Proceedings of the CUG 2004 Conference, Knoxville, Tennessee, USA, May, 17-21, 2004.
www.cug.org

Collective Reduction Operation on Cray X1 and Other Platforms

Rolf Rabenseifner and Panagiotis Adamidis

High-Performance Computing-Center (HLRS), University of Stuttgart
Allmandring 30, D-70550 Stuttgart, Germany
rabenseifner|adamidis@hlrs.de,
www.hlrs.de/people/rabenseifner/ | adamidis/

Abstract. A 5-year-profiling in production mode at the University of Stuttgart has shown that more than 40% of the execution time of Message Passing Interface (MPI) routines is spent in the collective communication routines MPI_Allreduce and MPI_Reduce. Although MPI implementations are now available for about 10 years and all vendors are committed to this Message Passing Interface standard, the vendors' and publicly available reduction algorithms could be accelerated with new algorithms by a factor between 3 (IBM, sum) and 100 (Cray T3E, maxloc) for long vectors. This paper presents five algorithms optimized for different choices of vector size and number of processes. The focus is on bandwidth dominated protocols for power-of-two and non-power-of-two number of processes, optimizing the load balance in communication and computation. The new algorithms are compared also on the Cray X1 with the current development version of Cray's MPI library (mpt.2.4.0.0.13)

Keywords. Message Passing, MPI, Collective Operations, Reduction.

1 Introduction

MPI_Reduce combines the elements provided in the input vector (buffer) of each process using an operation (e.g. sum, maximum), and returns the combined values in the output buffer of a chosen process named root. MPI_Allreduce is the same as MPI_Reduce, except that the result appears in the receive buffer of all processes.

MPI_Allreduce is one of the most important MPI routines and most vendors are using algorithms that can be improved by a factor of more than 2 for long vectors. Most current implementations are optimized only for short vectors. A 5-year-profiling [12] of most MPI based applications (in production mode) of all users of the Cray T3E 900 at our university has shown, that 8.54 % of the execution time is spent in MPI routines. 37.0 % of the MPI time is spent in MPI_Allreduce and 3.7 % in MPI_Reduce. Based on the profiled number of calls, transferred bytes, and used processes, combined with benchmark results for the vendor's reduction routines and the optimized algorithms, Fig. 1 show that the communication time can be reduced by a factor of 20 % (allreduce) and 54 % (reduce) with the new algorithms. The 5-year-profiling has also shown, that 25 %

of all execution time was spent with a non-power-of-two number of processes. Therefore, a second focus is the optimization for non-power-of-two numbers of processes.

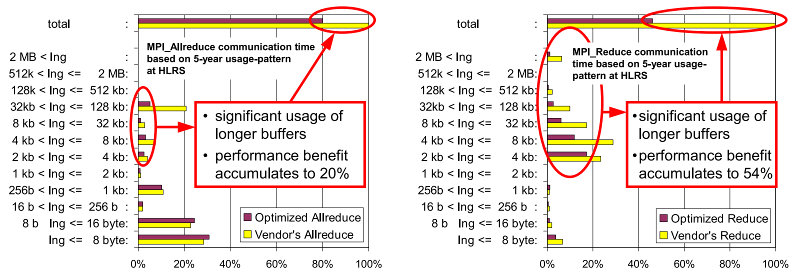

Fig. 1. Benefit of new allreduce and reduce protocols optimized for long vectors.

2 Related Work

Early work on collective communication implements the reduction operation as an inverse broadcast and do not try to optimize the protocols based on different buffer sizes [1]. Other work already handle allreduce as a combination of basic routines, e.g., [2] already proposed the *combine-to-all* (allreduce) as a combination of *distributed combine* (reduce_scatter) and *collect* (allgather). Collective algorithms for wide-area cluster are developed in [5, 7, 8], further protocol tuning can be found in [3, 4, 9, 15], and automatic tuning in [16]. The main focus of the work presented in this paper is to optimize the algorithms for different numbers of processes (non-power-of-two and power-of-two) **and** for different buffer sizes by using special reduce_scatter protocols without the performance penalties on normal rank-ordered scattering. The allgather protocol is chosen according the characteristics of the reduce_scatter part to achieve an optimal bandwidth for any number of processes and buffer size. This paper is based on [13] and extended by benchmark results on Cray X1 parallel shared memory vector systems.

3 Allreduce and Reduce Algorithms

3.1 Cost Model

To compare the algorithms, theoretical cost estimation and benchmark results are used. The cost estimation is based on the same flat model used by R. Thakur and B. Gropp in [15]. Each process has an input vector with n bytes, p is the number of MPI processes, γ the computation cost per vector byte executing one operation with two operands locally on any process. The total reduction effort is

$(p-1)n\gamma$. The total computation time with optimal load balance on p processes is therefore $\frac{p-1}{p}n\gamma$, i.e., less than $n\gamma$, which is independent of the number of processes!

The communication time is modeled as $\alpha + n\beta$, where α is the latency (or startup time) per message, and β is the transfer time per byte, and n the message size in bytes. It is assumed further that all processes can send and receive one message at the same time with this cost model, i.e., p parallel processes can send in parallel p messages each with n bytes (e.g., pairwise or in a ring pattern) with the communication time $\alpha + n\beta$. In reality, most networks are faster, if the processes communicate in parallel, but pairwise only in one direction (uni-directional between two processes), e.g., in the classical binary tree algorithms. Therefore $\alpha_{uni} + n\beta_{uni}$ is modeling the **uni**-directional communication, and $\alpha + n\beta$ is used with the **bi**-directional communication. The ratios are abbreviated with $f_\alpha = \alpha_{uni}/\alpha$ and $f_\beta = \beta_{uni}/\beta$. These factors are normally in the range 0.5 (simplex network) to 1.0 (full duplex network).

