MPI Reduction Operations for Sparse Floating-Point Data

Michael Hofmann[⋆] and Gudula Rünger

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

Abstract This paper presents a pipeline algorithm for `MPI_Reduce` that uses a *Run Length Encoding* (RLE) scheme to improve the global reduction of sparse floating-point data. The RLE scheme is directly incorporated into the reduction process and causes only low overheads in the worst case. The high throughput of the RLE scheme allows performance improvements when using high performance interconnects, too. Random sample data and sparse vector data from a parallel FEM application is used to demonstrate the performance of the new reduction algorithm for an HPC Cluster with InfiniBand interconnects.

Keywords: MPI, performance optimization, pipelining, reduction operation, run length encoding

1   Introduction

The *Message Passing Interface* (MPI) is the de facto standard for distributed memory parallel programming in the area of scientific high performance computing and the optimization of MPI libraries and communication operations is still an active field of research. Emerging high performance interconnects such as Quadrics, Myrinet, SCI, or InfiniBand have led to continuing efforts for improving the performance of MPI implementations, too. Especially for collective MPI operations, there exists a variety of different algorithms. Automatic tuning as well as static and dynamic optimizations are used to adapt to specific system architectures and applications by selecting appropriate algorithms or algorithmic parameters [1,2]. Good overall performance of communication operations requires a transition from latency-optimal algorithms for small messages to bandwidth-optimal algorithms for large messages. Pipelining techniques are used for achieving high bandwidth, especially with high performance interconnects [3,4]. Improved algorithms for global reduction operations (e.g., `MPI_Allreduce`) are presented in [5].

    The contribution of this paper is to apply the well known *Run Length Encoding* (RLE) to floating-point data and to incorporate it into a pipeline algorithm

---

for `MPI_Reduce`. The RLE scheme reduces the amount of communication and increases the performance, especially for sparse vector data. General purpose compression algorithms and specific algorithms for floating-point data are used to improve the performance of MPI communication operations on clusters with Fast Ethernet interconnects [6,7]. However, the throughputs of these algorithms are unstable and too small for high performance interconnects. Our target platform is the HPC Cluster CHiC [8] consisting of 530 compute nodes with InfiniBand interconnects. MPI reduction operations are used, for instance in parallel numerical methods for implementing global error control. We use random sample data as well as sparse vector data from a parallel FEM application to investigate the performance of the new reduction algorithm with RLE.

The rest of this paper is organized as follows. Section 2 investigates the feasibility of applying data compression for optimizing communication operations and introduces the RLE scheme for floating-point data. Section 3 describes a pipeline algorithm for `MPI_Reduce` and the application of the RLE scheme. Section 4 presents performance results and Section 5 concludes the paper.

2 Compression of Floating-Point Data

The communication time for large messages is mainly determined by the bandwidth of the communication network. To achieve a benefit from transferring compressed data instead of uncompressed data, the additional computational time of the compression algorithm has to be lower than the time saved during the communication. Under optimal conditions, the compression and decompression operations perfectly overlap and the message size is reduced so that the communication time can be neglected. To benefit from the compression in that case, the throughput of the compression/decompression operation has to be at least as high as the bandwidth of the communication network.

A general purpose data compression library like zlib [9] achieves throughputs of 0.5-22 MB/s (depending on the specified compression level) using a 2.6 GHz AMD Opteron processor. The algorithm of Ratanaworabhan et al. for compressing scientific floating-point data achieves throughputs of about 22-47 MB/s using a 3.0 GHz Pentium 4 processor [10]. When using high performance interconnects such as InfiniBand, the performance of these algorithms is insufficient. The HPC cluster CHiC reaches bandwidths of about 970 MB/s for unidirectional point-to-point communication with `MPI_Send`/`MPI_Recv`.

This estimation about the required throughput assumes that compression and decompression occur as additional tasks before and after the data transmission. Nevertheless, it is also possible to incorporate the compression algorithm into operations that are already existing. For example, the compression can be done when the message is copied to communication buffers or when the reduction operation of `MPI_Reduce` is applied. In that case, a compression/decompression throughput equal to the bandwidth of the communication network helps to prevent a loss of performance even if the size of the message can not be reduced.

