An optimisation of allreduce communication in message-passing systems

Andreas Jocksch[a], Noé Ohana[b], Emmanuel Lanti[b], Eirini Koutsaniti[a], Vasileios Karakasis[a], Laurent Villard[b]

[a] *CSCS, Swiss National Supercomputing Centre,*
Via Trevano 131
CH-6900 Lugano
Switzerland
[b] *Ecole Polytechnique Fédérale de Lausanne (EPFL),*
Swiss Plasma Center (SPC)
CH-1015 Lausanne
Switzerland

Abstract

Collective communication, namely the pattern `allreduce` in message-passing systems, is optimised based on measurements at the installation time of the library. The algorithms used are set up in an initialisation phase of the communication, as so-called persistent collective communication, introduced in the message passing interface (MPI) standard. Part of our `allreduce` algorithms are the patterns `reduce_scatter` and `allgatherv` which are also considered standalone. For the `allreduce` pattern for short messages the existing cyclic shift algorithm (Bruck's algorithm) is applied with a prefix operation. For `allreduce` and long messages our algorithm is based on `reduce_scatter` and `allgatherv`, where the cyclic shift algorithm is applied with a flexible number of communication ports per node. The algorithms for equal message sizes are used with non-equal message sizes together with a heuristic for rank reordering. Medium message sizes are communicated with an incomplete `reduce_scatter` followed by `allgatherv`. Furthermore, an optional recursive application of the cyclic shift algorithm is applied. All algorithms are applied at the node level. The data is gathered and scattered by the cores within the node and the communication algorithms are applied across the nodes. In general, our approach outperforms the non-persistent counterpart in established MPI libraries by up to one order of magnitude or shows equal performance, with a few exceptions of number of nodes and message sizes.

Keywords: MPI, collective communication, allgatherv, reduce_scatter, allreduce

1. Introduction

1.1. Motivation

Collective communication is a component of the Message Passing Interface (MPI) library [1]. While point-to-point communication provides basic functionality, collective communication can accommodate more complex algorithms inside the library. These algorithms can be very efficient with respect to the execution time [2].

The collective communication pattern `allreduce` is frequently used in applications of emerging domains, such as artificial intelligence, but also in classical fields of physics. In this contribution we optimise collective communication for the pattern `allreduce` and, as part of it, the patterns `reduce_scatter` and `allgatherv` [3, 2], which are also considered standalone.

Our implementation makes use of an initialisation phase and is non-blocking. This type of communication is called persistent communication and is efficient for repeatedly called communication patterns, since it allows for even more sophisticated optimisations to be provided within the library. Setting up the algorithm in every communication call would be inefficient due to the expensive initialisation phase. The implementation of persistent collective communication has been started to be discussed in the literature [4, 5, 6] and is the topic of our contribution for the aforementioned patterns. Persistent collective communication plays an increasingly important role and it has a separate interface in the MPI standard from version 4.0 onward. Our library is a prototype implementation of this new MPI feature.

1.2. Background

Various network topologies have emerged during the years; fat tree, hypercube, torus, and dragonfly are some examples. Beside their topology, networks are characterised by other properties, such as bandwidth and latency, which determine their performance. The network properties are described by simplified models like the logP model [7]. The properties lead to various different algorithms for collective communication, e.g. recursive multiplying, Bruck's algorithm [8], and the ring algorithm for `allgather` operations, as well as store and forward algorithms for personalised communication with small message sizes.

Nowadays, supercomputers are typically composed of many connected shared memory nodes, which provide fast communication between processor cores on the same node

Email address: `andreas.jocksch@cscs.ch` (Andreas Jocksch)

and slow communication between cores on different nodes. This property has been considered for optimising communication algorithms by several authors [9, 10, 11, 12, 13, 14, 15]. Good speedups compared to standard implementations (MPICH, MVAPICH, OpenMPI) have been shown. However, only part of this work has been included in publicly available implementations.

The networks we are optimising for are the Dragonfly Aries network of Cray XC40 KNL and the Slingshot network of HPE Cray EX, although our algorithms, including the implementation, should also be efficient on other network architectures which can be described with the following assumptions: For the optimisation of collective communication we consider mainly a fully connected network or a network with similar characteristics to a simple bandwidth-latency model for the communication cost. The network is assumed to have multiple ports for the communication. The ports are the connections of a node to the other nodes. All algorithms discussed are assumed to operate between nodes and only optionally between cores of the same node. Data exchange or data rearrangement within the node is assumed to have zero cost for our simple model.

The basic algorithms in this contribution are recursive multiplication/division and cyclic shift (Fig. 1). Figure 2 shows

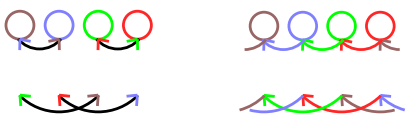

Figure 1: Recursive multiplying/dividing (doubling/halving; left) versus cyclic shift (also radix 2; right)

the data arrangement for the recursive multiplication and cyclic shift algorithms. The top blocks show the buffers filled with

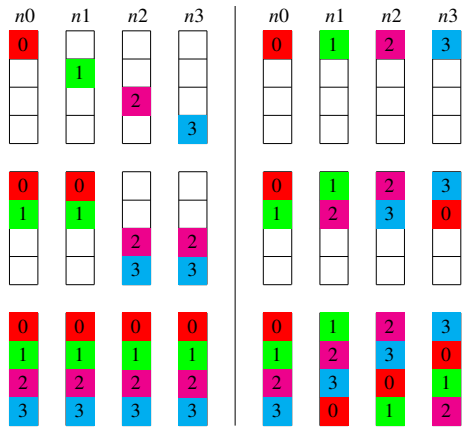

Figure 2: Scheme of recursive multiplying (left) and cyclic shift (right), radix 2, initial data (top), after step 1 (middle), after step 2 (bottom), nodes $n0$-$n3$

the initial data. In the two execution steps for radix 2 from top to bottom the buffers are filled further with the data communicated from the other nodes. The recursive multiplying algorithm has the advantage that with the last step the data is at the target and no local rearrangement on the node is necessary as for the cyclic shift algorithm. It is also an option to communicate the source data (top) to the destination (bottom) directly using one step with three so-called substeps. For recursive multiplication/division $s = \log_r p$ steps are required for p nodes and a radix r, ($r^s = p$). Within a step, $r-1$ messages need to be sent to different nodes, which can be done in our nomenclature with $r-1$ ports.

In this contribution the recursive exchange is viewed as recursive application of the cyclic shift algorithm. Thus every step of the recursive multiplication/division with radix two is a cyclic shift with two elements.

1.3. Our contribution

For our `allreduce` operations and short messages, Bruck's algorithm (the version for `allgather` [3]) is applied and further optimised by storing the results of the prefix operation instead of the actual data. This reduces the message sizes to be communicated for a fixed problem size.

Long messages are communicated with consecutive calls of `reduce_scatter` and `allgatherv`. We generalise the recursive multiplication/division and cyclic shift algorithms for `allreduce`, `reduce_scatter`, and `allgatherv` to allow for different numbers of ports for different steps, as done in [16] for `allreduce` only, using recursive exchange. In this way the underlying algorithms are adjusted for the particular network. We base the decisions for the different numbers of ports on measurements done at the installation time of the library. This procedure provides the majority of the performance improvement observed for the different cases.

Furthermore, for `reduce_scatter` and `allgatherv` with non-equal message sizes, which are part of `allreduce` for long messages or used standalone, we apply a heuristic for rank reordering in order to use the recursive multiplication/division and cyclic shift algorithms in an efficient way. At this point, we achieve additional speedups of 20%. The rank order is determined in the initialisation phase of the communication.

For medium message sizes we apply a partial `reduce_scatter` followed by `allgatherv` as suggested by Bruck and Ho [3]. This algorithm is also further formalised by the introduction of a prefix operation (as reduction) between lines and a selection of lines for the actual reduction.

In addition to the aforementioned optimisations, for the cyclic shift algorithm for `allreduce`, the prime factor decomposition for recursive multiplying [16] is generalised to a factorisation with multiple consecutive calls of Bruck's algorithm [3, 8].

We exploit the shared memory on the node by gathering and scattering messages between cores on the node before and after sending them over the network, respectively, as done in the literature. In order to accommodate all optimisations efficiently, we generate a bytecode in the initialisation phase which is interpreted in the execution phase, as demonstrated in [14].

Overall the most efficient parameters – numbers of ports of cyclic shift and the factors of the recursive application of cyclic shift – are selected based on an estimate of the best choice and

a refinement done by a parallel simulation of the different algorithmic parameters based on the initial benchmark without using the network. We obtain speedups of more than a factor of five with respect to Cray MPI and OpenMPI.

Subsequently we review related work in Sec. 1.4 and introduce in Sec. 2.1 an optimised routine for `allreduce` for short messages. Furthermore, we show in Sec. 2.2.1 and 2.2.2 optimised routines for `allgather` and `reduce_scatter_block`, respectively, as part of `allreduce` for long messages. The heuristic for non-equal message sizes is introduced in Sec. 2.2.3. The more complex algorithm for medium message sizes is shown in Sec. 2.3. The recursive application of the cyclic shift algorithm and the algorithmic complexity of our approach are discussed in Sec. 2.4 and 2.5, respectively. How the parameters of the algorithms are determined based on measurements at the library's installation time is discussed in Sec. 3. The implementation details are discussed in Sec. 4. Benchmarks made on a Dragonfly Aries and on a Slingshot network are presented in Sec. 5. In Sec. 6 we present the routines `allgatherv` and `reduce_scatter` with an application to matrix-vector multiplications of a Discrete Fourier Transform (DFT) Fourier filter of a plasma physics application, namely the ORB5 global electromagnetic Particle-In-Cell gyrokinetic turbulence code [17, 18]. Finally, we draw our conclusions in Sec. 7.