3.2 Principles

A classical implementation of MPI_Allreduce is the combination of MPI_Reduce (to a root process) followed by MPI_Bcast sending the result from root to all processes. This implies a bottle-neck on the root process. Also classical is the binary tree implementation of MPI_Reduce, which is a good algorithm for short vectors, but that causes a heavy load imbalance because in each step the number of active processes is halved. The optimized algorithms are based on a few principles:

Recursive vector halving: For long-vector reduction, the vector can be split into two parts and one half is reduced by the process itself and the other half is sent to a neighbor process for reduction. In the next step, again the buffers are halved, and so on.

Recursive vector doubling: To return the total result in the result vector, the split result vectors must be combined recursively. MPI_Allreduce can be implemented as a reduce-scatter (using recursive vector halving) followed by an allgather (using recursive vector doubling).

Recursive distance doubling: In step 1, each process transfers data at distance 1 (process P0 with P1, P2–P3, P4–P5, ...); in step 2, the distance is doubled, i.e., P0–P2 and P1–P3, P4–P6 and P5–P7; and so on until distance $\frac{p}{2}$.

Recursive distance halving: Same procedure, but starting with distance $p/2$, i.e., P0–P$\frac{p}{2}$, P1–P$(\frac{p}{2}+1)$, ..., and ending with distance 1, i.e., P0–P1, ...

Recursive vector and distance doubling and halving can be combined for different purposes, but always additional overhead causes load imbalance if the number of processes is not a power of two. Two principles can reduce the overhead in this case.

Binary blocks: The number of processes can be expressed as a sum of power-of-two values, i.e., all processes are located in subsets with power-of-two processes. Each subset is used to execute parts of the reduction protocol in a block. Overhead occurs in the combining of the blocks in some step of the protocol.

Ring algorithms: A reduce_scatter can be implemented by $p-1$ ring exchange steps with increasing strides. Each process computes all reduction operations for its own chunk of the result vector. In step i (i=1 .. p-1) each process sends the input vector chunk needed by $rank+i$ to that process and receives from $rank-i$ the data needed to reduce its own chunk. The allreduce can be completed by an allgather that is also implemented with ring exchange steps, but with constant stride 1. Each process sends its chunk of the result vector around the ring to the right ($rank+1$) until its left neighbor ($(rank+p-1) \bmod p$) has received it after $p-1$ steps. The following sections describe the algorithms in detail.

3.3 Binary Tree

Reduce: The classical binary tree always exchanges full vectors, uses recursive distance doubling, but with incomplete protocol, because in each step, half of the processes finish their work. It takes $\lceil \lg p \rceil$ steps and the time taken by this algorithm is $T_{red,tree} = \lceil \lg p \rceil (\alpha_{uni} + n\beta_{uni} + n\gamma))$.

For short vectors, this algorithm is optimal (compared to the following algorithms) due to its smallest latency term $\lceil \lg p \rceil \alpha_{uni}$.

Allreduce: The reduce algorithm is followed by a binary tree based broadcast. The total execution time is $T_{all,tree} = \lceil \lg p \rceil (2\alpha_{uni} + 2n\beta_{uni} + n\gamma))$.

3.4 Recursive Doubling

Allreduce: This algorithm is an optimization especially for short vectors. In each step of the recursive distance doubling, both processes in a pair exchange the input vector (in step 1) or its intermediate result vector (in steps 2 ... $\lceil \lg p \rceil$) with its partner process and both processes are computing the same reduction redundantly. After $\lceil \lg p \rceil$ steps, the identical result vector is available in all processes. It needs $T_{all,r.d.} = \lceil \lg p \rceil (\alpha + n\beta + n\gamma)) +$(if non-power-of-two $\alpha_{uni} + n\beta_{uni}$) This algorithm is in most cases optimal for short vectors.

3.5 Recursive Halving and Doubling

This algorithm is a combination of a reduce_scatter implemented with recursive vector halving and distance doubling[1] followed by a allgather implemented by a recursive vector doubling combined with recursive distance halving (for allreduce), or followed by gather implemented with a binary tree (for reduce).

In a first step, the number of processes p is reduced to a power-of-two value: $p' = 2^{\lfloor \lg p \rfloor}$. $r = p - p'$ is the number of processes that must be *removed* in this first step. The first $2r$ processes send pairwise from each even $rank$ to the odd

[1] A distance doubling (starting with distance 1) is used in contrary to the reduce_scatter algorithm in [15] that must use a distance halving (i.e., starting with distance $\frac{\#\text{processes}}{2}$) to guarantee a rank-ordered scatter. In our algorithm, any order of the scattered data is allowed, and therefore, the longest vectors can be exchanged with the nearest neighbor, which is an additional advantage on systems with a hierarchical network structure.

