Towards an optimal allreduce communication in message-passing systems

Andreas Jocksch
andreas.jocksch@cscs.ch
CSCS, Swiss National Supercomputing Centre,
Via Trevano 131
CH-6900 Lugano, Switzerland

Vasileios Karakasis
CSCS, Swiss National Supercomputing Centre,
Via Trevano 131
CH-6900 Lugano, Switzerland

No'e Ohana
Emmanuel Lanti
Ecole Polytechnique Fédérale de Lausanne (EPFL),
Swiss Plasma Center (SPC)
CH-1015 Lausanne, Switzerland

Laurent Villard
Ecole Polytechnique Fédérale de Lausanne (EPFL),
Swiss Plasma Center (SPC)
CH-1015 Lausanne, Switzerland

1 INTRODUCTION

In this contribution we introduce optimised collective communication for the pattern `allreduce`, with the additional initialisation phase, like for persistent collective communication [5, 6]. Without restriction of the generality of our approach, our `allreduce` is blocking.

More specifically, we generalise the cyclic shift algorithm (Bruck's algorithm [3]) to allow for different factors for different steps. In this way the underlying algorithms are adjusted for the particular network based on measurements. Large message sizes are handled with `reduce_scatter` and `allgatherv` based on cyclic shift.

For small message sizes, we generalise for `allreduce` the prime factor decomposition for recursive multiplying [9] to a factorisation with multiple consecutive calls of Bruck's algorithm (Fig. 1). The prime factors are combined using a greedy approach. Furthermore, for the case of small messages, Bruck's algorithm is further formalised by storing the results of the prefix operation instead of the actual data.

Together with the shared memory optimisation on the nodes, the procedures mentioned provide the majority of the performance improvement observed for the different cases. Non-persistent MPI implementations are mostly outperformed on the Cray XC40 and on an Infiniband cluster.

2 THE ALGORITHMS

For long messages we apply `reduce_scatter` followed by `allgatherv` (algorithm II in Ref. [2]) while these algorithms (Sec. 2.1) are implemented for non-equal message sizes [7]. In contrast to that, for short messages a variant of allgather (algorithm I in Ref. [2], Sec. 2.2) is utilised.

For all algorithms, 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 `allreduce` algorithm, and (III) distribution of the data to all tasks on the node locally.

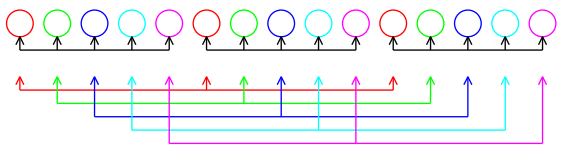

Figure 1: Independent cyclic shifts with prime factors 5 and 3

2.1 Reduce_scatter and allgatherv

We apply the cyclic shift algorithm [3]. In difference to the naive algorithm which sends all information directly, this algorithm does not send the information directly from the source to the destination rank but applies forwarding. Thus the amount of data sent through the network remains unchanged, but the number of communication steps is reduced. We speak about factors $f_1 \cdot f_2 \cdot \ldots \cdot f_s = p$ as done for recursive exchange in Ref. [9] since the formula with the radix $r^s = p$ is not valid for our general case.

Within the steps the message exchange can be done in parallel with a number of ports equal to the factor minus one. Since the size of the messages shrinks and grows from step to step for `reduce_scatter` and `allgatherv`, respectively, we consider this flexibility as essential and use different factors for the different steps of the algorithms (see Sec. 3).

For `reduce_scatter` we consider the following additional points. In this contribution we take only commutative reduction operations into account (see [10]). The cyclic shift algorithm known from `allgatherv` has also been applied in a similar way to `reduce_scatter` [1]. However, we believe that it should be formalised as the `allgather` algorithm. The `reduce_scatter` operation can be considered as the reversed `allgatherv` operation in the same way as reduce is

EuroMPI '20, September 21–24, 2020, Austin, TX

the reversed operation of broadcast. Therefore the same algorithms are applied in reversed order, as proposed in [2]. There is one major difference, however. While in the `allgatherv` case buffers might be used for sending with multiple MPI processes at the same time, this is only possible with an intra-node reduction for the `reduce_scatter` case. Thus the memory requirement is higher for `reduce_scatter`, since we assume that the receiving rank first gets the data in an empty buffer and second performs the arithmetic operation. As 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`.

2.2 Cyclic shift with prefix operation

We base our allreduce collective operation for short messages on the same algorithm as `allgatherv`. A naive implementation would use the `allgather` algorithm as illustrated in Figure 2 (left) for cyclic shift. We assume that the reduction operation is a sum. Here, we modify the scheme and do not store the values at the lines, but column-wise the partial sum (inclusive scan) from the top to the bottom (Fig. 2 right). While the l'th line shifted by k columns to the left is the $l+k$'th line in the original scheme (Fig. 2 left), 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 $k-1$ (Fig. 2 right).