1.4. Related work

Many efforts have been made in order to optimise the collective communication for message passing, some by exploiting the shared memory on the nodes. Rabenseifner and Träff [19] optimised the algorithm for `allreduce` for non-power of two tasks, supporting non-commutative operations by providing the same reduction order and bracketing for all elements of the result vector. Sack and Gropp [20] exploited multiple ports in torus networks in order to accelerate collective communication. End et al. [21] implemented a k-port `allreduce` and made a comparison with OpenMPI 1.6.5. Almási et al. [22] optimised the collectives operations for the BlueGene/L. Chakraborty et al. [23] developed MPI collectives using shared memory on the nodes with kernel assistance. Chan et al. [24] reimplemented all MPI collective communication routines. Faraj and Yuan [25] implemented MPI collective communication with an auto-tuning approach. Graham and Shipman [26] optimised shared memory collective communication. Karwande et al. [27] developed a MPI library which selects parameters of the algorithms during compile time of the code. Patarasuk and Yuan [28] proposed a bandwidth optimal `allreduce` algorithm for SMP clusters. Bienz et al. [29] exploited the shared memory on the node for local reduction on the node and communication between nodes using multiple cores per node. Zhou et al. [30] used the shared memory for `allgather` and `allreduce` in the MPI-MPI programming model and achieved good speedups. Träff and Hunold [31] utilised shared memory on the node for a redistribution of data for parallel send and receive operations for the case of multiple networks available per node. Bouhrour and Jaeger [32] implemented and optimised persistent collective operations and obtained good speedups, up to 3 for the `reduce` collective. A short overview over our new `allreduce` and the optimisation of `reduce_scatter` and `allgatherv` was given in [33, 34].

2. Adaptations to the algorithms

There are many possible patterns for message transfers between nodes. In our case, all nodes participate in communication with recursive exchange and cyclic shift. This applies to all steps and substeps of the communication; there are no idling nodes. The recursive exchange pattern is implemented in a way that every exchange is a cyclic shift and multiple steps of recursive exchange are cyclic shifts applied in a hierarchical order (Sec. 2.4).

We take the shared memory of the nodes into account and execute our algorithms according to the following steps: (I) rearrangement of the data of all tasks on the node locally in a shared memory segment, (II) communication of the single node data to all nodes with our `allgatherv`, `reduce_scatter`, or `allreduce` algorithm, and (III) distribution of the data to all tasks on the node locally.

2.1. Short messages

For reduction operations, one can distinguish between algorithms for commutative operations and non-commutative operations (see [2] and references therein). In this contribution, we consider commutative reduction operations only. We follow the literature and base our `allreduce` collective operation for small messages on the same algorithm as `allgatherv`. A naive implementation would use the `allgather` algorithm without any modification as illustrated in Fig. 3 (left) for cyclic shift and would reduce at the end the values on every node. The top line (row) of Fig. 3 contains the data to be reduced distributed to the different processors represented by the columns. For $r = 2$ the communication is done by cyclic shifting a block of 2^{step-1} lines, from the top 2^{step-1} lines to the bottom and 2^{step-1} columns to the left. In Fig. 3 all data received in one particular step has one colour. Note that for non 2^n nodes the last step involves less lines: in the example of Fig. 3 these are the last three lines marked in red. Without loss of generality, we assume that the reduction operation is a sum. Here, we modify the scheme and do not store the values at the lines but columnwise the partial sum (inclusive scan) from the top to the bottom (Fig. 3 right). While, in the original scheme (Fig. 3 left), the l'th line shifted by k columns to the left is the $l + k$'th line, in our modified version, for a block of lines from 1st to n'th shifted by k columns to the left and k lines to the bottom, the prefix sum for the shifted lines is computed by adding the prefix sum of line k (Fig. 3 right). For example, after the second communication step for the eighth column ($n7$) on the fourth line the value is $\sum_{i=7}^{A} i$. After the next communication step the value at line eight can be computed by adding the value from the fourth line of column one $\sum_{i=0}^{3} i$ which results in $\sum_{i=7}^{A} i + \sum_{i=0}^{3} i$. We denote line four as the parent line of line eight.

This idea allows for less lines to be communicated, since for computing the final result on the bottom line, for $r = 2$, only

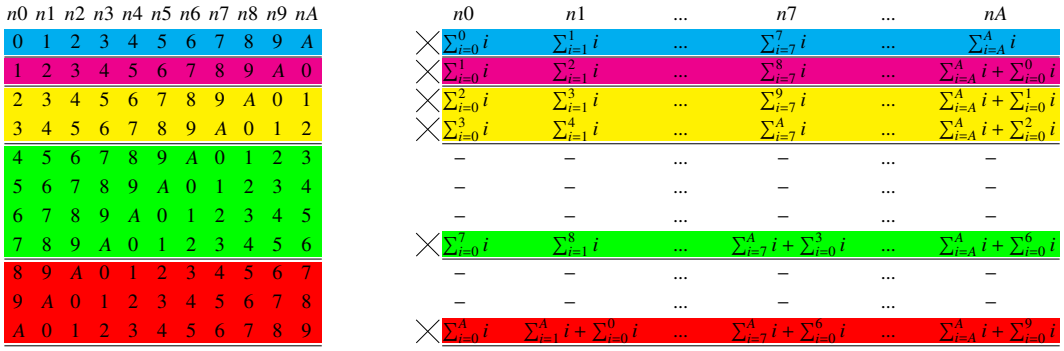

Figure 3: Cyclic shift algorithm adapted from `allgather` (left), nodes $n0$-nA with messages 0-A (hexadecimal notation), radix 2, horizontal lines ——— indicate the end of every step, all data received in one particular step has one colour, X are the lines required for `allreduce`, for sum reduction inclusive scans from top to bottom which are actually stored and communicated (right)

the lines which are marked with an X on the left are required, the rest of the lines are not needed and are displayed for the illustration of the algorithm only. In case of complete steps, e.g. for a radix of $r = 2$ and 2^n nodes the lines 2^{step} (destination lines of the step) need to be communicated only. For non 2^n nodes but a radix $r = 2$ more lines need to be communicated in some steps than the previous last line 2^{step-1} (Fig. 3). This is due to the incomplete last step of the cyclic shift. The additional lines double at most the data volume, since the last line requires an additional parent line to the one at 2^{step-1}, which is at the latest communicated in the previous step, and this parent line can be considered recursively as the new last line. In Fig. 3 it is the lines eight ($2^{(step=4)-1}$) and three which are the parents of line ten, therefore line three and four need to be communicated in step two. The generalisation to non-equal arbitrary numbers of ports is straightforward. It might be efficient to set the number of ports larger than one (see Sec. 3).

The detection of the lines that are not needed is done as follows (Alg. 1): All lines of all stages are marked as not needed.

Algorithm 1: Detection of lines to be commented out

```
lines_used[] ← false
done ← false
while not done do
  done ← true
  for stage ← max_stage to 0 do
    for line ← max_line[stage] to 0 do
      if from_task[stage, line] = task then
        if lines_used[from_line[stage, line]] = false then

          lines_used[from_line[stage, line]] ← true
          done ← false
        end if
      end if
    end for
  end for
end while
```

The lines of the last stage are analysed and if they represent the complete result they are marked as needed. Afterwards, all lines which contribute to the already marked lines are also marked as needed. We pass repeatedly in a loop from the last stage to the first one and set these marks. The loop ends if no modification is done during one pass.

2.2. Long messages

For long messages we follow Rabenseifner's approach [2] and perform a `reduce_scatter` followed by an `allgatherv`. These two routines will be discussed in the next sections. With the cyclic shift algorithm for these routines, we are not bound to any particular node count, such as the 2^n used in the literature.

In the following, we discuss (in reverse order) the optimisations for the fixed message size algorithms of `allgather` and `reduce_scatter_block` and the modifications for the variable message size versions algorithms applied.

2.2.1. Allgather

The allgather operation transmits a piece of information from every participating rank to every other rank. Thus, at the end of the operation every rank contains the same information, which is the collected data from all ranks.

Here we discuss equal message sizes (for non-equal ones see Sec. 2.2.3). There are several options conceivable to perform the operation. In the literature, the most commonly used ones, which reduce the number of communication steps, are based on recursive multiplying (typically doubling) or cyclic shift (Bruck's algorithm [8]), see Fig. 1. In contrast to the naive algorithm, these algorithms do not send the information directly from the source to the destination rank but apply forwarding. Thus, the amount of data sent through the network remains unchanged, but the number of communication steps is reduced. In Fig. 1 the algorithms are based on radix 2. At every step the information on every node is doubled (see Fig. 2): the data present in the two communication partners before the step becomes available to both of them after the step. We would like to emphasise that recursive multiplying and cyclic shift can be performed with radixes larger than two [8, 35] or different numbers of ports for different steps. Figure 4 shows a communication done in two steps with four and two ports. We speak about the number of ports since the formula with the radix $r^s = p$

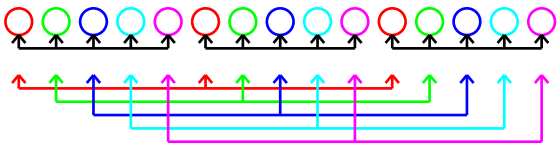

Figure 4: Recursive multiplying/dividing with four and two ports

is not valid in this case. The naive algorithm is equivalent to recursive multiplying or cyclic shift with a radix equal to the number of nodes. Any combination of these algorithms seems to be possible, but we will not discuss the combination of cyclic shift and recursive multiplying for the standalone application of `allgather`.

Within each step, the exchange of messages can be done in parallel using the different ports. As the size of the messages grows from step to step, we consider this flexibility as essential and use a different number of ports for the different steps of the cyclic shift algorithm (see Sec. 3).