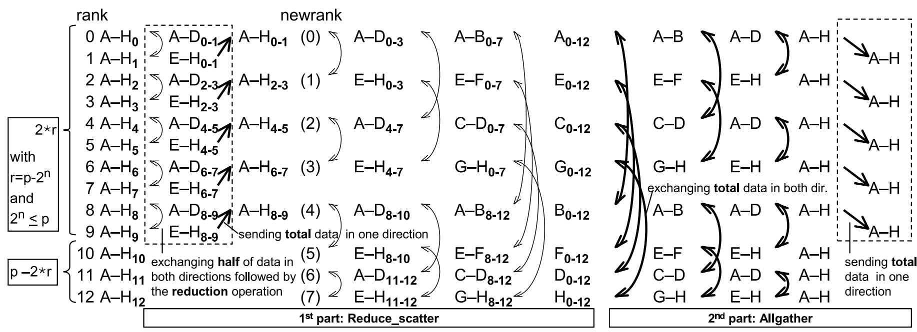

Fig. 2. Recursive Halving and Doubling. The figure shows the intermediate results after each buffer exchange (followed by a reduction operation in the 1st part). The dotted frames show the overhead caused by a non-power-of-two number of processes.

$(rank + 1)$ the second half of the input vector and from each odd $rank$ to the even $(rank - 1)$ the first half of the input vector. All $2r$ processes compute the reduction on their half.

Fig. 2 shows the protocol with an example on 13 processes. The input vectors and all reduction results will be divided into p' parts (A, B,..., H) by this algorithm, and therefore it is denoted with A–H$_{rank}$. After the first reduction, process P0 has computed A–D$_{0-1}$, denoting the reduction result of the first half of the vector (A–D) from the processes 0–1. P1 has computed E–H$_{0-1}$, P2 A–D$_{2-3}$, ... The first step is finished by sending those results from each odd process (1 ... $2r - 1$) to $rank - 1$ into the second part of the buffer.

Now, the first r even processes and the $p - 2r$ last processes are renumbered from 0 to $p' - 1$.

This first step needs $(1 + f_\alpha)\alpha + \frac{1+f_{beta}}{2}n\beta + \frac{1}{2}n\gamma$ and is not necessary, if the number of processes p was already a power-of-two.

Now we start with the first step of recursive vector halving and distance doubling, i.e., the even / odd ranked processes are sending the second / first half of their buffer to $rank' + 1$ / $rank' - 1$. Then the reduction is computed between the local buffer and the received buffer. This step costs $\alpha + \frac{1}{2}(n\beta + n\gamma)$.

In the next $\lg p' - 1$ steps, the buffers are recursively halved and the distance doubled. Now, each of the p' processes has $\frac{1}{p'}$ of the total reduction result vector, i.e., the reduce_scatter has scattered the result vector to the p' processes. All recursive steps cost $\lg p'\alpha + (1 - \frac{1}{p'})(n\beta + n\gamma)$.

The second part implements an allgather or gather to complete the allreduce or reduce operation.

Allreduce: Now, the contrary protocol is needed: Recursive vector doubling and distance halving, i.e., in the first step the process pairs exchange $\frac{1}{p'}$ of the buffer to achieve $\frac{2}{p'}$ of the result vector, and in the next step $\frac{2}{p'}$ is exchanged to get $\frac{4}{p'}$, and so on. A–B, A–D ... in Fig. 2 denote the already stored portion of the result vector. After each communication exchange step, the result buffer is

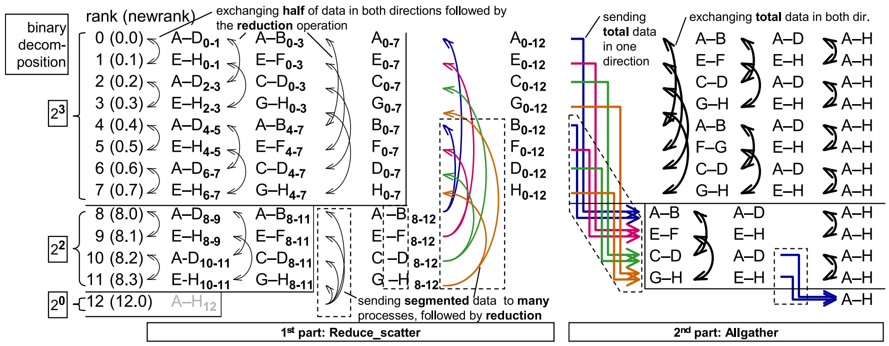

Fig. 3. Binary Blocks.

doubled and after $\lg p'$ steps, the p' processes have received the total reduction result. This allgather part costs $\lg p'\alpha + (1 - \frac{1}{p'})(n\beta)$.

If the number of processes is non-number-of-two, then the total result vector must be sent to the r *removed* processes. This causes the additional overhead $\alpha + n\beta$. The total implementation needs

- $T_{all,h\&d,n=2^{exp}} = 2\lg p\alpha + 2n\beta + n\gamma - \frac{1}{p}(2n\beta + n\gamma)$
$\simeq 2\lg p\alpha + 2n\beta + n\gamma$ if p is power-of-two,
- $T_{all,h\&d,n\neq 2^{exp}} = (2\lg p' + 2 + f_\alpha)\alpha + (3 + \frac{1+f_{beta}}{2})n\beta + \frac{3}{2}n\gamma - \frac{1}{p'}(2n\beta + n\gamma)$
$\simeq (3 + 2\lfloor\lg p\rfloor)\alpha + 4n\beta + \frac{3}{2}n\gamma$ if p is non-power-of-two (with $p' = 2^{\lfloor\lg p\rfloor}$).

This protocol is good for long vectors and power-of-two processes. For non-power-of-two processes, the transfer overhead is doubled and the computation overhead is enlarged by $\frac{3}{2}$. The *binary blocks* protocol (see below) can reduce this overhead in many cases.