This formalism allows for less lines to be communicated, since for computing the final result on the bottom line, for $r = 2$ only 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 algorithmic complexity becomes equivalent to the one of the binary exchange algorithm. In the more general case, if always $(f_1 - 1) \cdot (f_2 - 1) \cdot ... \cdot (f_i - 1)$ lines according to the non-optimised algorithm are communicated including the last step, only one line per substep needs to be communicated. This case corresponds to the approach of [9].

Therefore the node count is decomposed in prime factors (Fig. 1). If the prime factors are smaller than a target factor f_i (e.g. $f_i = 13$, for 12 cores per node) they are combined according to a greedy approach. For prime factors much larger than the target factor f_i e.g. $f_i = 167$ we apply cyclic shift operations with multiple steps for every prime factor i.e. two factors 13 for 167.

3 PARAMETRISATION AND IMPLEMENTATION

The simple bandwidth-latency network model does not give any indication which factors f_i to choose for bandwidth dominated communication. Furthermore it is not clear which factors to apply for the specific hardware. In order to choose the optimal parameters we apply a tuning approach (implemented for long messages only). At the installation phase of the library, measurements of communication times are done for different message sizes. Based on that, the factors f_i are chosen. For all possible combinations of factors the communication time is estimated from interpolations of the measurements performed during installation.

The total number of messages communicated between nodes is in any case smaller with our shared memory approach than for a naive implementation, since messages are merged.

Our experiments show that it is efficient to apply high and low numbers of communication factors for relatively short messages and long messages, respectively. This is supported by the findings in [8], where a saturation effect for long messages is described.

Our collective communication routines are based on the MPI point-to-point communication routines. In order to accommodate all optimisations efficiently, we generate a bytecode in the initialisation phase which is interpreted in the execution phase [7]. Since the algorithms are purely deterministic, numerical results of the reductions are bit-reproducible.

4 BENCHMARKS

Benchmarks are made on an empty Cray XC40 KNL cluster comprising 64-core Intel(R) Xeon Phi(TM) CPU 7230 processors running at 1.30GHz. The processors are configured in flat memory mode with quadrant clustering using MCDRAM. Furthermore we use an empty Infiniband cluster using 17 nodes with two 12 core Haswell(TM) E5-2650v.3 2.66GHz CPUs.

We use the OSU microbenchmarks, which were adapted for our communication routines. For our routine, the Cray nodes were used in groups of 12 cores each, forming 5 virtual nodes per KNL. The 12 cores per virtual node were chosen in order to mimic systems with 12 cores per node. Figure 3 shows the results for different messages sizes (left) and for a varying number of nodes with a fixed small (middle) and large (right) message size. Our routine outperforms Cray MPI (cray-mpich/7.7.10) and OpenMPI 4.0.2 in most cases. Figure 4 shows the performance of the routine on the infiniband cluster for varying message sizes. Our routine is mostly faster than MVAPICH 2.2.

Our results are comparable with the speedups of End et al. [4] who have implemented a k-port allreduce and have made a comparison with OpenMPI 1.6.5.

5 CONCLUSIONS

In this paper, we optimised the collective communication operation `allreduce` where we made extensive use of an initialisation phase. The initialisation phase allowed for several optimisations. The proper algorithms and their parameters are chosen according to network performance measurements at the installation time of the library. Our `allreduce` for long messages is based on consecutivee calls of `allgatherv` and `reduce_scatter` while for short messages a prime factor decomposition of the number of nodes with `allgather` is applied.

$n0$	$n1$	$n2$	$n3$	$n4$	$n5$	$n6$	$n7$	$n8$	$n9$	nA	$n0$	$n1$	...	nA
0	1	2	3	4	5	6	7	8	9	A	$\sum_{i=0}^{0} i$	$\sum_{i=1}^{1} i$	...	$\sum_{i=A}^{A} i$
1	2	3	4	5	6	7	8	9	A	0	$\sum_{i=0}^{1} i$	$\sum_{i=1}^{2} i$	...	$\sum_{i=A}^{A} i + \sum_{i=0}^{0} i$
2	3	4	5	6	7	8	9	A	0	1	$\sum_{i=0}^{2} i$	$\sum_{i=1}^{3} i$	...	$\sum_{i=A}^{A} i + \sum_{i=0}^{1} i$
3	4	5	6	7	8	9	A	0	1	2	$\sum_{i=0}^{3} i$	$\sum_{i=1}^{4} i$	...	$\sum_{i=A}^{A} i + \sum_{i=0}^{2} i$
4	5	6	7	8	9	A	0	1	2	3	–	–	...	–
5	6	7	8	9	A	0	1	2	3	4	–	–	...	–
6	7	8	9	A	0	1	2	3	4	5	–	–	...	–
7	8	9	A	0	1	2	3	4	5	6	$\sum_{i=0}^{7} i$	$\sum_{i=1}^{8} i$	...	$\sum_{i=A}^{A} i + \sum_{i=0}^{6} i$
8	9	A	0	1	2	3	4	5	6	7	–	–	...	–
9	A	0	1	2	3	4	5	6	7	8	–	–	...	–
A	0	1	2	3	4	5	6	7	8	9	$\sum_{i=0}^{A} i$	$\sum_{i=1}^{A} i + \sum_{i=0}^{0} i$	...	$\sum_{i=A}^{A} i + \sum_{i=0}^{9} i$