2.1 Run Length Encoding for Floating-Point Data

Run Length Encoding is a well known compression scheme that works by replacing repetitions of equal values with the information about the number of repetitions. Non-repeating values remain unchanged. The RLE scheme is useful for data that contains long sequences of equal values, e.g. sparse vector data with many zero values. The encoded repetition of a value requires a marker that is distinguishable from the not-encoded values. We use *Not a Number* (NaN) values in the IEEE 754 representation of floating point numbers as markers. Considering 64-Bit floating-point numbers, NaN values have an arbitrary sign bit, all 11 bits of the exponent set to one and a non-zero mantissa (52 bits). We use the non-zero mantissa to save the number of repetitions as a 52-Bit integer.

This RLE scheme for floating-point data can be adapted to different use cases. If only repetitions of one fixed value (e.g., zero) are considered, then every NaN in the encoded data represents a sequence of at least two of these values. If repetitions of arbitrary values are considered, then the specific value that is repeated has to be saved together with the NaN. In this case, it is appropriate to skip the encoding of sequences of size two, since their encoded size is the same as their original size. This RLE scheme does not increase the size of the encoded data. If NaN values are included in the original data, a distinction between original and encoded NaNs is required. To preserves most of the information of the original NaN, the arbitrary sign bit can be used for this distinction.

2.2 Throughputs of RLE for Sparse Floating-Point Data

The throughput of the RLE scheme is evaluated using an implementation for repetitions of zeros. Compressible random sample data is used that consists of floating-point vectors with randomly placed non-zero values. All operations are written in C and compiled with the PathScale 3.1 compiler (optimization `-O3`). The throughput values are calculated according to the size of *one* vector. Throughputs with respect to the total amount of data read and written by the operations (without RLE) can be obtained, by applying a factor of two for the copy operation and a factor of three for the vector addition operations.

Figure 1 (left) shows throughputs of the compression and decompression operation depending on the amount of non-zero values in the input data. The throughputs of both operations strongly depend on the amount of non-zero values. The compression operation shows a significant loss of performance when having more than 20 % non-zero values. With 100 % non-zero values the performance of both operations is comparable to the copy operation.

Figure 1 (right) shows throughputs of vector addition operations with and without RLE depending on the amount of non-zero values in the input data. Operations $a_C + b \rightarrow a_C$ and $a_C + b \rightarrow c_C$ represent the addition of a compressed vector a_C and an uncompressed vector b where the result (in a_C or c_C) is compressed, too. The compressed version using only two vector arrays (a and b) reaches about 70-95 % of the performance of the uncompressed version. Similar to
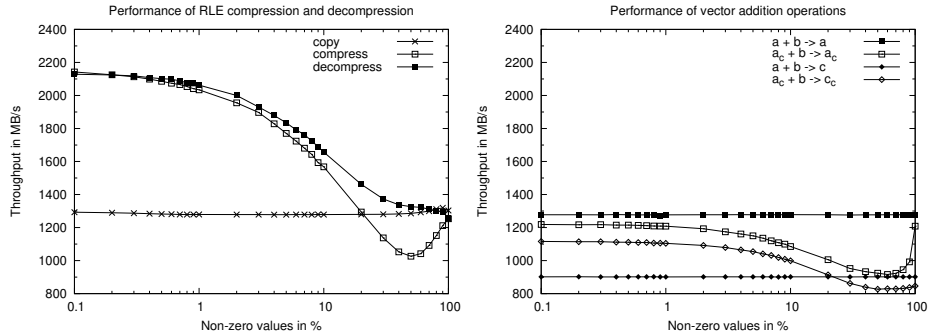
Figure 1. Throughputs of RLE compression and decompression operations (left) and vector addition operations with and without RLE (right).

the compression operation, the results show a loss of performance if the amount of non-zero values increases, but still good results with 100 % non-zero values.

In comparison to these results, the `memcpy` operation of the PathScale compiler achieved throughputs of about about 3 GB/s and the vector addition operation (`daxpy`) of the AMD Core Math Library achieved throughputs of about 2 GB/s. The performance of these highly optimized operations shows, that there is still room for improving the compiler optimized implementations. The addition operation in conjunction with repetitions of zeros provides several characteristics that ease the implementation of the corresponding vector operations. Nevertheless, the RLE scheme is not limited to repetitions of zeros (see Section 2.1) and applicable to other operations, too. A general vector operation with RLE can be achieved, by incorporating the corresponding compression operation into the process of writing the resulting vector.