Reduce: The same protocol is used, but the pairwise exchange with sendrecv is substituted by single message passing. In the first step, each process with the bit with the value $p'/2$ in its new rank identical to that bit in root rank must receive a result buffer segment and the other processes must send their segment. In the next step only the receiving processes continue and the bit is shifted 1 position right (i.e., $p'/4$). And so on. The time needed for this gather operation is $\lg p'\alpha_{uni} + (1 - \frac{1}{p'})n\beta_{uni}$.

In the case that the original root process is one of the *removed* processes, then the role of this process and its partner in the first step are exchanged after the first reduction in the reduce_scatter protocol. This causes no additional overhead.

The total implementation needs

- $T_{red,h\&d,n=2^{exp}} = \lg p(1 + f_\alpha)\alpha + (1 + f_\beta)n\beta + n\gamma - \frac{1}{p}(n(\beta + \beta_{uni}) + n\gamma)$
$\simeq 2\lg p\alpha + 2n\beta + n\gamma$ if p is power-of-two,
- $T_{red,h\&d,n\neq 2^{exp}} = \lg p'(1 + f_\alpha)\alpha + (1 + f_\alpha)\alpha + (1 + \frac{1+f_{beta}}{2} + f_\beta)n\beta + \frac{3}{2}n\gamma - \frac{1}{p'}((1 + f_\beta)n\beta + n\gamma)$
$\simeq (2 + 2\lfloor\lg p\rfloor)\alpha + 3n\beta + \frac{3}{2}n\gamma$ if p is non-power-of-two (with $p' = 2^{\lfloor\lg p\rfloor}$).

3.6 Binary Blocks

The algorithm starts with a binary block decomposition of all processes in blocks with power-of-two number of processes, see example in Fig. 3. Each block executes its own reduce_scatter with the recursive buffer halving and distance doubling algorithm as described in the previous section. Then, starting with the smallest block, the intermediate result (or the input vector in case of 2^0 process) is split into the segments of the intermediate result in the next higher block, sent to the processes there and the reduction operation is executed there. This causes a load imbalance in computation and communication compared to the execution in the larger blocks. In our example, in the 3rd exchange step in the $\boxed{2^3}$ block, each process sends one segment (e.g., B in P0), receives one segment (A), and computes the reduction of one segment (A). The load imbalance is introduced here by the blocks $\boxed{2^2}$ and $\boxed{2^0}$: In the $\boxed{2^2}$ block, each process receives and reduces 2 segments (e.g. A–B on P8), while in the 2^0 block (here only P12), each process has to send as many messages as the ratio of the two block sizes (here $2^2/2^0$). At the end of the 1st part, the highest block must be recombined with the next smaller block. Again, the ratio of the block sizes determines the overhead.

Therefore, the maximum difference between the ratio of two successive blocks, especially in the low range of exponents, determines the imbalance. On the other hand, this difference may be small, e.g., the most used non-power-of-two numbers of processes on our Cray T3E fall into the categories $\delta_{\mathrm{expo,max}} = 1$ (96 [12 % of system usage with MPI applications], and 60 PEs [procesing elements]), $\delta_{\mathrm{expo,max}} = 2$ (61, 80, 235, 251 PEs), and $\delta_{\mathrm{expo,max}} = 3$ (36, 77, 100 PEs).[2]

Allreduce: The 2nd part is an allgather implemented with buffer doubling and distance halving in each block as in the algorithm in the previous section. The input must be provided in the processes of the smaller blocks always with pairs of messages from processes of the next larger block.

Reduce: If the root is outside of the largest block, then the intermediate result segment of rank 0 is sent to root and root plays the role of rank 0. A binary tree is used to gather the result segments into the root process.

For power-of-two number of processes, the binary block algorithms are identical to the halving and doubling algorithm in the previous section.

3.7 Ring

While the algorithms in the last two sections are optimal for power-of-two process numbers and long vectors, for medium non-power-of-two number of processes and long vectors there exist another good algorithm. It uses the *pairwise exchange* algorithm for reduce_scatter and *ring* algorithm for allgather (for

[2] $\delta_{\mathrm{expo,max}}$ is the maximal difference of two consecutive exponents in the binary representation of the number of processes, e.g., $100 = 2^6 + 2^5 + 2^2$, $\delta_{\mathrm{expo,max}} = \max(6-5, 5-2) = 3$.

allreduce), as described in [15], and for reduce, all processes send their result segment directly to root. Both algorithms are good in bandwidth usage for non-power-of-two number of processes, but the latency scales with the number of processes. Therefore this algorithm can be used only for a small number of processes. Independent of whether p is power-of-two or not, the total implementation needs $T_{all,ring} = 2(p-1)\alpha + 2n\beta + n\gamma - \frac{1}{p}(2n\beta + n\gamma)$ for allreduce, and $T_{red,ring} = (p-1)(\alpha + \alpha_{uni}) + n(\beta + \beta_{uni}) + n\gamma - \frac{1}{p}(n(\beta + \beta_{uni}) + n\gamma)$ for reduce.

