Versatile Communication Algorithms for Data Analysis

Tom Peterka[⋆] and Robert Ross

Argonne National Laboratory

Abstract. Large-scale parallel data analysis, where global information from a variety of problem domains is resolved in a distributed memory space, relies on communication. Three communication algorithms motivated by data analysis workloads—merge based reduction, swap based reduction, and neighborhood exchange—are presented, and their performance is benchmarked. These algorithms communicate custom data types among blocks assigned to processes in flexible ways, and their performance is optimized by tunable parameters. Performance is compared with an MPI implementation and with previous communication algorithms on an IBM Blue Gene/P supercomputer at a variety of message sizes and process counts.

Keywords: communication for large-scale parallel data analysis

1 Introduction

Large-scale parallel data analysis and visualization often involve intense communication of information in a distributed-memory HPC architecture, for example, when data are analyzed in situ during a computational simulation. Thus, efficient and usable communication algorithms are fundamental to scalable data analysis. While MPI's collectives suffice for some of these tasks, MPI alone does not provide custom domain decompositions, partial reductions, or neighborhood exchanges. Even when a comparable MPI function does exist, configurable algorithms that allow tuning for a target architecture and data movement pattern may outperform MPI implementations for the same task. Our solution is to write such algorithms in a library built on top of MPI.

This paper examines three communication algorithms implemented in such a library. We describe how these algorithms offer capabilities beyond MPI's stock functions. These capabilities include the ability to communicate among blocks instead of processes, so that blocks can be mapped to processes in flexible ways. For example, multiple blocks can be mapped to one MPI process. Reductions are based on configurable radices and rounds and can be either partial or complete depending on these parameters. Neighborhood communication is also included.

Although these communication algorithms have been successfully applied in our prior work to a variety of data analysis tasks, the contribution of this paper is a thorough benchmarking of their performance. We compare with a popular MPI implementation for test configurations where a comparable MPI function can be used. We also compare performance with previous visualization algorithms, in particular, with a highly tuned image compositing algorithm and with our previous implementation of neighborhood exchange in parallel particle tracing.

[⋆] e-mail: tpeterka@mcs.anl.gov

Table 1. Examples of Communication Patterns in Data Analysis

Analysis Kernel	Communication Pattern
Particle tracing [5]	Neighborhood exchange
Information entropy [6]	Merge based reduction
Morse-Smale complex [7]	Merge based reduction
Computational geometry [8]	Neighborhood exchange
Region growing [9]	Neighborhood exchange
Sort-last rendering [3]	Swap based reduction

2 Background and Related Work

Many algorithms for collectives have been published in the message-passing literature, including [1, 2]. The visualization community has developed similar communication algorithms for image compositing [3].

Parallel scientific data analysis and visualization algorithms share a common set of communication patterns. Table 1 shows a representative sample of data analysis kernels and the communication pattern used in each. Some analyses also generate multiple combinations and iterations of these same core patterns. The right-hand column of the table reveals three common communication kernels: merge based reduction, swap based reduction, and neighborhood exchange. These patterns are described further in Section 3.

Algorithms for these three patterns are implemented in a prototype library called DIY (Do-It-Yourself analysis) [4] that the user calls in conjunction with custom local analysis operations. DIY is lightweight, consisting of approximately 15 K lines of code and 800 KB as a statically linked library. DIY's communication algorithms have hooks for custom reduction operators that act on user-defined data types, as in MPI. Additionally, DIY allows communication among arbitrary subsets of the domain, which are called *blocks*, without the user having to worry about which process actually owns a given block. Blocks are assigned to processes during the initialization of DIY, and a process may own more than one block. In the remainder of this paper, we will follow DIY's terminology and say that blocks communicate with each other rather than processes.

Deciding which communication pattern to select for a particular task depends on several factors. If the operation is not associative and the order of information flow through the domain is data-dependent, then global reduction cannot be used, and neighborhood communication is selected instead. For associative operations, swap based reduction is appropriate when data items are homogeneous, contiguous buffers that can be subdivided, and the user wants a distributed result, as in `MPI_Reduce_scatter`. When data items are heterogeneous and cannot be scattered, merge based reduction is used, similar to `MPI_Reduce`.