3 `MPI_Reduce` with Run Length Encoding

As shown in Section 2, the throughput of data compression algorithms is insufficient in comparison to the bandwidth of high speed interconnects like InfiniBand. Therefore, we incorporate the RLE scheme into the already existing process of applying the reduction operation of `MPI_Reduce`. We start with a pipeline algorithm for `MPI_Reduce` that is appropriate to achieve high bandwidths for large messages. Each process P_i sends data only to process P_{i+1}. The last process P_p is the root process of `MPI_Reduce`. Process P_i performs operation $\boldsymbol{a} \otimes \boldsymbol{b_i} \rightarrow \boldsymbol{a}$ to apply the reduction operation $\otimes$ to the incoming data $\boldsymbol{a}$ and its local data $\boldsymbol{b_i}$. The result is placed in $\boldsymbol{a}$ and send to process P_{i+1}. The data is divided into equal blocks and the sending and receiving of blocks is overlapped using `MPI_Sendrecv`.

We incorporate the compression and the decompression with the RLE scheme into the process of performing the reduction operation for floating-point data.

The regular operation $\boldsymbol{a} \otimes \boldsymbol{b_i} \to \boldsymbol{a}$ is replaced by the compressed version $\boldsymbol{a_C} \otimes \boldsymbol{b_i} \to \boldsymbol{a_C}$. The reduction operation $\otimes$ is applied to the compressed incoming data $\boldsymbol{a_C}$ and the uncompressed data $\boldsymbol{b_i}$ of process P_i. The compressed result is placed in $\boldsymbol{a_C}$ and is sent to process P_{i+1}. The first process P_1 has no incoming data and therefore no reduction operation to perform. We avoid the overhead of an additional compression operation by sending uncompressed data from P_1 to P_2. The compression is initiated by process P_2 using operation $\boldsymbol{a} \otimes \boldsymbol{b_i} \to \boldsymbol{a_C}$. The root process P_p uses the operation $\boldsymbol{a_C} \otimes \boldsymbol{b_p} \to \boldsymbol{a}$ to obtain the uncompressed final result. The RLE scheme can be used together with predefined and user-defined reduction operations. The entire compression and decompression process is hidden in the `MPI_Reduce` operation and requires no changes to the application.

4   Performance Results

Performance results are obtained using the HPC Cluster CHiC consisting of 530 compute nodes each with two 2.6 GHz Dual-Core AMD Opteron processors, 4 GiB main memory and InfiniBand interconnect. One process is used per node.
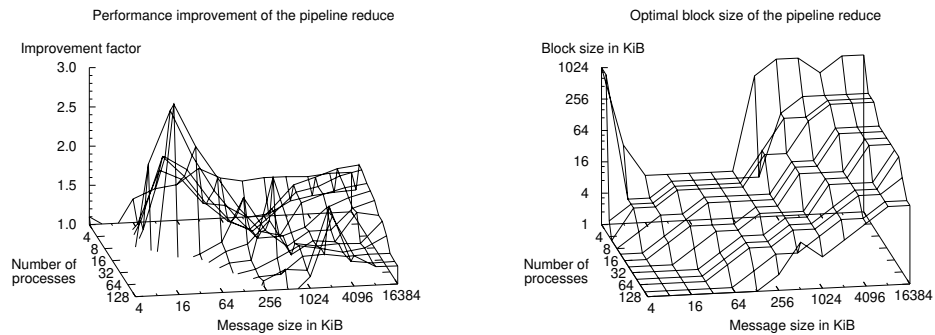

Figure 2. Performance improvement of the pipeline reduce algorithm in comparison to the native `MPI_Reduce` (left) and optimal block size (right).