4 Choosing the Fastest Algorithm

Based on the number of processes and the vector (input buffer) length, the reduction routine must decide which algorithm should be used. Fig. 4 shows the fastest protocol on a Cray T3E 900 with 540 PEs. For buffer sizes less than or equal to 32 byte, *recursive doubling* is the best, for buffer sizes less than or equal to 1 KB, mainly *vendor's algorithm* (for power-of-two) and *binary tree* (for non-power-of-two) are the best but there is not a big difference to *recursive doubling*. For longer buffer sizes, the *ring* is good for some buffer sizes and some #processes less than 32 PEs. A detailed decision is done for each #processes value, e.g., for 15 processes, *ring* is used if length $\geq$ 64 KB. In general, on a Cray T3E 900, the binary block algorithm is faster if $\delta_{\mathrm{expo,max}} < \lg(\frac{vector\ size}{1 Byte})/2.0 - 2.5$ and

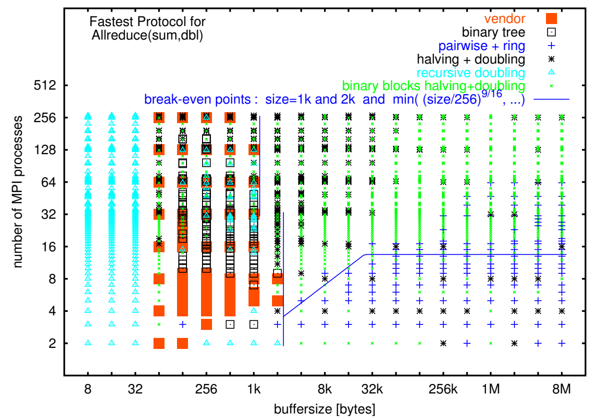

Fig. 4. The fastest protocol for Allreduce(double, sum) on a Cray T3E 900.

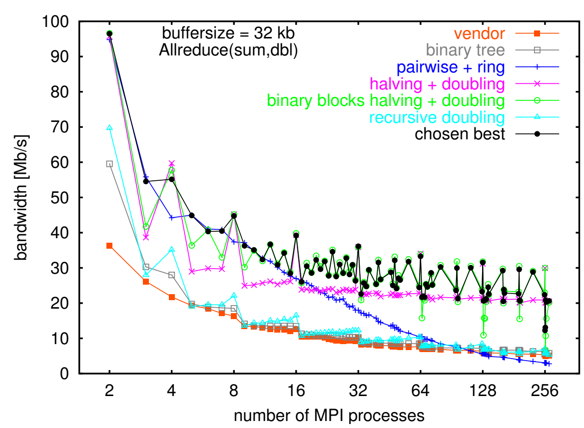

Fig. 5. Bandwidth comparison for Allreduce(double, sum) with 32 KB vectors on a Cray T3E 900.

vector size $\geq$ 16 KB and more than 32 processes are used. In a few cases, e.g., 33 PEs and less then 32 KB, *halving&doubling* is the fastest algorithm.

Fig. 5 shows for 32 KB buffer size that the new protocols are clearly better than the vendor's protocol (MPT.1.4.0.4) and the binary tree for all numbers of processes. Up to 32 PEs, all numbers of processes are measured. For more than 32 PEs, only selected values with small and large $\delta_{\mathrm{expo,max}}$ are measured. One can verify, that *binary blocks*' bandwidth depends strongly on $\delta_{\mathrm{expo,max}}$ and that *halving&doubling* is faster on 33, 65, 66, 97, 128-131, ... PEs. The *ring* is faster on 3, 5, 7, 9-11, and 17 PEs.

5 Comparison

Fig. 6 shows that with the pure MPI programming model (i.e., 1 MPI process per CPU) on the IBM SP, the benefit is about 1.5x for buffer sizes 8–64 KB, and 2x – 5x for larger buffers. With the hybrid programming model (1 MPI process per SMP node), only for buffer sizes 4–128 KB and more than 4 nodes, the benefit is about 1.5x – 3x.

Fig. 7 compares the new algorithm with the old MPICH-1' algorithm (without the halving&doubling). The new algorithms show a performance benefit of 3x – 7x with pure MPI and 2x – 5x with the hybrid model.

Fig. 8 shows, that in many cases the new algorithms are 3x – 5x faster than the vendors algorithm with operation MPI_SUM and due to the very slow implementation of structured derived datatypes, a factor up to 100x with MPI_MAXLOC.

On Cray X1, we compare the new algorithms with the current development version of Cray's MPI library (mpt.2.4.0.0.13). Our measurements have shown, that the shared memory based implementation of MPI_Allreduce and MPI_Reduce [10] has an up to 14 times shorter latency (6–14 μs) as the protocols based on point-to-point message passing and presented in this paper (39–137 μs) at MPI_Allreduce computing the sum of vectors, each with 1 double element. On the other hand, Fig. 9 shows that the new MPI_Allreduce protocols are significantly faster for longer vectors. Looking at 96 and more MSPs (Multi Streaming Processors, consisting internally of 4 CPUs) and 32 kB (= 4k doubles) and more vector size, we can see that the new presented protocols are more than 35 % faster than Cray's mpt. For 96 and more MPSs and vector sizes with 256 kB (=32k doubles) and more, the new protocols are 4 to 10 times faster than Cray's mpt, although [10] states that this mpt uses already butterfly protocols for longer buffers. The lower diagram indicates, which protocol has achieved the best bandwidth.

Fig. 10 shows, that for MPI_Reduce, the differences are significantly smaller: With more than 8 MSPs, and at least 2 MB buffer size, one can see that the new protocols are faster than Cray's mpt, but only with a ratio between 1.14 and 2.01.