3 Method

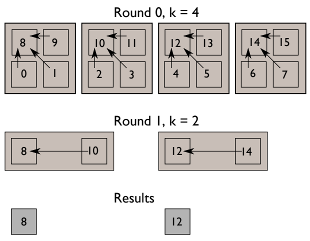
DIY's merge and swap based reductions allow configurable radix messaging. Communication occurs in rounds; and in each round, groups are formed of blocks that communicate with each other. The number of blocks per group in a round is called the *k-value*. By selecting the number of rounds and the k-value in each round, the user can tailor the communication pattern to the hardware characteristics of the architecture. One also can select a smaller number of rounds than a full reduction would require. Partial reductions are useful for some applications, such as simplification of topological structures [7].

3.1 Merge Based Reduction

The merge based communication pattern is used for associative reduction of heterogeneous data that cannot be readily distributed and instead must be merged in place at a smaller number of blocks during each round. Topological graph structures such as Morse-Smale complexes are reduced this way [7]. Algorithm 1 was first published in 2011 [4]. The inset at the right shows an example of a partial reduction with two rounds of merging using $k = 4$ in the first round and $k = 2$ in the second round.

Algorithm 1 Merge algorithm

```
1: mark all my local blocks as active
2: for all rounds do
3:     for all my local active blocks do
4:         identify blocks in same group as this block
5:         select one block of the group to be the root
6:         if block is not root of this group then
7:             post nonblocking send to the root block
8:             mark block as inactive
9:         else
10:            post nonblocking receive for all other blocks of the group
11:        end if
12:    end for
13:    wait for all sends/receives to complete
14:    for all local root blocks of groups do
15:        collect messages from blocks in this group
16:        call user-defined merge operation
17:    end for
18: end for
19: return number of finished blocks
```

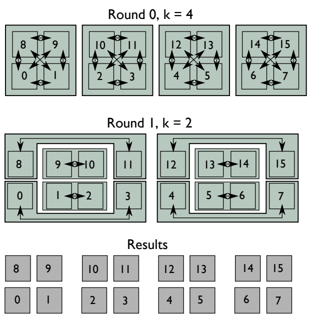


3.2 Swap Based Reduction

The swap based communication pattern is used for associative reduction of homogeneous contiguous data buffers that remain distributed instead of being merged into a smaller number of blocks. This case occurs in sort-last parallel rendering, when multiple image buffers are blended together. In fact, Algorithm 2 is a generalization of the radix-k image compositing algorithm first published in 2009 [10]. The inset at the right shows an example of a partial reduction with two rounds of swapping using $k = 4$ in the first round and $k = 2$ in the second round.

Algorithm 2 Swap algorithm

```
 1: for all rounds do
 2:     for all my local blocks do
 3:         identify other blocks in same group as this block
 4:         compute fraction of item to exchange
 5:         for all member blocks in same group as this block do
 6:             post asynchronous send of fraction of item
 7:         end for
 8:         for all member blocks in same group as this block do
 9:             post asynchronous receive of fraction of item
10:         end for
11:     end for
12:     wait for sends/receives to complete
13:     for all blocks do
14:         collect messages from blocks in this group
15:         call user-defined reduce operation
16:     end for
17: end for
18: return location of reduced fraction within each block
```



3.3 Neighborhood Communication

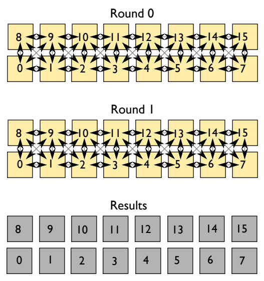
For nonassociative operators, information traverses a domain iteratively, one neighborhood at a time. An example is tracing streamlines through a flow dataset, when the communication pattern depends entirely on the input vector field. Algorithm 3 is a generalization of the particle exchange algorithm first published in 2011 [5]. The inset at the right shows an example of two rounds of neighborhood exchange. In the `PostMessages` procedure, blocks post nonblocking messages to their neighbors, and return to check on the status of received messages in the `TestMessages` procedure. The number of messages for which to wait during each call to `TestMessages` is adjustable, and this adjustable level of synchrony is a key reason for the performance improvement of this algorithm over its predecessors.