2.2.2. *Reduce_scatter_block*

The optimisation of `reduce_scatter` operations has been the key aspect of several studies [36, 37]. The `reduce_scatter` operation can be considered as the reversed `allgatherv` operation in the same way as reduce is the reversed operation of broadcast. Therefore, the same algorithms are applied in reverse order: Recursive division and cyclic shift. We illustrate the cyclic shift algorithm in Fig. 5, in analogue to the display of the `allgather` algorithm in [8]. While the cyclic shift algorithm known from `allgather` has been analysed for the application to `reduce_scatter_block` [3] and it has been applied as a similar algorithmic scheme to this collective communication [37], we apply the same optimisations as for `allgather` with different numbers of ports for the different steps to `reduce_scatter_block`. There is one major difference between forward execution and reverse execution of the algorithm, however. While in the `allgatherv` case, buffers might be used for sending through multiple ports at the same time, in the `reduce_scatter` case this is only possible with an intra-node reduction. Thus, the memory requirement is higher for `reduce_scatter`, since we assume that the receiving node first gets the data in an empty buffer and, second, performs the arithmetic operation. The memory requirements are increasing with an increasing number of ports.

2.2.3. *Non-equal message sizes*

For the collective communications `allgather` and `reduce_scatter_block` with non-equal message sizes (`allgatherv`, `reduce_scatter`), the principle that every rank performs the same number of operations with the same message sizes, which is due to symmetry, no longer applies. This gives room for optimisations. However, in our approach we will leave the basic algorithms unmodified. We exploit the option of rank (or node) reordering for the algorithm (not for the network). Our heuristic for non-equal message sizes is to pair small messages with large messages for the different communication steps. The different ranks are grouped in a tree-like order (Fig. 6). For every pairing step with an odd number of messages, the smallest message is taken out and added in the next step with an odd number of messages. For the resulting messages, as for an even number of messages, the smallest one will be paired with the largest one, the second smallest one with the second largest one, and so on. The two messages within one pair are sorted. The sums of the message sizes of the pairs become the message sizes of the next step. For example, in Fig. 6 the nodes will be ordered $n0$, $n3$, $n4$, $n1$, $n6$, $n2$, $n5$. While for equal message sizes recursive multiplying and cyclic shifting (for `allgatherv`) require the same execution time, for non-equal message sizes they do not. The example in Fig. 7 shows both algorithms applied to reordered messages of sizes 1, 1, 0, 2 at the beginning of the communication with a communication time $T_{comm} = 4$ for recursive multiplying (left) and $T_{comm} = 5$ the cyclic shift (right), assuming zero latency and a bandwidth of one. In this case, but not in general, all arrangements other than the one presented for the recursive multiplying algorithm give $T_{comm} = 5$, and for the cyclic shift all arrangements give $T_{comm} = 5$.

A further example is the ordered messages of size 0, 1, 0, 1, 0, 1, 0, 1. For $r = 2$ both the recursive multiply and the cyclic shift algorithm equalise in the first step the message sizes to eight times one. The following two steps both double the message size to two and four. Contrary to our heuristic, the worst rank order appears to be messages sorted by size 0, 0, 0, 0, 1, 1, 1, 1. After step one and two at least one message has the size two and four, respectively. Thus the communication time is increased.

We would like to remark that our binary pairing seems to be the most natural scheme for the binary exchange algorithm. For higher radixes or flexible numbers of ports, higher order conjunctions (instead of pairing), e.g., smallest message, median message, and largest message for $r = 3$, seem to match the communication ideal.

In the current implementation the rank reordering procedure is executed redundantly on all nodes for each pairing step using the quicksort algorithm.

2.3. *Medium size messages*

For medium message sizes we apply a trade-off between the algorithms for short and long messages discussed by Bruck and Ho [3] and recently applied by Kolmakov and Zhang [38]. An incomplete `reduce_scatter` followed by an `allgather` with reduction is applied. As for the short message algorithm we apply the scheme with the prefix operation between lines and again not all lines are needed.

In order to implement the algorithms as flexible as possible, we introduce a mini language in which the algorithms are expressed. Figure 8 shows the code for the `allreduce` realised with partial `reduce_scatter` and `allgatherv` for node five, eight nodes, rank at the node zero, and four tasks per node. Hashes `#` indicate comment lines, which for `#STAGE` correspond to the lines skipped in Fig. 3. The keywords `STAGE`, `FRAC`, `TO`, and `FROM` followed by numbers describe the stage of the

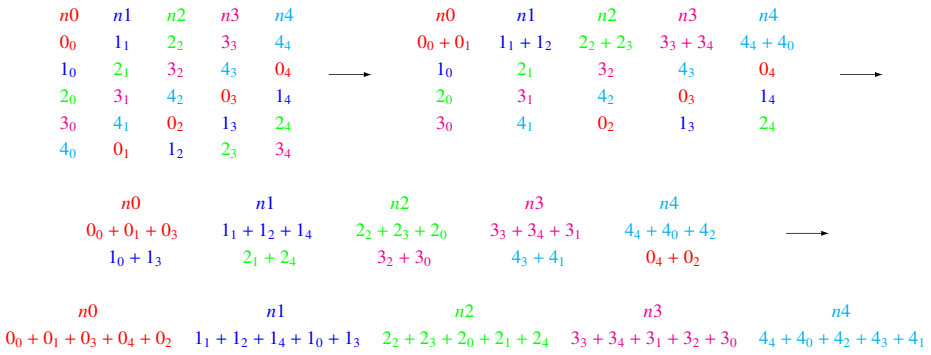

Figure 5: Cyclic shift for `reduce_scatter`, radix $r = 2$, nodes $n0$-$n4$, numbers 0-4 are messages to be reduced on the corresponding destination node, subscripts 0-4 indicate the source node, + is the reduction operation

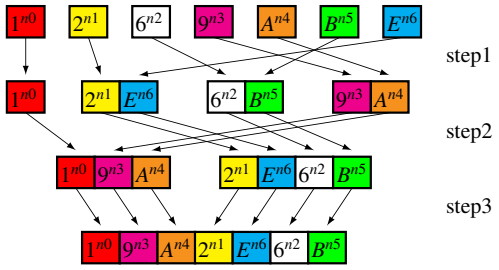

Figure 6: Pairing of messages/nodes, numbers 1, 2, 6, 9, *A*, *B*, and *E* are sizes of the messages (hexadecimal notation), superscripts $n0$-$n6$ are the nodes

Algorithm 2: Selection of lines for partial reduction

```
used[] ← true {loops with negative steps}
for ref_line ← max_line to max_line_old do
  for line ← max_line_old − 1 to 0 do
    if frac[stage, line] = frac[stage, ref_line] &&
    used[frac[stage, line]] then
      from_value[stage, ref_line] ← task {local
      reduction}
      from_line[stage, ref_line] ← line
      used[frac[stage, line]] ← false
    end if
  end for
end for
```

execution, the fraction of the input data, the ranks to which data is sent, and the ranks from which data is received, respectively. The fraction value specifies at which node the part of the message was located after a fictive complete `reduce_scatter` of the initial data. The sizes of these fractions are given by `MESSAGE_SIZES`. The data received is indicated with tuples `X|Y` where `X` is the rank number and `Y` is the line number from which it is received. The rank numbers for data sent to or received from might be the same rank as the process itself, which means the data resides on the rank or might be combined with other lines in reduction operations. A rank number of minus one represents data input or output. At every stage the fraction values remain and only the "to" and "from" values are adapted. For the partial `reduce_scatter` phase lines received from other nodes are combined with reduction operations to the lines present. For the `allgatherv` phase, with the progress of the execution, lines are added at the bottom. After every addition reduction operations are performed between lines received and present already. The procedure of selecting lines for the reduction is the following (Alg. 2): The fraction value of a line received is from bottom to top compared with the fraction value of the lines present. If the same value is found, the line found is reduced to the new line at the bottom.

The output shown in Fig. 8 is processed further in multiple steps (see Sec. 4), which need the parameters `MESSAGE_SIZES` and `DATA_TYPE`. Also, the rank numbers correspond to the node numbers and will be replaced with the real MPI ranks later, before the byte code generation.

Despite this generalisation to medium message sizes, the algorithm does not necessarily provide the optimum with respect to communication cost. One can find better (or closer to optimal) solutions when considering a further degree of freedom, the decomposition into prime factors.

2.4. Decomposition into prime factors

The algorithms discussed, can be applied independently for the factors of the number of nodes. These factors can be the prime factors or combinations of them. The factorisation follows the idea of [16]. Our algorithm can be considered to have three phases: (I) An incomplete `reduce_scatter`, (II) an `allreduce` of the data, and (III) an `allgather`. Every phase has at least one factor, where the product of phase one and two factors is the number of nodes, while phase three is the reverted phase one with respect to factors, single phases can have the factor one only. The code for the recursive application of cyclic shift in our mini language is shown in Fig. 9. While in Fig. 8 the `TO` and `FROM` values are cyclic with respect to all nodes in Fig. 9 they are cyclic with respect to the local group. The initial data is recursively cyclic shifted (unchanged for node zero). With the progress of the execution, lines are added at the bottom. It becomes visible (number of lines `TO` and `FROM` for every step) that for the parameters chosen, the decomposition into prime

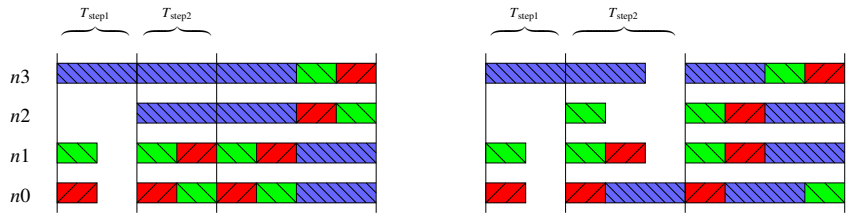

Figure 7: Modelled execution for non-equal message sizes for recursive doubling (left) and cyclic shift (right) with radix 2; nodes $n0$-$n3$ with initial message sizes 1, 1, 0, 2; execution times of the two substeps T_{step1} and T_{step2} proportional to the message size; different hashes indicate different message tags; the longest message (horizontal extent) determines the communication time

factors leads to a smaller data volume that needs to be communicated compared to the application of the algorithm to the whole problem (Fig. 8).