The reduction operation loop is compiled with the pragma function _Pragma("_CRI concurrent"). The new algorithms vectorize and multi-stream on the MSPs, including the minloc and maxloc operation, on all available datatypes,

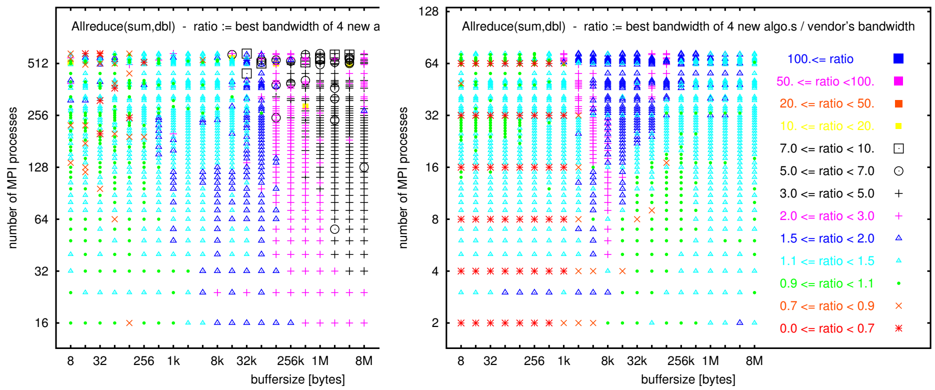

Fig. 6. Ratio of bandwidth of the fastest protocol (without *recursive doubling*) on a IBM SP at SDSC and 1 MPI process per CPU (left) and per SMP node (right)

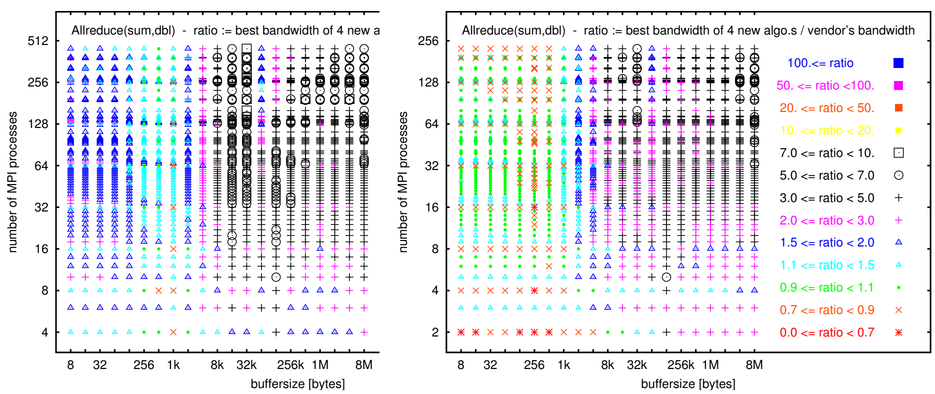

Fig. 7. Ratio of bandwidth of the fastest protocol (without *recursive doubling*) on a Myrinet cluster with dual-CPU PCs (HELICS cluster, University of Heidelberg) and 1 MPI process per CPU (left) and per SMP node (right)

except on short and byte datatypes. Internally, all datatypes are mapped to the appropriate number of MPI_BYTE elements, before MPI point-to-point message passing routines are called. E.g., with 116 MSPs and 8 MB vector size, the minimal execution time is 6.84 ms and 11.67 ms (allreduce with sum and maxloc), and 5.04 ms and 10.94 ms (reduce with sum and maxloc), which implies following bandwidth values (based on the 8 MB) per process: 1227 MB/s and 719 MB/s (allreduce) and 1664 MB/s and 767 MB/s (reduce). This speed is achieved with the binary block protocol. On 64 MSPs and with recursive halving and doubling, one can achieve 1362 MB/s and 909 MB/s (allreduce) and 1792 MB/s and 1048 MB/s (reduce).

The used mpt.2.4.0.0.13 is an intermediate development version from Cray. The MPI_MAXLOC and MPI_MINLOC operations are not yet optimized. There-