4 Performance

Our tests were run on *Intrepid*, a 557-teraflop IBM Blue Gene/P supercomputer operated by the Argonne Leadership Computing Facility (ALCF) at Argonne National

Algorithm 3 Neighborhood Exchange Algorithm

```
 1: procedure PACK MESSAGES
 2:     for all my local blocks do
 3:         for all all processes in my neighborhood do
 4:             pack message of block IDs and item counts destined for that process
 5:             pack message of item payloads destined for that process
 6:         end for
 7:     end for
 8: end procedure
 9: procedure POST MESSAGES
10:     for all packed ID and count messages do
11:         post nonblocking send of counts message
12:         post nonblocking send of payloads message
13:         post nonblocking receive of counts message
14:     end for
15: end procedure
16: procedure TEST MESSAGES
17:     while number of arrived messages < desired number of arrivals do
18:         wait for some more counts messages to arrive
19:         parse counts message and post blocking send for matching payload message
20:     end while
21: end procedure
```



Laboratory. The test program was compiled with the IBM `xlcxx_r` compiler using `-O3 -qarch=450d -qtune=450` optimizations.

4.1 Reduction

The parameters for our tests were chosen so that our results could be compared against MPI; hence, the merge and swap algorithms performed a full reduction. This means that the number of rounds and k-values per round produced a merged result in a single block, and the swapped result was scattered among all blocks and was equivalent to all blocks communicating with each other. We used one DIY block per MPI process and tested block counts that were powers of two. Tests were run in symmetric multiprocessor mode, one MPI process per node.

Since the swap based reduction is a generalization of the radix-k image compositing algorithm, we also wanted to configure our tests to be able to compare against radix-k. Thus, our reduction operator is the noncommutative *over* operator [11], a linear combination of elements in a floating-point buffer that represents the red, green, blue, and opacity channels of pixels in an image. Our message sizes are based on images of various resolutions at 16 bytes per pixel.

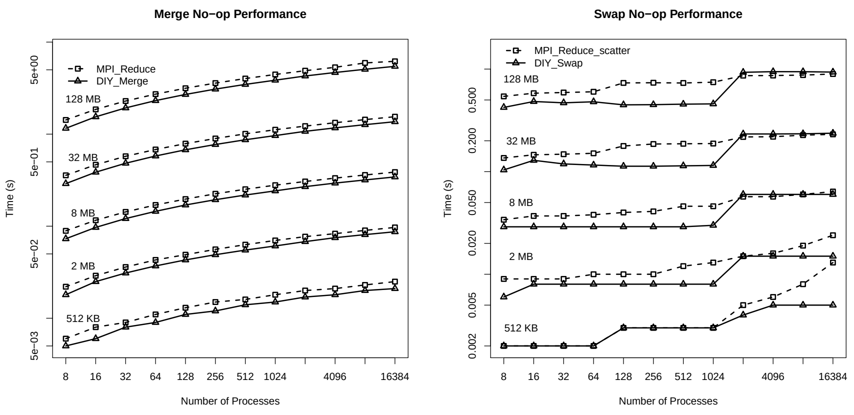


Fig. 1. Communication time only for our merge algorithm compared with MPI's reduction algorithm (left) and our swap algorithm compared with MPI's reduce-scatter algorithm (right).

We first disabled the reduction operator and tested only the communication cost. Figure 1 shows this result for merge and swap reduction compared with `MPI_Reduce` and `MPI_Reduce_scatter`, respectively. For merging, we found $k = 2$ to perform best; for swapping, $k = 8$ was used. In the merge test, DIY was approximately 10% faster than the BG/P MPI implementation; in the swap test, DIY was up to 60% faster at 1,024 processes.

