel Computing*, 38(1–2):26–36, 2012.