Visualizing MPI Collective Communication

Christopher Atala
Department of Computer Science
Wake Forest University
Winston-Salem, NC, USA
atalcd18@wfu.edu

Meredith Morrison
Department of Computer Science
Wake Forest University
Winston-Salem, NC, USA
morrmg20@wfu.edu

Grey Ballard
Department of Computer Science
Wake Forest University
Winston-Salem, NC, USA
ballard@wfu.edu

***Abstract*—Communication collectives are at the heart of distributed-memory parallel algorithms and the Message Passing Interface. In parallel computing courses, students can learn about collectives not only to utilize them as building blocks to implement other algorithms, but also as exemplars for designing and analyzing efficient algorithms. We develop a visualization tool to help students understand different algorithms for collective operations as well as evaluate and analyze the algorithms' efficiencies. Our implementation is written in C++ with OpenMP and uses the Thread Safe Graphics Library. We simulate distributed-memory message passing to implement the algorithms, and the threads concurrently illustrate their local memories and message passing using a shared canvas. Our tool includes visualizations of different algorithms for Scatter, Gather, ReduceScatter, AllGather, Broadcast, Reduce, AllReduce, and AlltoAll.**

I. Introduction

Nearly all distributed-memory parallel algorithms involve collective communication operations such as Broadcast and Reduce. Parallel computing courses that include Message Passing Interface (MPI) programming must educate students on the functionality of the most commonly used collectives. We contend that studying the design and analysis of algorithms for these collectives is an ideal way to present MPI-based cost models and illustrate trade-offs between network bandwidth and latency costs. By understanding collectives and evaluating the algorithms that implement them, students are better equipped to design and analyze parallel algorithms for more complex problems. In this work, we develop a visualization tool to illustrate the underlying algorithms that implement the most commonly used collectives. This tool enables students to more easily understand, analyze, and compare alternative algorithm design decisions.

Despite the apparent simplicity of collective operations, the most efficient algorithm for implementing each collective is not always obvious. We address the Broadcast collective in Sec. II and focus on its algorithms as an example concept that our visualization tool helps to elucidate. A Broadcast starts with data in only one processor's memory and ends with the data stored in all processors' memories. Most advanced students are able to observe the benefits of the so-called binomial tree Broadcast algorithm: at each step of the algorithm, each processor with the data simultaneously shares it with a unique processor that does not have it. Each step of the algorithm doubles the number of processors with the data, so it requires a logarithmic number of steps. It is less obvious that the binomial tree Broadcast algorithm is suboptimal in terms of the amount of data communicated, as counted along the critical path. We present an alternative approach in Sec. II and analyze its costs compared with the binomial tree algorithm.

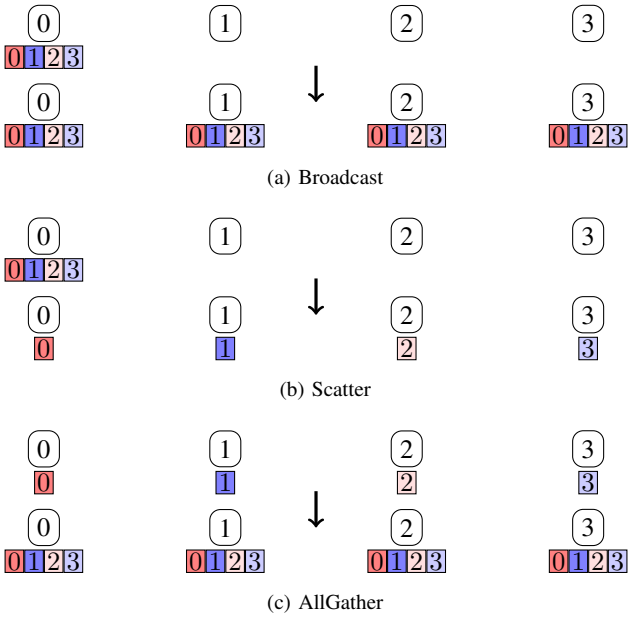
We describe the details of the implementation of our visualization tool in Sec. III. Our code utilizes the Thread Safe Graphics Library (TSGL) [1], an OpenMP-based library designed for pedagogical demonstrations of concurrency and parallel computing. TSGL provides an interface to augment shared-memory parallel programs, allowing threads to concurrently draw to a shared canvas to illustrate computations as they execute them. In order to visualize distributed-memory algorithms, we divide the canvas across the processors' local memories and depict the data stored by each processor. We simulate MPI sends and receives using shared memory, and we augment those functions to animate messages exchanged between processors as they occur. As an example use of the tool, we show how the alternative algorithms for Broadcast are visualized and compared.

As we describe in Sec. IV, our tool is more general and includes implementations of eight of the most commonly used collectives. The data size and number of processors are configurable, and we provide options to visualize alternative algorithms for some collectives such as Broadcast. The intended use of the tool is for in-class demonstrations as well as student interaction and code extension. We plan to incorporate our implementation with the public TSGL repository for broader dissemination.