Since the number of nodes in Figs. 8, 9 is the special case 2^3, we would like to demonstrate the effect for the non-2^n number of nodes 39. Table 1 shows the number of steps and the amount of data to be communicated for different decomposition factors for a fixed number of ports equal to one ($r = 2$). For 39 nodes the number of steps is the same for the cyclic shift

factors(numbers of ports)	steps	size
39(1 1 1 1 1 1)	6	8
3(1 1) 13(1 1 1 1)	6	7
13(-1 -1 -1 -1) 3(-1 -1) 3(1 1) 13(1 1 1 1)	12	76/39
39(-1 -1 -1 -1 -1 -1) 39(1 1 1 1 1 1)	12	76/39

Table 1: Number of communication steps and communication size for 39 nodes and different factorisations, phase (I), phase (II), phase (III)

applied directly to the 39 nodes or applied to 13 and 3 nodes. However, the amount of data communicated differs, it is 8 times the basic message size for 39 and 7 times the basic message size for 13 and 3. For long messages it is more efficient to apply `reduce_scatter` followed by `allgatherv` where it makes no difference between the direct application of cyclic shift (factor 39) or the split into factors 13 and 3. In both cases 12 steps are required and 76/39 times the basic message size. The 76/39 value, slightly smaller than 2, can be explained by our specific implementation. The 6 steps solutions are optimal with respect to latency, while the 12 steps solutions are bandwidth optimal.

If the numbers of ports equal the factors minus one the `allgatherv` algorithm corresponds to the approach of [16]. Alternative algorithms have been described in the literature, e.g. the binary blocks algorithm [2]. Comparisons and possible combinations with these algorithms are still under investigation.

2.5. Algorithmic complexity

Since our algorithm's parameters depend on measurements, the execution time is case-dependent and not to be determined by a simple formula. However, in the limit of very long messages or a very large number of processors combined with very short messages the automatic parameter selection no longer changes and shows equal numbers of ports. The algorithmic complexity of `allgather` for equal numbers of ports $n_{ports} + 1 = r$ is

$$T_{comm} = \alpha \log_r p + \beta((p-1)/(r-1)/p)n \,, \tag{1}$$

as shown in [8], where T_{comm} is the time spent in communication, n/p the number of bytes sent per node (assuming equal message sizes), p the number of nodes participating, r the radix of the algorithm, α the time required for a single step, and β the time required for a single byte sent per node.

Since the algorithms for `allgatherv` and `reduce_scatter` differ in the direction of execution only, the algorithmic complexity is the same for both cases, except that the cost of reduction needs to be added for `reduce_scatter`. It is

$$T_{comm} = \alpha \log_r p + \beta((p-1)/(r-1)/p)n + \gamma((p-1)/(r-1)/p)n \,, \tag{2}$$

where γ is the computational cost per byte for the reduction [2].

In the limits of very short and very long messages, these formulas also apply to our `allreduce`. For very short messages Eq. (1) represents the algorithm, since the deletion of lines can be neglected for the latency dominated communication. For very long messages the sum of Eqns. (1) and (2) is representative, lines are not deleted in those algorithms. In this case the algorithmic complexity is the sum of the complexity of `reduce_scatter` and `allgatherv`.

A more theoretical aspect is the algorithmic complexity of the initialisation phase. The example of the rank reordering procedure for the current implementation shows the complexity

$$T_{reorder} = \delta p (\log_2 p)^2 \,, \tag{3}$$

where δ is the time required for a basic operation – compare and swap if necessary – of the sorting algorithm and p the number of nodes (the exponent of two in Eq. (3) appears because of the repeated execution of the quicksort). Other solutions for the sorting as a distributed sorting can be used [39], which become relevant for many MPI tasks [40], in order to reduce the algorithmic complexity of the initialisation. Otherwise, the initialisation cost would dominate for a very large number of tasks (see Eqns. (1), (2), and (3)).

3. Parametrisation

The simple bandwidth-latency network model does not give any indication about which numbers of ports (or radix r) to

```
 PARAMETER COLLECTIVE_TYPE ALLREDUCE
 PARAMETER NODE 5                # group with 8 nodes
 PARAMETER NUM_NODES 8           # 1 reduce_scatter and
 PARAMETER NUM_PORTS 8(-1 1 1 1) # 3 allgatherv; 1 port each
 PARAMETER NODE_RANK 0                       #
 PARAMETER NODE_SIZE 4                       # parameters
 PARAMETER MESSAGE_SIZES 8 8 8 16 8 16 8 16 # for further
 PARAMETER DATA_TYPE LONG_INT                # processing
# initial data
 STAGE 0 FRAC 5 TO 5 FROM -1|0
 STAGE 0 FRAC 6 TO 5 FROM -1|1
 STAGE 0 FRAC 7 TO 5 FROM -1|2
 STAGE 0 FRAC 0 TO 5 FROM -1|3
 STAGE 0 FRAC 1 TO 1 FROM -1|4
 STAGE 0 FRAC 2 TO 1 FROM -1|5
 STAGE 0 FRAC 3 TO 1 FROM -1|6
 STAGE 0 FRAC 4 TO 1 FROM -1|7
# reduce_scatter step
 STAGE 1 FRAC 5 TO 5 4 FROM 5|0 1|4
 STAGE 1 FRAC 6 TO 5 4 FROM 5|1 1|5
 STAGE 1 FRAC 7 TO 5 4 FROM 5|2 1|6
 STAGE 1 FRAC 0 TO 5 4 FROM 5|3 1|7
# 1st allgatherv step
 STAGE 2 FRAC 5 TO 5 3 FROM 5|0
 STAGE 2 FRAC 6 TO 5 FROM 5|1
 STAGE 2 FRAC 7 TO 5 FROM 5|2
 STAGE 2 FRAC 0 TO 5 FROM 5|3
 STAGE 2 FRAC 6 TO 5 3 FROM 6|0 5|1
 STAGE 2 FRAC 7 TO 5 3 FROM 6|1 5|2
 STAGE 2 FRAC 0 TO 5 3 FROM 6|2 5|3
 STAGE 2 FRAC 1 TO 5 3 FROM 6|3
# 2nd allgatherv step
 STAGE 3 FRAC 5 TO 5 1 FROM 5|0
 STAGE 3 FRAC 6 TO 5 FROM 5|1
 STAGE 3 FRAC 7 TO 5 FROM 5|2
 STAGE 3 FRAC 0 TO 5 FROM 5|3
 STAGE 3 FRAC 6 TO 5 1 FROM 5|4
 STAGE 3 FRAC 7 TO 5 FROM 5|5
 STAGE 3 FRAC 0 TO 5 FROM 5|6
 STAGE 3 FRAC 1 TO 5 FROM 5|7
 STAGE 3 FRAC 7 TO 5 1 FROM 7|0 5|5
#STAGE 3 FRAC 0 TO 5 FROM 7|9 5|6
#STAGE 3 FRAC 1 TO 5 FROM 7|9 5|7
#STAGE 3 FRAC 2 TO 5 FROM 7|9
 STAGE 3 FRAC 0 TO 5 1 FROM 7|4 5|6
 STAGE 3 FRAC 1 TO 5 1 FROM 7|5 5|7
 STAGE 3 FRAC 2 TO 5 1 FROM 7|6
 STAGE 3 FRAC 3 TO 5 1 FROM 7|7
# 3rd allgatherv step
 STAGE 4 FRAC 5 TO 5 FROM 5|0
 STAGE 4 FRAC 6 TO 5 FROM 5|1
 STAGE 4 FRAC 7 TO 5 FROM 5|2
 STAGE 4 FRAC 0 TO 5 FROM 5|3
 STAGE 4 FRAC 6 TO 5 FROM 5|4
 STAGE 4 FRAC 7 TO 5 FROM 5|5
 STAGE 4 FRAC 0 TO 5 FROM 5|6
 STAGE 4 FRAC 1 TO 5 FROM 5|7
 STAGE 4 FRAC 7 TO 5 FROM 5|8
#STAGE 4 FRAC 0 TO 5 FROM 5|9
#STAGE 4 FRAC 1 TO 5 FROM 5|10
#STAGE 4 FRAC 2 TO 5 FROM 5|11
 STAGE 4 FRAC 0 TO 5 -1 FROM 5|9      # result
 STAGE 4 FRAC 1 TO 5 FROM 5|10
 STAGE 4 FRAC 2 TO 5 FROM 5|11
 STAGE 4 FRAC 3 TO 5 FROM 5|12
 STAGE 4 FRAC 1 TO 5 -1 FROM 1|0 5|10 # result
#STAGE 4 FRAC 2 TO 5 FROM 1|14 5|11
#STAGE 4 FRAC 3 TO 5 FROM 1|14 5|12
#STAGE 4 FRAC 4 TO 5 FROM 1|14
 STAGE 4 FRAC 2 TO 5 -1 FROM 1|4 5|11 # result
#STAGE 4 FRAC 3 TO 5 FROM 1|15 5|12
#STAGE 4 FRAC 4 TO 5 FROM 1|15
#STAGE 4 FRAC 5 TO 5 FROM 1|15 5|0
 STAGE 4 FRAC 3 TO 5 -1 FROM 1|8 5|12 # result
#STAGE 4 FRAC 4 TO 5 FROM 1|16
#STAGE 4 FRAC 5 TO 5 FROM 1|16 5|0
#STAGE 4 FRAC 6 TO 5 FROM 1|16 5|4
 STAGE 4 FRAC 4 TO 5 -1 FROM 1|9      # result
 STAGE 4 FRAC 5 TO 5 -1 FROM 1|10 5|0 # result
 STAGE 4 FRAC 6 TO 5 -1 FROM 1|11 5|4 # result
 STAGE 4 FRAC 7 TO 5 -1 FROM 1|12 5|8 # result
```

Figure 8: Allreduce with partial `reduce_scatter` and `allgatherv`

choose for bandwidth dominated communication. Furthermore, it is not always clear how many ports per node are available. In order to choose the optimal parameters, we apply a tuning approach. At the installation phase of the library, measurements of communication times are done for different message sizes. Based on that, the numbers of ports are chosen. For all possible combinations of numbers of ports, the communication time is estimated from interpolations of the measurements performed during installation. This try-all method is applied for simplicity, since the overall execution time is negligible but can be reduced significantly, e.g., if the optimum number of ports is determined independently for every step, which leads to a good approximation of the global optimum.