Next, we enabled the reduction operator, with the results in Figure 2 for $k = 2$ merge reduction and $k = 8$ swap reduction. The difference between MPI and DIY is minimal

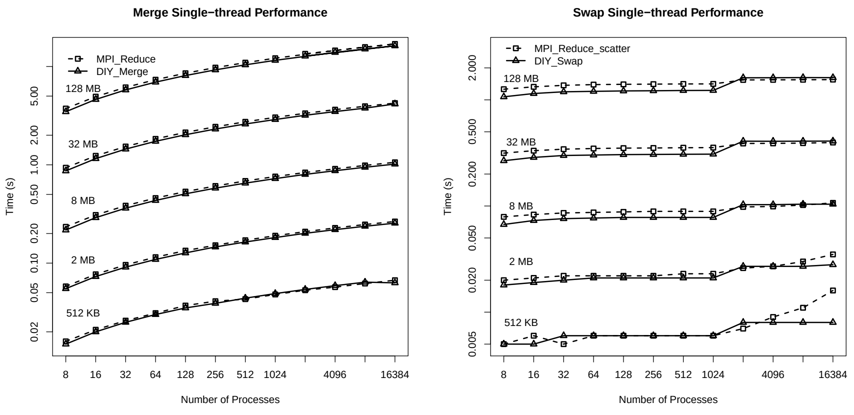


Fig. 2. Communication and single-threaded compositing operator for our merge algorithm compared with MPI's reduction algorithm (left) and our swap algorithm compared with MPI's reduce-scatter algorithm (right).

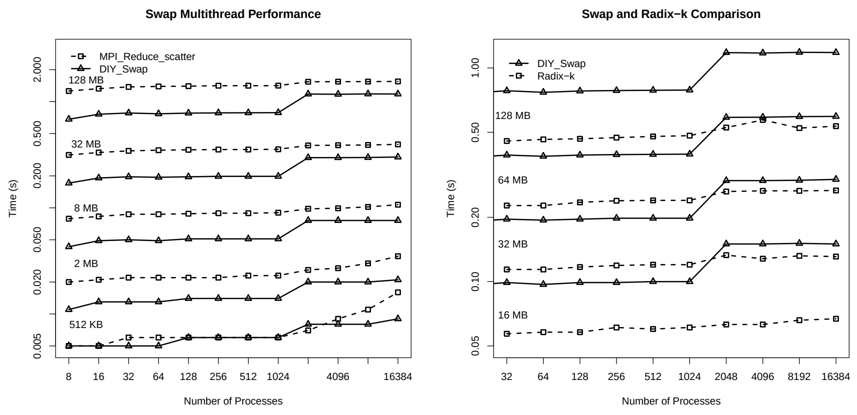


Fig. 3. Communication and multithreaded compositing operator for our swap algorithm compared with MPI's reduce-scatter algorithm (left) and compared with the radix-k algorithm of [10] (right).

because the cost of computing the over operator is expensive enough to mask the gains in the communication algorithm. Moreover, when $k = 8$, the computation is performed by looping over the eight blocks that need to be reduced locally, which serializes the computation.

Having eight blocks available for reduction, however, opens new possibilities for thread-level parallelism that did not exist when $k = 2$ or in `MPI_Reduce_scatter`. When the loop over the eight blocks is multithreaded with openMP in the DIY version, the graph on the left of Figure 3 results. The multithreaded DIY swap algorithm is up to 1.8 times faster than `MPI_Reduce_scatter` at 1024 processes, and approximately 1.4 times faster than the single-threaded DIY swap in Figure 2.

Within the local *over* operator of the DIY swap version, the outer loop over the blocks that were received was thread-parallelized, and this loop exists only in the DIY version. The inner loop over block elements remained serial in both DIY and MPI. Since the over operator is noncommutative, we wanted to ensure that the same reduction order