The impact of the pipeline reduce algorithm is demonstrated first. Figure 2 (left) shows the relative improvement of the pipeline reduce algorithm in comparison to the native `MPI_Reduce` of OpenMPI (version 1.2.4) depending on the number of processes and the message size. The results show a decrease in performance for message sizes up to 8 KiB in general and up to 256 KiB for large numbers of processes. For large message sizes, the pipeline reduce algorithm achieves improvements up to a factor of about 1.8 (single peaks show improvements up to about 2.9). Figure 2 (right) shows the corresponding block sizes of the pipeline algorithm that achieve the best performance. The results show that the block size increases with increasing message sizes and decreasing numbers of processes.
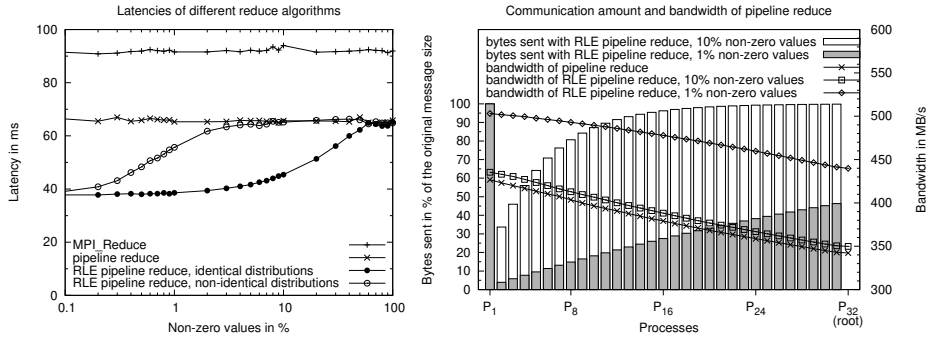
Figure 3. Latency of the native `MPI_Reduce` and the pipeline reduce with and without RLE using different distributions of non-zero values (left). Communication amount and bandwidth of the different processes participating in the pipeline reduce (right).

Figure 3 (left) shows latency results for `MPI_Reduce` and the pipeline reduce with and without RLE depending on the amount of non-zero values in the input data with 16 MiB messages, and 128 processes. The RLE scheme is incorporated into the vector addition operation to compress repetitions of zero values. Results are shown for identical and non-identical random distributions of non-zero values in the different messages of the processes. In contrast to the constant performance of `MPI_Reduce` and the pipeline reduce, RLE pipeline reduce shows a dependence on the amount of non-zero values. With 0.1 % non-zero values the latency of RLE pipeline reduce falls below 40 ms. With 100 % non-zero values, using the RLE scheme causes no overhead and is as fast as pipeline reduce without RLE (about 65 ms). The distribution of the non-zero values has a significant influence on the performance of RLE pipeline reduce, too. With identical distributions on all processes, improvements are achieved with up to 50 % non-zero values while with non-identical distributions the improvements vanish when using more than 10 % non-zero values.

Figure 3 (right) shows the amount of data sent by the individual processes using 1 % and 10 % non-identically distributed non-zero values, 16 MiB messages, and 32 processes. Bandwidths calculated from the latencies of the individual processes are also shown. Since process P_1 sends uncompressed data to P_2, a decreased amount of transferred data is first observed for P_2. Because of the non-identical distributions of the non-zero values, the number of non-zero values increases when the data approaches the root process. This reduces the efficiency of the RLE scheme and increases the amount of data transferred by the latter processes in the pipeline. With 10 % non-zero values, starting at process P_{11} over 90 % of the original data is transferred. With 1 % non-zero values, the amount of transferred data increases more slowly resulting in a higher improvement.

Next, we use data from a parallel adaptive FEM application. The elements of the floating-point vectors correspond to the nodes of the mesh used by the FEM application. Adaptive mesh refinement increases the number of mesh nodes

and therefore the size of the vectors, too. According to the distribution of the mesh nodes to the different processes, each process contributes only to a subset of elements of the solution vector. The complete solution vector is obtained by a summation of all local contributions using `MPI_Reduce`. Figure 4 (left) shows an example for the sparse structure of the vectors supplied to `MPI_Reduce`. The vectors consist of 564 elements with about 6-11 % non-identically distributed non-zero values.
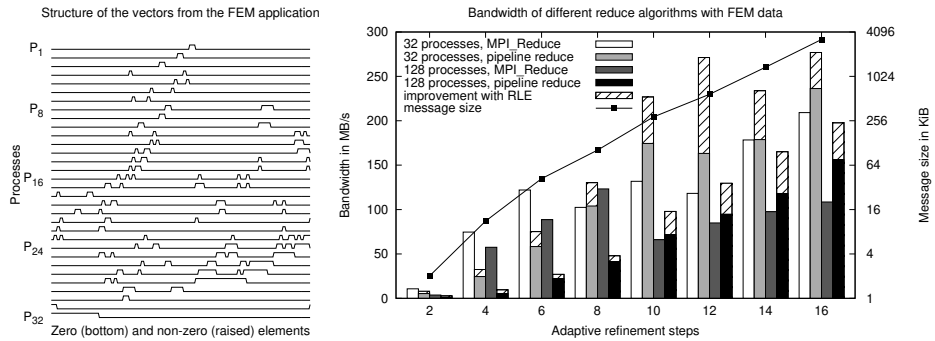

Figure 4. Zero and non-zero elements of vectors from the FEM application (left). Bandwidth of the native `MPI_Reduce` and the pipeline reduce with and without RLE using data of the FEM application (right).