The total number of messages between nodes is, in any case, smaller with our shared memory approach than for a naive implementation, since messages are merged. This is advantageous with respect to network congestion. The option of splitting the messages between nodes and using multiple senders and receivers for their transmission is not exploited here. However, the load on the network might still affect the communication. If required, the measurement runs can be done with different well defined workloads on the network in the background as done by the GPCNeT benchmark [41]. Thus, the parameters of the algorithms can be adapted to the network load.

Our experiments show that it is efficient to apply high and low radixes for short messages and long messages, respectively. This is supported by the findings in [42], where a saturation effect for long messages is described. The back-load of the network boosts this effect. The $r-1$ ports can be physical ports in the sense of multiple cores performing communication or logical ports if one core performs multiple non-blocking point-to-point communications.

The determination procedure of numbers of ports with the try-all method is an estimate. In order to consider the lines cancelled for `allreduce` we take the parameters of the best estimations and distribute them to the MPI ranks. Each rank simulates a couple of parameter sets based on the benchmark. Thus the simulation is serially executed for a single rank but parallelised over the ranks. For messages occurring in intermediate communication steps their sizes are not necessarily equal between nodes. Therefore the sizes are rounded up to the highest occurring value of the parallel paths of the fictive nodes (we

```
 PARAMETER COLLECTIVE_TYPE ALLREDUCE_GROUP
 PARAMETER NODE 5      # group with 2 nodes reduce_scatter,
 PARAMETER NUM_NODES 8 # group with 4 nodes 2 allgatherv,
 PARAMETER NUM_PORTS 2(-1) 4(1 1) 2(1) #
# and group with 2 nodes allgatherv; 1 port each
 PARAMETER NODE_RANK 0                     #
 PARAMETER NODE_SIZE 4                     # parameters
 PARAMETER MESSAGE_SIZES 8 8 8 16 8 16 8 16 # for further
 PARAMETER DATA_TYPE LONG_INT              # processing
# initial data
 STAGE 0 FRAC 6 TO 5 FROM -1|0
 STAGE 0 FRAC 7 TO 5 FROM -1|1
 STAGE 0 FRAC 4 TO 5 FROM -1|2
 STAGE 0 FRAC 5 TO 5 FROM -1|3
 STAGE 0 FRAC 2 TO 4 FROM -1|4
 STAGE 0 FRAC 3 TO 4 FROM -1|5
 STAGE 0 FRAC 0 TO 4 FROM -1|6
 STAGE 0 FRAC 1 TO 4 FROM -1|7
# reduce_scatter step, group 1
 STAGE 1 FRAC 6 TO 5 3 FROM 5|0 4|1
 STAGE 1 FRAC 7 TO 5 3 FROM 5|1 4|2
 STAGE 1 FRAC 4 TO 5 3 FROM 5|2 4|3
 STAGE 1 FRAC 5 TO 5 3 FROM 5|3 4|4
# 1st allgatherv step, group 2
 STAGE 2 FRAC 6 TO 5 FROM 5|0
 STAGE 2 FRAC 7 TO 5 FROM 5|1
 STAGE 2 FRAC 4 TO 5 FROM 5|2
 STAGE 2 FRAC 5 TO 5 FROM 5|3
 STAGE 2 FRAC 7 TO 5 1 FROM 7|0 5|1
 STAGE 2 FRAC 4 TO 5 1 FROM 7|1 5|2
 STAGE 2 FRAC 5 TO 5 1 FROM 7|2 5|3
 STAGE 2 FRAC 6 TO 5 1 FROM 7|3 5|0
# 2nd allgatherv step, group 2
 STAGE 3 FRAC 6 TO 5 FROM 5|0
 STAGE 3 FRAC 7 TO 5 FROM 5|1
 STAGE 3 FRAC 4 TO 5 FROM 5|2
 STAGE 3 FRAC 5 TO 5 FROM 5|3
 STAGE 3 FRAC 7 TO 5 FROM 5|4
 STAGE 3 FRAC 4 TO 5 FROM 5|5
 STAGE 3 FRAC 5 TO 5 FROM 5|6
 STAGE 3 FRAC 6 TO 5 FROM 5|7
 STAGE 3 FRAC 5 TO 5 4 FROM 1|4 5|6
 STAGE 3 FRAC 6 TO 5 4 FROM 1|5 5|7
 STAGE 3 FRAC 7 TO 5 4 FROM 1|6 5|4
 STAGE 3 FRAC 4 TO 5 4 FROM 1|7 5|5
# 3rd allgatherv step, group 3
 STAGE 4 FRAC 6 TO 5 FROM 5|0
 STAGE 4 FRAC 7 TO 5 FROM 5|1
 STAGE 4 FRAC 4 TO 5 FROM 5|2
 STAGE 4 FRAC 5 TO 5 FROM 5|3
 STAGE 4 FRAC 7 TO 5 FROM 5|4
 STAGE 4 FRAC 4 TO 5 FROM 5|5
 STAGE 4 FRAC 5 TO 5 FROM 5|6
 STAGE 4 FRAC 6 TO 5 FROM 5|7
 STAGE 4 FRAC 5 TO 5 -1 FROM 5|8  # result
 STAGE 4 FRAC 6 TO 5 -1 FROM 5|9  # result
 STAGE 4 FRAC 7 TO 5 -1 FROM 5|10 # result
 STAGE 4 FRAC 4 TO 5 -1 FROM 5|11 # result
 STAGE 4 FRAC 1 TO 5 -1 FROM 4|8  # result
 STAGE 4 FRAC 2 TO 5 -1 FROM 4|9  # result
 STAGE 4 FRAC 3 TO 5 -1 FROM 4|10 # result
 STAGE 4 FRAC 0 TO 5 -1 FROM 4|11 # result
```

Figure 9: Allreduce with groups of partial `reduce_scatter` and `allgatherv`

simulate only one). The deviation from the solution of variable message sizes is minor and could only become relevant for very big data types (Sec. 2.2.3).

For the decomposition into groups we apply the following heuristic: either no decomposition is done — all ranks are considered as one group — or the number of ranks is decomposed into prime numbers, where prime numbers smaller than the used numbers of ports plus one are combined together. For medium message sizes the initial partial `reduce_scatter` phase is done with a number of ports equal to the square root of the number of nodes minus one if possible.

4. Implementation details

The separation of the initialisation phase of the algorithms from the actual communication is beneficial, since a significant amount of computation has to be done in order to determine the parameters, algorithms, single step message sizes and communicating ranks. The execution time of the initialisation is approximately independent from the message size, and therefore not negligible especially for short messages. The cost of initialisation is amortised by repeated calls of the execution routines which are highly optimised. We have chosen to encode the whole algorithm in a special bytecode in the initialisation phase, without any ifs/jumps [14]. In the execution phase this bytecode is interpreted. We have many algorithmic choices in the code generation phase, without disadvantage in the execution phase.

The intermediate step of the mini language presented in Sec. 2.3 is translated into an assembler code which is optimised and further translated into the byte code. Up to the generation of the byte code, the algorithmic part of the code does not use any MPI.

Our collective communication routines are based on the MPI point-to-point communication routines `MPI_Irecv`, `MPI_Isend`, `MPI_Waitall`. For the execution of the byte code no MPI communicator besides `MPI_COMM_WORLD` is needed. Since the algorithms are purely deterministic, numerical results of the reductions are bit-reproducible.

At the node level some optimisations are made. The local reduction on the node is done with two different algorithms. For short messages each task copies its data to the shared memory segment, a barrier is called, and every rank reduces a chunk of data in a loop over all segments copied in from the different tasks. In case of long messages every task copies in a chunk of its data such that the shared memory buffer is completely initialised by all tasks. Then, in a loop, a barrier is called and the next chunk is not just copied in but combined with the existing data with the reduction operation. The first option requires the data to be touched twice and one barrier, while the second option requires only one touch of the data and many barriers. In order to avoid a further barrier which would be required at the beginning of the collective or at the end of it, since the shared memory buffer might be used for a copyout operation of the first call of the collective and the copyin operation of the second consecutive call of the collective at the same time, for short messages we allocate two buffers and use them alternatingly.

Current limitations of the implementation are the following: The same number of tasks must be involved in the collective communication on all nodes. In the automatic determination of the algorithmic parameters, no trade-off between performance and memory usage is implemented. Only intra-node communicators [43] are implemented and only contiguous data types are supported.

It is not necessary to rewrite our routines for equal message sizes, `allgather` and `reduce_scatter_block`, since they would not perform better than the versions for non-equal message sizes, but wrappers might be convenient for this feature. Furthermore, the operations `bcast` and `reduce` are covered by setting up `allgatherv` and `reduce_scatter`, respectively, with all message sizes equal to zero except of one. Then our algorithms simplify to a tree algorithm without explicit implementation. We follow the approach of the libNBC library [44] and implement the non-blocking feature with a progress routine. All MPI tasks on the node have to call this routine. This can be realised with an additional OpenMP helper thread per task.

The source code of our collective communication implementation will be made publicly available on github if the contribution will be accepted.

5. Benchmarks

Benchmarks are made on an empty Cray XC40 KNL cluster comprising 64-core Intel Xeon Phi CPU 7230 processors running at 1.30GHz. The processors are configured in flat memory mode with quadrant clustering using MCDRAM. The network topology is Dragonfly with Aries routing. Furthermore we use a HPE Cray EX system with Slingshot network and AMD EPYC 7742 with 64 cores at 2.25GHz with low back-load on the network.