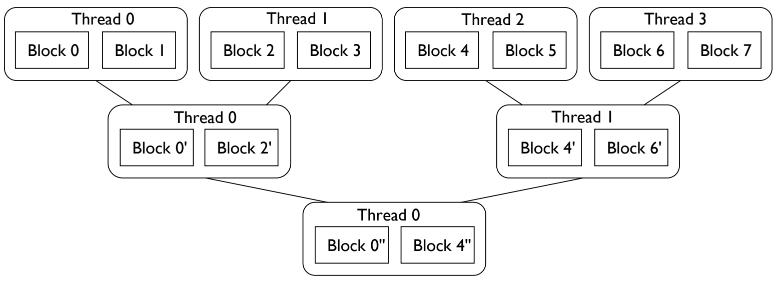
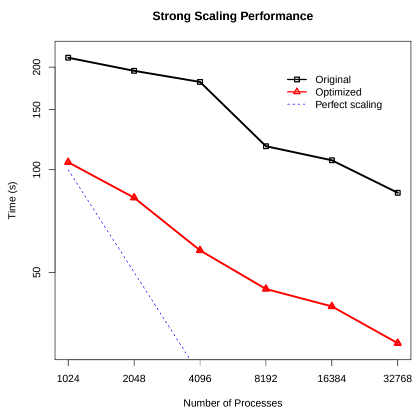


Fig. 4. Local multithreaded tree reduction of eight blocks.



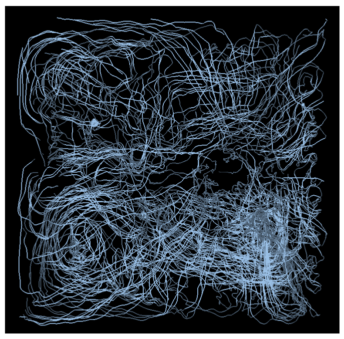


Fig. 5. Strong scaling performance for compute plus communicate time shows three times improvement over previously published results (left) for particles traced in a thermal hydraulics flow field (right).

was maintained in both versions. In our tests, this order is in increasing block global identification number. To maintain this order, we employed a local tree reduction as shown in Figure 4. The idea of reducing local blocks in a tree as opposed to a linear order was introduced by Moreland et al. [12], and we borrowed that idea for our thread ordering.

The right-hand side of Figure 3 compares our multithreaded swap performance with the standalone version of the the radix-k algorithm [10]. It shows that DIY's performance is approximately two times slower than radix-k. Our swap reduction is more general than radix-k because it supports multiple blocks per process and generic data items, and this generality comes with some overhead. We expect that some of this performance gap can be recovered through further optimization of DIY, and some will remain. Motivated by this comparison, we will continue to work to improve performance.

4.2 Neighborhood Exchange

To demonstrate the scalability of the nearest neighbor communication algorithm, and in particular its use of tunable synchronization, we present an example from parallel particle tracing. A common and intuitive way to visualize a static or time-varying flow field is to trace paths that are derived from the trajectory of massless particles injected into the field and advected through it using numerical integration. In a data-parallel distributed-memory environment, the communication pattern that results is a neighborhood exchange. Local computation of integral curves within a block is interleaved with the exchange of particles across block boundaries in an iterative fashion.

The test shown in Figure 5 was run in virtual node mode with one MPI process per core and eight DIY blocks per process. The left side of the figure shows strong scaling of the compute plus communicate time and excludes file I/O. It compares Algorithm 3 with a previous algorithm published in 2011 [5]. The main improvement is due to the adjustable number of arrivals parameter in line 17 of Algorithm 3. This algorithm is approximately three times faster than the original in 2011.

The test dataset comes from a computational fluid dynamics simulation of thermal hydraulics in a nuclear reactor. The problem is large in data size (2048^3 grid points), in the number of particles traced (256 K), and in the computation applied to each particle (1000 integration steps). While 0.25 million particles are too many to visualize, a very dense tracing such as this is necessary for accurate follow-on analysis of the field lines. A much smaller number of particles traced in the same flow field is shown in the right side of Figure 5.

5 Summary

We presented three communication patterns that are common to many data analysis tasks. For global reduction, we designed configurable algorithms for merging and swapping that feature configurable number of rounds and k-value per round. The neighborhood exchange pattern features a configurable degree of synchronization and flexible identification of blocks that constitute a neighborhood. Implemented in a design that communicates user-defined data items among blocks instead of processes, the result is a set of versatile communication algorithms that have proven to be very useful in numerous data analysis applications.