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

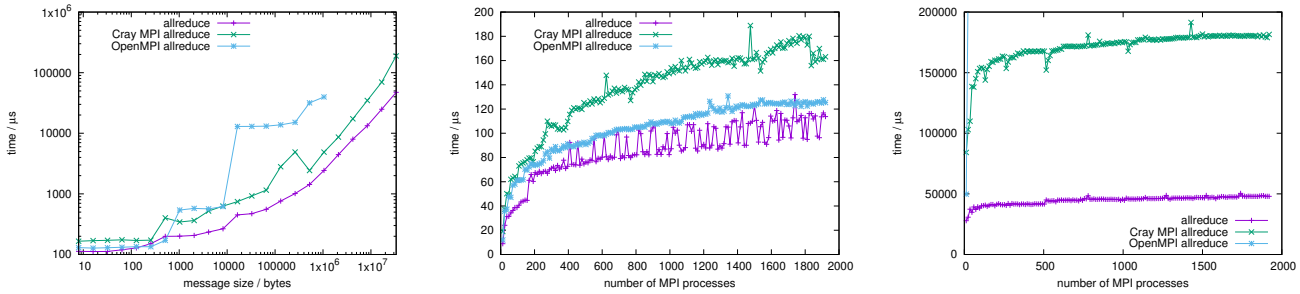

Figure 3: Allreduce on 160 nodes with 9600 MPI processes (left), with 8 bytes (middle) and 33554432 bytes (right) Cray

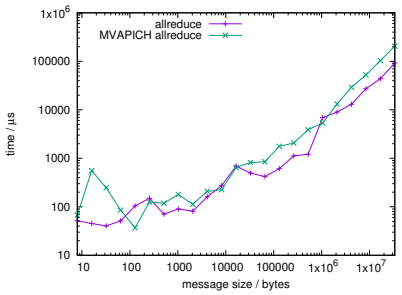

Figure 4: Allreduce on 17 nodes with 408 MPI processes Infiniband

On the XC40 KNL, Cray MPI and OpenMPI are outperformed for long messages and short messages and on the infiniband cluster we mostly outperform MVAPICH.

REFERENCES

[1] Massimo Bernaschi, Giulio Iannello, and Mario Lauria. 2003. Efficient implementation of reduce-scatter in MPI. *Journal of Systems Architecture* 49, 3 (2003), 89–108.

[2] Jehoshua Bruck and Ching-Tien Ho. 1993. Efficient global combine operations in multi-port message-passing systems. *Parallel Processing Letters* 3, 04 (1993), 335–346.

[3] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on parallel and distributed systems* 8, 11 (1997), 1143–1156.

[4] Vanessa End, C Simmendinger, R Yahyapour, and Thomas Alrutz. 2016. Butterfly-like Algorithms for GASPI Split-Phase Allreduce. *International Journal on Advances in Systems and Measurements* 9 (2016).

[5] Torsten Hoefler and Timo Schneider. 2012. Optimization principles for collective neighborhood communications. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.

[6] Daniel J Holmes, Bradley Morgan, Anthony Skjellum, Purushotham V Bangalore, and Srinivas Sridharan. 2019. Planning for performance: Enhancing achievable performance for MPI through persistent collective operations. *Parallel Comput.* 81 (2019), 32–57.

[7] Andreas Jocksch, Noe Ohana, Emmanuel Lanti, Vasileios Karakasis, and Laurent Villard. 2020. Optimised allgatherv, reduce_scatter and allreduce communication in message-passing systems. arXiv:2006.13112 [cs.DC]

[8] Benjamin Scott Parsons. 2015. *Accelerating MPI collective communications through hierarchical algorithms with flexible internode communication and imbalance awareness*. Ph.D. Dissertation. Perdue University.

[9] Martin Ruefenacht, Mark Bull, and Stephen Booth. 2017. Generalisation of recursive doubling for AllReduce: Now with simulation. *Parallel Comput.* 69 (2017), 24–44.

[10] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.