Figure 4 (right) shows bandwidths of the root process for `MPI_Reduce` and the pipeline reduce with and without RLE using the data of the FEM application after different adaptive refinement steps. As previously seen in Figure 2, pipeline reduce achieves performance improvements only for large messages while the native `MPI_Reduce` is better for small messages. With 32 processes, improvements are achieved after more than six refinement steps ($\approx 42\,\mathrm{KiB}$ messages) and with 128 processes after more than eight refinement steps ($\approx 72\,\mathrm{KiB}$ messages). Pipeline reduce with RLE has always a higher performance than without RLE. However, the improvements are most significant for large messages. In comparison to the native `MPI_Reduce`, the bandwidth of RLE pipeline reduce increases up to 228 % with 32 processes and up to 182 % with 128 processes.

Instead of integrating this kind of optimization into the MPI operations, it is also possible to utilize an appropriate sparse vector format inside the application. However, these unconventional formats prevent the usage of operations like `MPI_Reduce` and require that optimized communication algorithms are implemented on the application level, too. The RLE scheme is rather simple and the high throughputs of the RLE operations prevent a loss of performance when using incompressible input data. Integrated into an MPI library, the RLE compression scheme could be enabled by default or optionally used with a new special MPI datatype.

5 Conclusion

In this paper, we have shown that a fast RLE scheme can be used to improve the performance of `MPI_Reduce` even with high performance interconnects such as InfiniBand. We have introduced an RLE scheme for floating-point data and incorporated the compression and decompression process into the reduction operation of `MPI_Reduce`. Performance results show that the pipeline reduce algorithm and the RLE scheme lead to significant performance improvements for large messages. The improvements due to the RLE scheme strongly depend on the input data. However, the marginal overhead of the RLE scheme prevents a decrease in performance when using incompressible input data. Results with sparse floating-point data from a parallel FEM application show improvements in bandwidth up to a factor of two.

Acknowledgment

References

1. Faraj, A., Yuan, X., Lowenthal, D.: STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations. In: ICS '06: Proc. of the 20th annual international conference on Supercomputing, ACM (2006) 199–208
2. Pješivac-Grbović, J., Bosilca, G., Fagg, G.E., Angskun, T., Dongarra, J.J.: MPI collective algorithm selection and quadtree encoding. Parallel Computing **33**(9) (2007) 613–623
3. Worringen, J.: Pipelining and Overlapping for MPI Collective Operations. In: LCN '03: Proc. of the 28th Annual IEEE International Conference on Local Computer Networks, IEEE CS (2003) 548–557
4. Almási, G., et al.: Optimization of MPI Collective Communication on BlueGene/L Systems. In: ICS '05: Proc. of the 19th annual international conference on Supercomputing. (2005) 253–262
5. Rabenseifner, R., Träff, J.L.: More Efficient Reduction Algorithms for Non-Power-of-Two Number of Processors in Message-Passing Parallel Systems. In: Proc. of the 11th EuroPVM/MPI. Volume 3241 of LNCS., Springer (2004) 36–46
6. Calderón, A., García, F., Carretero, J., Fernández, J., Pérez, O.: New Techniques for Collective Communications in Clusters: A Case Study with MPI. In: ICPP '01: Proc. of the Int. Conf. on Parallel Processing, IEEE CS (2001) 185–194
7. Ke, J., Burtscher, M., Speight, E.: Runtime Compression of MPI Messages to Improve the Performance and Scalability of Parallel Applications. In: SC '04: Proc. of the ACM/IEEE Conf. on Supercomputing, IEEE CS (2004) 59
8. http://www.tu-chemnitz.de/chic/
9. http://www.zlib.net/
10. Ratanaworabhan, P., Ke, J., Burtscher, M.: Fast Lossless Compression of Scientific Floating-Point Data. In: DCC '06: Proceedings of the Data Compression Conference, IEEE CS (2006) 133–142