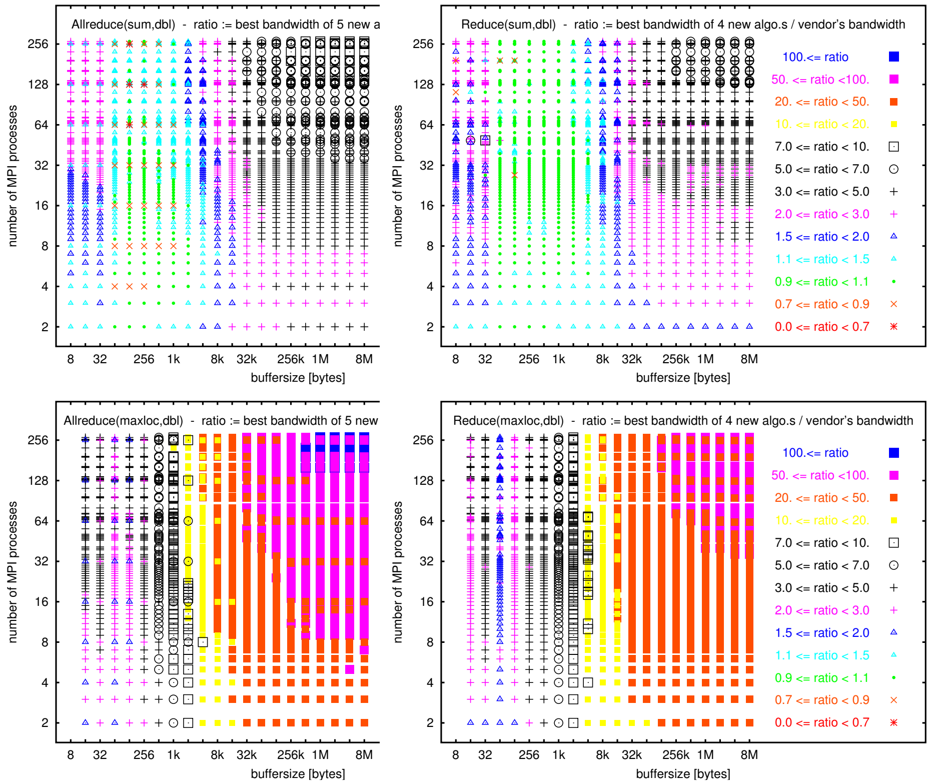

Fig. 8. Ratio of bandwidth of the fastest protocol (*recursive doubling* [allreduce only], *binary tree*, *ring*, *halving&doubling*, and *binary blocks*) compared to the vendors algorithm for Allreduce (left) / Reduce (right) and operation MPI_SUM (1st row) / MPI_MAXLOC (2nd row) on a Cray T3E 900.

fore the comparison of the new protocols with Cray's mpt shows still a ratio up to 1800 with allreduce and up to 20 with reduce. The extreme performance bug of allreduce may be based on performance problems with an internally used bcast on derived datatypes. These problems should be solved before this mpt.2.4 is delivered as product.

6 Conclusions and Future Work

Although principal work on optimizing collective routines is quite old [2], there is a lack of fast implementations for *allreduce* and *reduce* in MPI libraries for a wide range of number of processes and buffer sizes. Based on the author's algorithm from 1997 [11], an efficient algorithm for power-of-two and non-power-of-two number of processes is presented in this paper. Medium non-power-of-two number of processes could be additionally optimized with a special ring

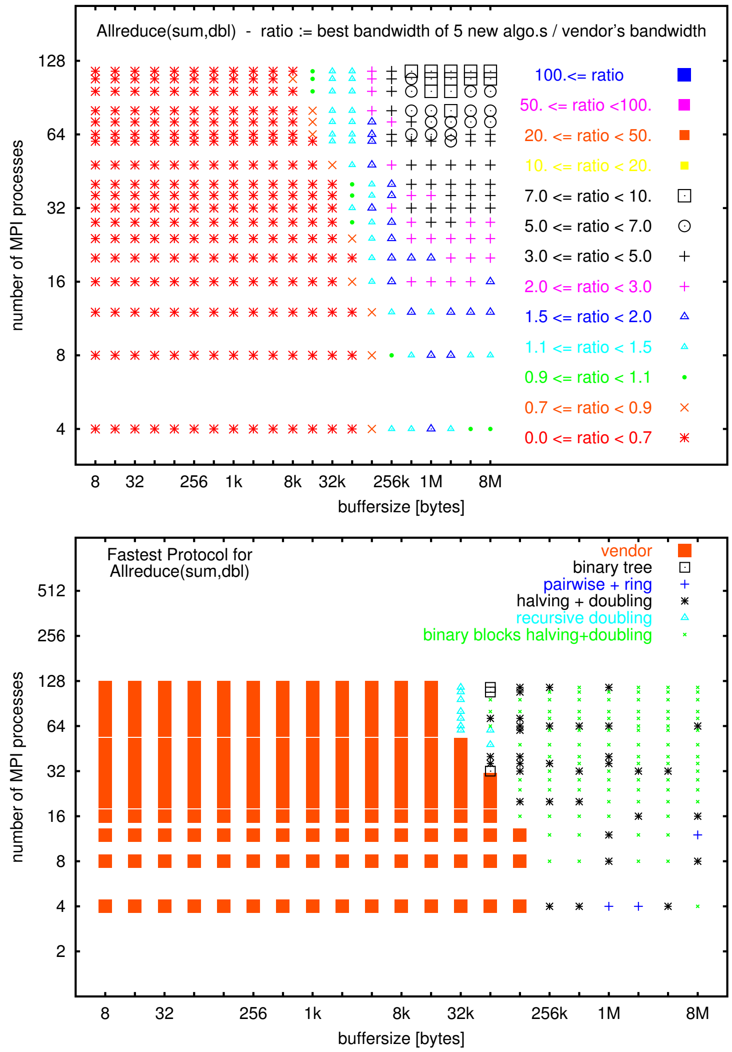

Fig. 9. Ratio of bandwidth of the fastest protocol (*recursive doubling*, *binary tree*, *ring*, *halving&doubling*, and *binary blocks*) compared to Cray mpt.2.4.0.0.13 algorithm for MPI_Allreduce and operation MPI_SUM on a Cray X1 in MSP mode (upper diagram) and the fastest protocol (lower diagram).