We mostly follow the OSU microbenchmarks [45], which were adapted for our communication routines. Our routines and the reference routines are called alternatingly in order to minimise any effects of network back-load. Figure 10 (left) shows the communication time in relation to the message size for our persistent `allgatherv` routine and for the non-persistent MPI `allgatherv` routine for 1920 tasks (cores) on 160 nodes. The 12 cores per KNL were chosen in order to mimic our production system Piz Daint with 12 cores per node. The message sizes refer to the message of the send buffer before communication. Our routine is faster than the one of Cray MPI (cray-mpich/7.7.16) on the Aries and the Slingshot network especially for small message sizes. Figure 10 (right) shows the same properties for `reduce_scatter`. Contrary to the definition of the OSU microbenchmarks, the message size refers to the message in the receive buffer after communication. The speedup of our routine compared to Cray MPI and to OpenMPI (4.1.0) is significant. We believe that, besides our algorithmic improvements implemented for the communication, also the local reduction on the node is done more efficiently in our routines. Figure 11 shows the results for `allreduce`. Table 2 shows the number of ports and the factorisation for different message sizes on the Aries network. Positive numbers of ports indicate `allgatherv` steps, negative ones `reduce_scatter` steps. For every factor its corresponding group is given. For short messages up to 8192 bytes `allreduce` is the pure `allgatherv` cyclic shift algorithm with selected lines. The maximum number of ports is used. From 16384 bytes on, the algorithm switches to a `reduce_scatter` followed by `allgatherv`. With increasing message size, more steps are performed with less ports. For 33554432 bytes only one port is used, except for the step in the middle. A special case is 262144 bytes with a partial `reduce_scatter` followed by `allgatherv`, where not groups of 160 nodes are used, but the factors 16 and 10 (Sec. 2.4).

size/bytes	factors(numbers of ports)	time/μs
8	160(12 12)	$8.211 \cdot 10^1$
⋮		
8192	160(12 12)	$2.728 \cdot 10^2$
16384	160(-12 -12) 160(12 12)	$2.997 \cdot 10^2$
⋮		
131072	160(-12 -12) 160(12 12)	$6.314 \cdot 10^2$
262144	16(-3 -3) 10(9) 16(3 3)	$8.627 \cdot 10^2$
524288	160(-2 -4 -10) 160(10 4 2)	$1.217 \cdot 10^3$
1048576	160(-2 -5 -8) 160(8 5 2)	$1.857 \cdot 10^3$
2097152	160(-1 -2 -2 -8) 160(8 2 2 1)	$3.149 \cdot 10^3$
4194304	160(-1 -1 -1 -2 -6) 160(6 2 1 1 1)	$5.677 \cdot 10^3$
8388608	160(-1 -1 -1 -1 -1 -4) 160(4 1 1 1 1 1)	$1.166 \cdot 10^4$
16777216	160(-1 -1 -1 -1 -1 -5) 160(5 1 1 1 1 1)	$1.997 \cdot 10^4$
33554432	160(-1 -1 -1 -1 -1 -1 -2) 160(2 1 1 1 1 1 1)	$4.185 \cdot 10^4$

Table 2: Numbers of ports in groups (factors) for `allreduce` with different message sizes and 160 nodes (Aries network), phase (I), phase (II), phase (III)

Figure 12 shows the performance of `allreduce` for a varying number of nodes on the Aries and Slingshot network, with a fixed message size. Our results are at least comparable with the speedups exploiting shared memory nodes shown in the literature. However, the advantage of our algorithms becomes also visible for one task (core) per node only. Figure 13 shows the performance of `allreduce` for one task per node only, for one task per node using one port, or at maximum 12 ports calling multiple non-blocking point-to-point communications at the same time (over-subscription). The latter option is most efficient. The better performance for medium message sizes shows the strength of our approach in particular. While for short messages the cancellation of lines brings only small advantages, for larger message sizes it becomes relevant and extends the range of the pure `allgatherv` and partial `reduce_scatter` with `allgatherv` towards longer messages. We note that for a complete `reduce_scatter` with `allgatherv`, line cancellation does not apply. Also the varying number of ports has its strength for medium message sizes while for very short and very long messages it degenerates to constant numbers. Speedups for non-equal message sizes are shown in Sec. 6.

We do not show the results for the experimental implementation of persistent collectives in OpenMPI since their performance is lower than the corresponding non-persistent one. Our algorithms can also be applied to non-persistent collective communication especially for longer messages where the initialisation time is not that significant in relation to the whole execu-

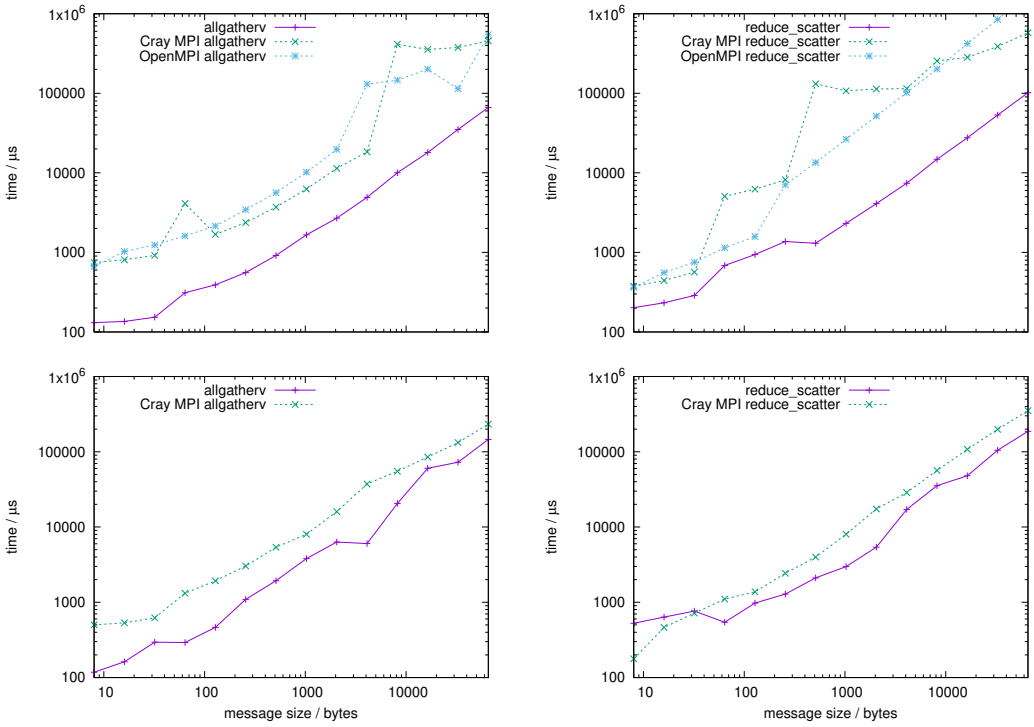

Figure 10: Allgatherv (left) and Reduce_scatter (right) on 160 nodes with 1920 tasks Aries (top) and Slingshot (bottom)

tion time. However, we believe that for all optimisation features used, the persistence is essential. Our approach of the mini language makes the implementation more convenient but leads to an additional overhead. Thus we claim that the implementations of non-persistent and persistent collectives might differ significantly, although the functionality is very similar.

Our routines outperform Cray MPI and OpenMPI in all cases except at message sizes of 4096 bytes for multiple tasks per node, where the Cray MPI on the Slingshot network is superior (Fig. 12 bottom, middle). Since for one task per node our routine outperforms CrayMPI in that case (not shown, except the 160 nodes datapoint in Fig. 13) we assume that the optimisations for the initial reduction at the node level of CrayMPI are more sophisticated than ours, in particular the cache usage. The same explanation applies to the outliers for small messages in Fig. 10 (bottom, right) and Fig. 11 (bottom). The peaks in all graphs especially in the ones for small messages sizes are intermittent slowdowns of the systems and not caused by the algorithms, despite two seconds measurement time per datapoint.

Finally, it must be noted that the fixed order of operations for the reductions could be relaxed taking into account the message arrival patterns and giving up bit-reproducibility. This would allow our implementation to achieve further speedups.

6. Fourier filter

The optimised `allgatherv` and `reduce_scatter` routines are applied to a Fourier filter which is part of the plasma physics application ORB5 [17]. Its task is to transform data on a regular 3D mesh which is periodic in two directions from real space to spectral space in these two periodic directions and to select a fraction of modes to be processed further. The reverse spectral space to real space transformation is also part of the procedure. The data arrangement of the code is the following. The application uses a toroidal computational domain, for parallelisation a 1D domain decomposition in toroidal direction, and an additional domain cloning technique. The filter reduces the number of Fourier modes to a band in poloidal-toroidal mode numbers. For general configurations the number of Fourier modes processed further in the field solver is not a multiple of the number of nodes allocated, the messages have non-equal size. It might even happen that part of the nodes are idling during the field solve procedure and will either receive or send messages only. The poloidal (m) and toroidal (n) Fourier mode numbers that are retained in the filter are such that $m \in [nq(r) - \Delta m, nq(r) + \Delta m]$, with $q(r)$ a given function and Δm a user-chosen parameter. Table 3 illustrates an example of the Fourier table (n, m) for the surface $q(r) = 2$ and $\Delta m = 2$. The

$m \backslash n$	-2	-1	0	1	2	3	4	5	6	7	8
0	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	0	0	0	0	0	0
1	0	0	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$	0	0	0	0
2	0	0	0	0	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$	$a_{3,8}$	$a_{3,9}$	0	0
3	0	0	0	0	0	0	$a_{4,7}$	$a_{4,8}$	$a_{4,9}$	$a_{4,10}$	$a_{4,11}$

Table 3: Example of toroidal (n) and poloidal (m) Fourier modes retained in the filter on a given radial surface, here $q(r) = 2$ and $\Delta m = 2$.

filter varies in radial direction. Two options are implemented in the code, a solution with Fast Fourier Transforms (FFTs) and one with a DFT matrix, other ones are possible.