Performance and scalability benchmarks were presented for all three algorithms. We compared the two reductions with MPI. While we designed the experiment to be comparable with MPI's reductions (one block per process and full reduction), it is important to realize that DIY provides richer functionality that in general cannot be expressed by a few MPI calls. Nevertheless, our algorithms were faster than the MPI implementation in almost all the cases tested. The neighborhood exchange was compared with an earlier algorithm, with three times faster performance in a test of parallel particle tracing of a scientific dataset.

DIY's versatility also accounts for lower performance compared with single-purpose algorithms such as radix-k for image compositing. In particular, DIY does not overlap communication and computation deep in the communication loop the way radix-k does, because the reduction operator is in the user's code.

In our ongoing work, we are continuing to look for ways to overlap communication and computation in our general-purpose library, to approach the performance of algorithms like radix-k. We are also continuing to add new features to DIY, including versatile information exchange patterns within a neighborhood. For example, blocks may talk to only a subsets of blocks within a neighborhood, and these subsets can be chosen in various ways. We also continue to build new analysis applications on top of DIY, which in turn drives further innovation in the library.

Acknowledgments

We gratefully acknowledge the use of the resources of the Argonne Leadership Computing Facility at Argonne National Laboratory. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. Work is also supported by the DOE Office of Science, Advanced Scientific Computing Research award No. DE-FC02-06ER25777, program manager Lucy Nowell.

References

1. Kumar, S., Dozsa, G., Berg, J., Cernohous, B., Miller, D., Ratterman, J., Smith, B., Heidelberger, P.: Architecture of the Component Collective Messaging Interface. In: Euro PVM/MPI '08: Proceedings of the 15th annual European PVM/MPI Users' Group Meeting, New York, NY, USA, Springer (2008) 23–32
2. Sack, P., Gropp, W.: Faster Topology-Aware Collective Algorithms through Non-Minimal Communication. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '12, New York, NY, USA, ACM (2012) 45–54
3. Ma, Kwan-Liu and Painter, James S. and Hansen, Charles D. and Krogh, Michael F.: Parallel Volume Rendering Using Binary-Swap Compositing. IEEE Computer Graphics and Applications **14**(4) (1994) 59–68
4. Peterka, T., Ross, R., Kendall, W., Gyulassy, A., Pascucci, V., Shen, H.W., Lee, T.Y., Chaudhuri, A.: Scalable Parallel Building Blocks for Custom Data Analysis. In: Proceedings of the 2011 IEEE Large Data Analysis and Visualization Symposium LDAV'11, Providence, RI (2011)
5. Peterka, T., Ross, R., Nouanesengsy, B., Lee, T.Y., Shen, H.W., Kendall, W., Huang, J.: A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields. In: Proceedings of IPDPS 11, Anchorage AK (2011)
6. Xu, L., Lee, T.Y., Shen, H.W.: An Information-Theoretic Framework for Flow Visualization. IEEE Transactions on Visualization and Computer Graphics **16** (November 2010) 1216–1224
7. Gyulassy, A., Peterka, T., Pascucci, V., Ross, R.: Characterizing the Parallel Computation of Morse-Smale Complexes. In: Proceedings of IPDPS '12, Shanghai, China (2012)
8. Schaap, W.E.: DTFE: The Delaunay Tesselation Field Estimator, University of Groningen, The Netherlands (2007) Ph.D. Dissertation.
9. Chen, J., Silver, D., Jiang, L.: The Feature Tree: Visualizing Feature Tracking in Distributed AMR Datasets. In: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics. PVG '03, Washington, DC, USA, IEEE Computer Society (2003)
10. Peterka, T., Goodell, D., Ross, R., Shen, H.W., Thakur, R.: A Configurable Algorithm for Parallel Image-Compositing Applications. In: Proceedings of SC 09, Portland OR (2009)
11. Porter, T., Duff, T.: Compositing Digital Images. In: Proceedings of 11th Annual Conference on Computer Graphics and Interactive Techniques. (1984) 253–259
12. Moreland, K., Kendall, W., Peterka, T., Huang, J.: An Image Compositing Solution at Scale. In: Proceedings of SC11, Seattle, WA (2011)