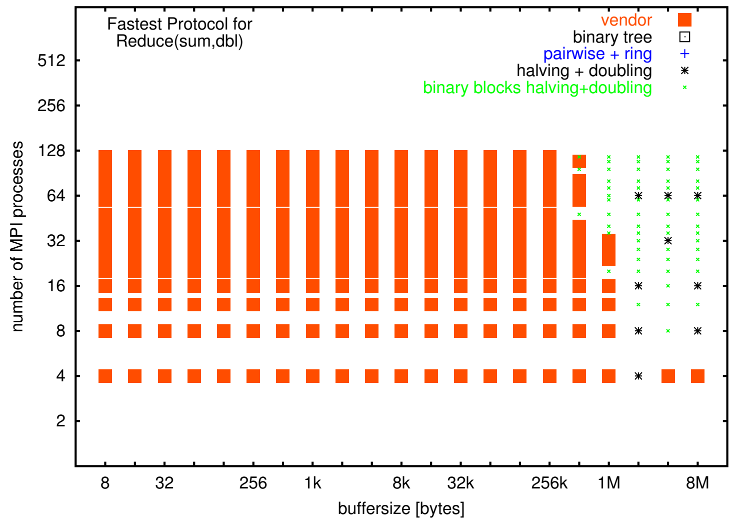

Fig. 10. Ratio of bandwidth of the fastest protocol (*binary tree*, *ring*, *halving&doubling*, and *binary blocks*) compared to Cray mpt.2.4.0.0.13 algorithm for MPI_Reduce and operation MPI_SUM on a Cray X1 in MSP mode.

algorithm. The *halving&doubling* is already included into MPICH-2 and it is planned to include the other bandwidth-optimized algorithms [11, 15]. Future work will further optimize latency and bandwidth for any number of processes by combining the principles used in Sect. 3.3–3.7 into one algorithm and selecting on each recursion level instead of selecting one of those algorithms for all levels [14].

Cray's mpt.2.4.0.0.13 already shows excellent latency for smallest vectors. For long vectors, there is still a big gap between the speed that can be reached and the speed implemented by Cray's mpt intermediate development version with ratios up to 2 for reduce(sum), 10 for allreduce(sum), 20 for reduce(maxloc), and 1800 for allreduce(maxloc). This gap may be or should be closed in the mpt.2.4 product version.

Acknowledgments

The authors would like to acknowledge their colleagues and all the people that supported this project with suggestions and helpful discussions. They would especially like to thank Rajeev Thakur and Jesper Larsson Träff for the helpful discussion on optimized reduction algorithm and Gerhard Wellein, Thomas Ludwig, Ana Kovatcheva, Rajeev Thakur, Monika Wierse, Howard Pritchard, Patrick H. Worley, Terry Hewitt, Mike Pettipher, Adrian Tate for their benchmarking support.

References

1. V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis and M. Snir, *CCL: A portable and tunable collective communication library for scalable*

parallel computers, in IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 2, Feb. 1995, pp 154–164.
2. M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Gejin, and J. Watts, *Interprocessor collective communication library (InterCom)*, in Proceedings of Supercomputing '94, Nov. 1994.
3. Edward K. Blum, Xin Wang, and Patrick Leung, *Architectures and message-passing algorithms for cluster computing: Design and performance*, in Parallel Computing 26 (2000) 313–332.
4. J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, *Efficient algorithms for all-to-all communications in multiport message-passing systems*, in IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 11, Nov. 1997, pp 1143–1156.
5. E. Gabriel, M. Resch, and R. Rühle, *Implementing MPI with optimized algorithms for metacomputing*, in Proceedings of the MPIDC'99, Atlanta, USA, March 1999, pp 31–41.
6. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Rel. 1.1, June 1995, `www.mpi-forum.org`.
7. N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, *Exploiting hierarchy in parallel computer networks to optimize collective operation performance*, in Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS '00), 2000, pp 377–384.
8. Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, Raoul A. F. Bhoedjang, *MPI's reduction operations in clustered wide area systems*, in Proceedings of the Message Passing Interface Developer's and User's Conference 1999 (MPIDC'99), Atlanta, USA, March 1999, pp 43–52.
9. Man D. Knies, F. Ray Barriuso, William J. Harrod, George B. Adams III, *SLICC: A low latency interface for collective communications*, in Proceedings of the 1994 conference on Supercomputing, Washington, D.C., Nov. 14–18, 1994, pp 89–96.
10. Howard Pritchard, Jeff Nicholson, and Jim Schwarzmeier, *Optimizing MPI Collectives for the Cray X1*, in Proceeding of the CUG 2004 conference, Knoxville, Tennessee, USA, May, 17-21, 2004.
11. Rolf Rabenseifner, *A new optimized MPI reduce and allreduce algorithm*, Nov. 1997. `http://www.hlrs.de/mpi/myreduce.html`
12. Rolf Rabenseifner, *Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512*, Proceedings of the Message Passing Interface Developer's and User's Conference 1999 (MPIDC'99), Atlanta, USA, March 1999, pp 77–85. `http://www.hlrs.de/people/rabenseifner/publ/publications.html`
13. R. Rabenseifner, *Optimization of collective reduction operations*, in M. Bubak et al. (Eds.): International Conference on Computational Science (ICCS 2004), June 6-9, Krakow, Poland, LNCS 3036, pp 1–9, 2004.
14. Rolf Rabenseifner and Jesper L. Träff, *More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems*, to be published in Recent Advances in Parallel Virtual Machine and Message Passing Interface, proceedings of the 11th European PVM/MPI Users' Group Meeting, LNCS, J. Dongarra, et al. (Eds.), Springer, 2004.
15. Rajeev Thakur and William D. Gropp, *Improving the performance of collective operations in MPICH*, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, proceedings of the 10th European PVM/MPI Users' Group Meeting, LNCS 2840, J. Dongarra, D. Laforenza, S. Orlando (Eds.), 2003, 257–267.
16. Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra, *Automatically tuned collective communications*, in Proceedings of SC2000, Nov. 2000.