In the DFT matrix approach the real space vector $\mathbf{r}$ is trans-

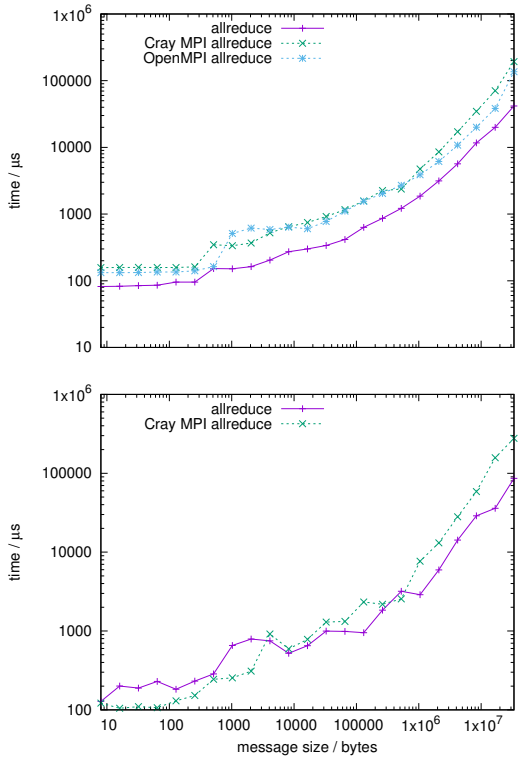

Figure 11: Allreduce on 160 nodes with 1920 tasks Aries (top) and Slingshot (bottom)

formed to the spectral space vector $\mathbf{s}$ by multiplying with the matrix $\mathbf{F}$.

$$\mathbf{s} = \mathbf{F}\mathbf{r} \tag{4}$$

This operation with N^2 complexity is efficient in our case since the transformation matrix $\mathbf{F}$ is very sparse. Here the start is a FFT in poloidal direction followed by the matrix-vector multiplication. Thus the matrix

$$\mathbf{F} = \begin{bmatrix} & 0 & & \\ \hline \omega_N^{l\cdot 0} & \omega_N^{l\cdot 1} & \dots & \omega_N^{l\cdot(N-1)} \\ \vdots & \vdots & & \vdots \\ \omega_N^{m\cdot 0} & \omega_N^{m\cdot 1} & \dots & \omega_N^{m\cdot(N-1)} \\ \hline & 0 & & \end{bmatrix}, \quad \omega_N = e^{-i2\pi/N} \tag{5}$$

transforms a single line in toroidal direction. Only the values necessary to be computed are communicated. The computation and communication from $\mathbf{r}$, which is distributed over the nodes, to $\mathbf{s}$ is done such that $\mathbf{s}$ is distributed as equal as possible over the nodes. For the backward transformation the reverse operations apply.

Benchmarks are performed for a simplified version of the plasma physics application [46]. Application parameters are $n_\phi = 512$, $n_\theta = 1024$, $n_r = 512$ in toroidal, poloidal, and radial direction, respectively, with 12 clones and a B-spline order of 2. For every timestep 4 substeps of a Runge-Kutta algorithm time-advancing scheme are required, each of these applying exactly the same Fourier filtering, so 4 executions of these operations (from real space to spectral space, Fourier filtering, and back from spectral space to real space) are performed per step. The parameter $\Delta m = 5$ is chosen, while only two Fourier modes are kept in the toroidal direction.

As a result, two messages of length 90464 bytes and all other messages of length zero bytes are gathered and distributed to all nodes participating and the reverse is done for the reduction. For those parameters the benchmarks of the `allgatherv` and the `reduce_scatter` routines are carried out. Figure 14 shows their performance per single call in comparison to the Cray MPI reference implementation. In order to quantify the effect of rank reordering we included graphs (Fig. 14) for a worst case ordering, messages sorted according to size. For the Cray MPI reference implementation the rank orders are chosen randomly.

7. Conclusions

In this paper, we optimised the persistent collective communication operations `allgatherv`, `reduce_scatter`, and `allreduce`. The initialisation phase allowed for several optimisations, namely an extensive choice of algorithms, such as a recursive application of cyclic shifting (Bruck's algorithm), with different number of ports/substeps for different steps. The proper algorithms and their parameters are chosen according to network performance measurements at the installation time of the library. For `allgatherv` and `reduce_scatter`, we considered explicitly the occurrence of non-equal message sizes in our algorithms with a rank reordering heuristic. Our `allreduce` for small messages is based on Bruck's `allgather` algorithm with a prefix operation, where we delete lines not needed, whereas for long messages our optimised `reduce_scatter` and `allgatherv` are consecutively called. Medium message sizes are covered by an incomplete `reduce_scatter` followed by `allgatherv`, where again line cancellation is applied.

The existing implementations of Cray MPI and OpenMPI are outperformed significantly for small and medium message sizes for `allgatherv`. Although our `reduce_scatter` is slower than the reference for small messages on the Slingshot network and a small number of nodes, it outperforms the reference clearly for all other cases. Our `allreduce` is faster than the existing implementation again except for 4096 byte messages on the Slingshot network.

For non-equal message sizes, our routines show additional speedups if the ranks are reordered.

Acknowledgements

The authors would like to thank Maria-Grazia Giuffreda, Timothy W. Robinson, and Jean-Guillaume Piccinali (CSCS) for helpful discussions. Furthermore we would like to thank the anonymous reviewers for their constructive remarks.

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 and 2019-2020 under grant agreement No 633053. The views and

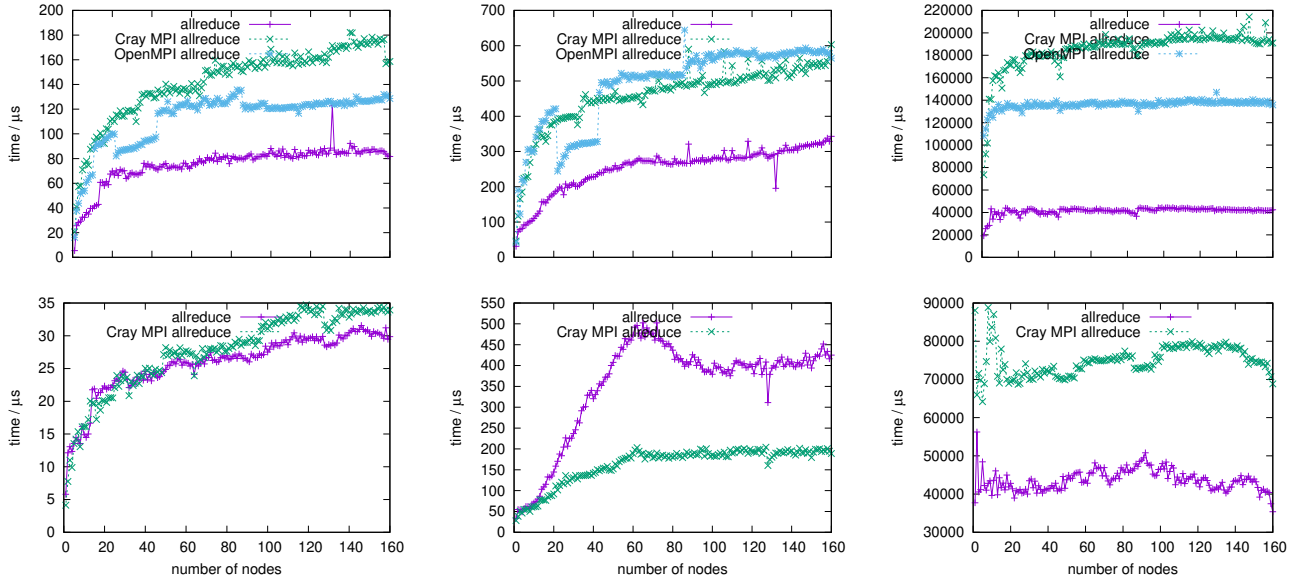

Figure 12: Allreduce, Aries network (top) and Slingshot network (bottom) with 8 bytes (left) 4096 bytes (middle), and 33554432 bytes (right), 12 tasks per node

opinions expressed herein do not necessarily reflect those of the European Commission. This work was partly supported by the Swiss National Science Foundation.

References

[1] W. Gropp, E. Lusk, A. Skjellum, Using MPI: portable parallel programming with the message-passing interface, Vol. 1, MIT press, 1999.

[2] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in MPICH, The International Journal of High Performance Computing Applications 19 (1) (2005) 49–66.

[3] J. Bruck, C.-T. Ho, Efficient global combine operations in multi-port message-passing systems, Parallel Processing Letters 3 (04) (1993) 335–346.

[4] T. Hoefler, T. Schneider, Optimization principles for collective neighborhood communications, in: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE, 2012, pp. 1–10.

[5] Message Passing Interface Forum, MPI: A Message-Passing Interface standard, version 4.0 (2019).
URL `www.mpi-forum.org/mpi-40`

[6] D. J. Holmes, B. Morgan, A. Skjellum, P. V. Bangalore, S. Sridharan, Planning for performance: Enhancing achievable performance for MPI through persistent collective operations, Parallel Computing 81 (2019) 32–57.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. Von Eicken, LogP: Towards a realistic model of parallel computation, in: ACM Sigplan Notices, Vol. 28, ACM, 1993, pp. 1–12.

[8] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, D. Weathersby, Efficient algorithms for all-to-all communications in multiport message-passing systems, IEEE Transactions on parallel and distributed systems 8 (11) (1997) 1143–1156.

[9] V. Tipparaju, J. Nieplocha, D. Panda, Fast collective operations using shared and remote memory access protocols on clusters, in: Proceedings International Parallel and Distributed Processing Symposium, IEEE, 2003, pp. 10–pp.

[10] B. Tu, M. Zou, J. Zhan, X. Zhao, J. Fan, Multi-core aware optimization for MPI collectives, in: 2008 IEEE International Conference on Cluster Computing, IEEE, 2008, pp. 322–325.

[11] S. Li, T. Hoefler, C. Hu, M. Snir, Improved MPI collectives for MPI processes in shared address spaces, Cluster computing 17 (4) (2014) 1139–1155.

[12] A. Venkatesh, S. Potluri, R. Rajachandrasekar, M. Luo, K. Hamidouche, D. K. Panda, High performance alltoall and allgather designs for infiniband MIC clusters, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE, 2014, pp. 637–646.

[13] S. Li, Y. Zhang, T. Hoefler, Cache-oblivious MPI all-to-all communications based on morton order, IEEE Trans. Parall. and Distr. Syst. 29 (3) (2018) 542–555.

[14] A. Jocksch, M. Kraushaar, D. Daverio, Optimized all-to-all communication on multicore architectures applied to FFTs with pencil decomposition, Concurr. Comp.-Pract. E. (2018) e4964.

[15] M. Bayatpour, J. Hashmi, S. Chakraborty, H. Subramoni, P. Kousha, D. Panda, SALaR: Scalable and adaptive designs for large message reduction collectives, 2018.

[16] M. Ruefenacht, M. Bull, S. Booth, Generalisation of recursive doubling for allreduce: Now with simulation, Parallel Comput. 69 (2017) 24–44.

[17] S. Jolliet, A. Bottino, P. Angelino, R. Hatzky, T.-M. Tran, B. Mcmillan, O. Sauter, K. Appert, Y. Idomura, L. Villard, A global collisionless PIC code in magnetic coordinates, Comput. Phys. Commun. 177 (5) (2007) 409–425.

[18] E. Lanti, N. Ohana, N. Tronko, T. Hayward-Schneider, A. Bottino, B. McMillan, A. Mishchenko, A. Scheinberg, A. Biancalani, P. Angelino, S. Brunner, J. Dominski, P. Donnel, C. Gheller, R. Hatzky, A. Jocksch, S. Jolliet, Z. Lu, J. Martin Collar, I. Novikau, E. Sonnendrücker, T. Vernay, L. Villard, ORB5: A global electromagnetic gyrokinetic code using the PIC approach in toroidal geometry, Computer Physics Communications 251 (2020) 107072. doi:https://doi.org/10.1016/j.cpc.2019.107072.

[19] R. Rabenseifner, J. L. Träff, More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems, in: D. Kranzlmüller, P. Kacsuk, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 36–46.

[20] P. Sack, W. Gropp, Faster topology-aware collective algorithms through non-minimal communication, SIGPLAN Not. 47 (8) (2012) 45–54. doi:10.1145/2370036.2145823.

[21] V. End, C. Simmendinger, R. Yahyapour, T. Alrutz, Butterfly-like algorithms for GASPI split-phase allreduce, International Journal on Advances in Systems and Measurements 9 (2016).

[22] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, Y. Zheng, Optimization of MPI collective communication on BlueGene/L systems, in: Proceedings of the

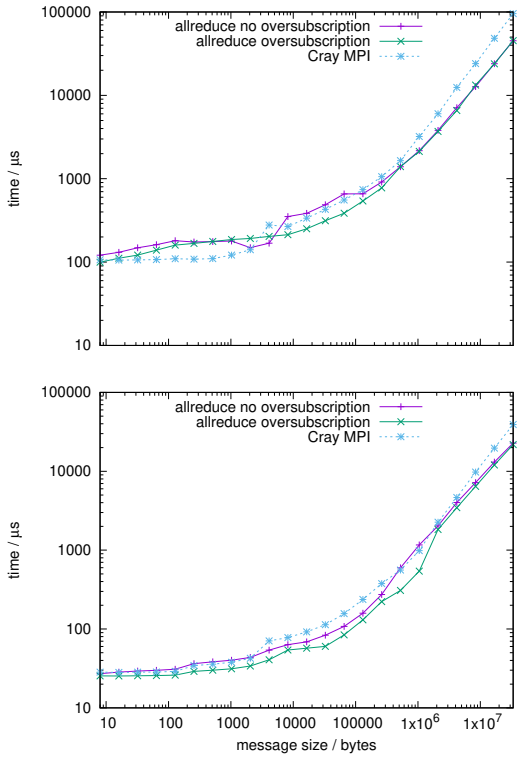

Figure 13: Allreduce for varying message size on 160 nodes with 1 task per node, Aries (top) and Slingshot (bottom)

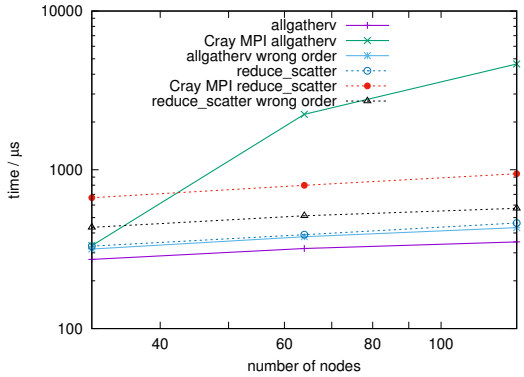

Figure 14: Execution times of `allgatherv` and `reduce_scatter`, two times 90464 bytes the rest zero bytes, rank reordering, reference and without rank reordering, 1 task per node (Aries)

19th annual international conference on Supercomputing, ACM, 2005, pp. 253–262.

[23] S. Chakraborty, H. Subramoni, D. K. Panda, Contention-aware kernel-assisted MPI collectives for multi-/many-core systems, in: 2017 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2017, pp. 13–24.

[24] E. Chan, M. Heimlich, A. Purkayastha, R. Van De Geijn, Collective communication: theory, practice, and experience, Concurr. Comp.-Pract. E. 19 (13) (2007) 1749–1783.

[25] A. Faraj, X. Yuan, Automatic generation and tuning of MPI collective communication routines, in: Proceedings of the 19th annual international conference on Supercomputing, ACM, 2005, pp. 393–402.

[26] R. L. Graham, G. Shipman, MPI support for multi-core architectures: Optimized shared memory collectives, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2008, pp. 130–140.

[27] A. Karwande, X. Yuan, D. K. Lowenthal, CC–MPI: a compiled communication capable MPI prototype for ethernet switched clusters, in: ACM Sigplan Notices, Vol. 38, ACM, 2003, pp. 95–106.

[28] P. Patarasuk, X. Yuan, Bandwidth optimal all-reduce algorithms for clusters of workstations, Journal of Parallel and Distributed Computing 69 (2) (2009) 117–124.

[29] A. Bienz, L. Olson, W. Gropp, Node-aware improvements to allreduce, in: 2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI), IEEE, 2019, pp. 19–28.

[30] H. Zhou, J. Gracia, N. Zhou, R. Schneider, Collectives in hybrid MPI+ MPI code: Design, practice and performance, Parallel Computing 99 (2020) 102669.

[31] J. L. Träff, S. Hunold, Decomposing MPI collectives for exploiting multi-lane communication, in: 2020 IEEE International Conference on Cluster Computing (CLUSTER), 2020, pp. 270–280. doi:10.1109/CLUSTER49012.2020.00037.

[32] S. Bouhrour, J. Jaeger, Implementation and performance evaluation of MPI persistent collectives in MPC: A case study, in: 27th European MPI Users' Group Meeting, EuroMPI/USA '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 51–60. doi:10.1145/3416315.3416321.

[33] A. Jocksch, N. Ohana, E. Lanti, V. Karakasis, L. Villard, Optimised allgatherv, reduce_scatter and allreduce communication in message-passing systems (2020). arXiv:2006.13112.

[34] A. Jocksch, N. Ohana, E. Lanti, V. Karakasis, L. Villard, Towards an optimal allreduce communication in message-passing systems, in: EuroMPI/USA, 2020.

[35] Y. Qian, A. Afsahi, High performance RDMA-based multi-port all-gather on multi-rail QsNet II, in: 21st International Symposium on High Performance Computing Systems and Applications (HPCS'07), IEEE, 2007, pp. 3–3.

[36] J. L. Träff, An improved algorithm for (non-commutative) reduce-scatter with an application, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2005, pp. 129–137.

[37] M. Bernaschi, G. Iannello, M. Lauria, Efficient implementation of reduce-scatter in MPI, Journal of Systems Architecture 49 (3) (2003) 89–108.

[38] D. Kolmakov, X. Zhang, A generalization of the allreduce operation (2020). arXiv:2004.09362.

[39] M. Jeon, D. Kim, Parallel merge sort with load balancing, International Journal of Parallel Programming 31 (1) (2003) 21–33.

[40] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, J. L. Träff, MPI on a million processors, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2009, pp. 20–30.

[41] S. Chunduri, T. Groves, P. Mendygral, B. Austin, J. Balma, K. Kandalla, K. Kumaran, G. Lockwood, S. Parker, S. Warren, N. Wichmann, N. Wright, GPCNeT: Designing a benchmark suite for inducing and measuring contention in HPC networks, in: Proc. Int. Conf. High Performance Computing, Networking, Storage, and Analysis, SC'19, Argone National Laboratory, November 2019.

[42] B. S. Parsons, Accelerating MPI collective communications through hierarchical algorithms with flexible inter-node communication and imbalance awareness, Ph.D. thesis, Perdue University (2015).

[43] Q. Kang, A. Agrawal, A. Choudhary, W.-k. Liao, Optimal algorithms for half-duplex inter-group all-to-all broadcast on fully connected and ring topologies, in: SC'18: Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis, IEEE, 2018.

[44] T. Hoefler, A. Lumsdaine, W. Rehm, Implementation and performance analysis of non-blocking collective operations for MPI, in: SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, IEEE, 2007, pp. 1–10.

[45] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, D. Panda, OMB-GPU: A micro-benchmark suite for evaluating MPI libraries on GPU clusters, 2012.

[46] N. Ohana, A. Jocksch, E. Lanti, T. Tran, S. Brunner, C. Gheller, F. Hariri, L. Villard, Towards the optimization of a gyrokinetic particle-in-cell (PIC) code on large-scale hybrid architectures, in: Journal of Physics: Conference Series, Vol. 775, IOP Publishing, 2016, p. 012010.