II. Bandwidth-Efficient Broadcast Algorithm

We describe here an efficient algorithm for broadcasting data in the MPI model. For more details and descriptions of efficient algorithms for a wider range of collectives, see [2], [3]. In the standard MPI model, the communication cost for sending data of size n from a processor to any other processor is approximated as $\alpha+\beta n$, where α is the per-message latency and β is the per-word (inverse) bandwidth. Each processor may communicate with only one processor at a time, but pairs of processors can communicate simultaneously with no resource contention. The Broadcast collective is shown in Fig. 1a: one processor starts with all the data, and after the collective, all processors own the data.

This work is supported by the National Science Foundation under Grant Numbers CCF-1942892 and OAC-2106920. CA and MM are co-first authors.

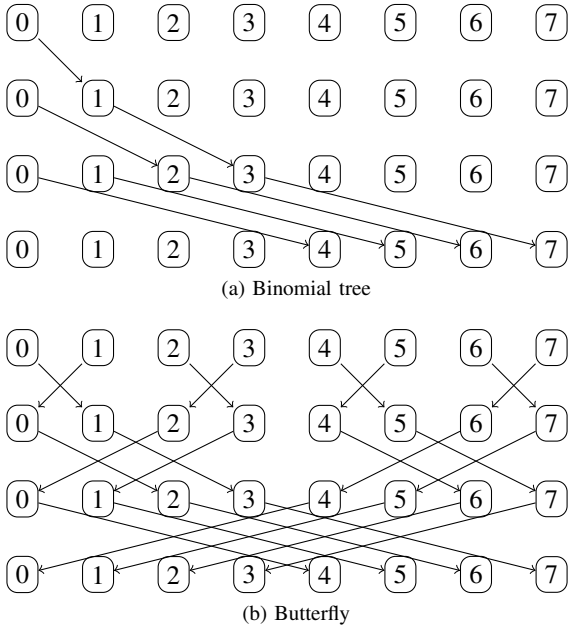


(a) Broadcast

(b) Scatter

(c) AllGather

Fig. 1. Inputs/outputs of three collective communication operations



(a) Binomial tree

(b) Butterfly

Fig. 2. Binomial tree and butterfly communication patterns

The simplest algorithm for broadcasting data from one processor to all other processors is for the root processor to send the data individually to each other processor. The cost of this algorithm for data of size n across P processors is $(P-1)(\alpha+\beta n)$. We can reduce this cost by exploiting the fact that processors can communicate simultaneously: at each step, we can double the number of processors that have the data if all processors with data communicate with unique partners that do not yet have the data. This algorithm is known as the binomial tree algorithm, which is given as Alg. 1 and uses the communication pattern depicted in Fig. 2a. It has cost $\lceil \log P \rceil (\alpha + \beta n)$, which is significantly cheaper.

Algorithm 1 Binomial Tree Broadcast

```
1:  function BinomialTreeBroadcast(buf, cnt)
2:      rank ← MPI_Comm_rank
3:      size ← MPI_Comm_size
4:      for i ← 0 to log2(size) − 1 do
5:          if rank < 2^i then
6:              Send(buf, cnt, rank + 2^i)
7:          else if rank < 2^(i+1) then
8:              Recv(buf, cnt, rank − 2^i)
9:          end if
10:     end for
11: end function
```

When n is large, we can further reduce this cost with a more clever idea, using a combination of two other collectives: Scatter and AllGather. The Scatter collective (Fig. 1b) starts with all data on one processor and ends with each processor owning $1/P$th of the data. The AllGather collective (Fig. 1c) starts with data partitioned across all processors and ends with all processors owning all the data. Thus, for sufficiently large n, Scatter followed by AllGather achieves the desired Broadcast collective. Scatter can be implemented using a binomial tree communication pattern as shown in Fig. 2a; AllGather is performed using a butterfly communication pattern (also known as bidirectional exchange) as shown in Fig. 2b. Simplified versions of these algorithms are given as Algs. 2 and 3, where the number of processors is a power of two, the root is rank 0, and the same buffer is used for input and output. The advantage of using these two collectives is that their costs are smaller than the binomial tree Broadcast. The sizes of the messages in these collectives vary from $n/2$ down to n/P, where n is the amount of data being broadcast. This pattern is sometimes referred to as recursive halving/doubling. Assuming P is a power of two, each has cost

$$\sum_{i=0}^{\log P-1} \left(\alpha + \beta\, 2^i \cdot \frac{n}{P}\right) = \alpha \log P + \beta \frac{P-1}{P} n.$$

Thus, the latency cost of the Scatter/AllGather combination is twice that of the binomial tree Broadcast, but the bandwidth cost is reduced by a factor of $O(\log P)$.

III. TSGL Visualization of MPI Collectives

The Thread-Safe Graphics Library (TSGL) [1] is written in C++ and OpenMP, and it enables multithreaded visualization in real time. Implementations of parallel algorithms can be augmented with calls to TSGL functions so that threads can concurrently illustrate and animate a shared canvas as they progress through the computation. The TSGL repository (https://github.com/Calvin-CS/TSGL) includes examples of parallel algorithms for computations such as image processing [4], dynamic programming [5], sorting [6], and others.

In order to use TSGL to visualize a distributed-memory parallel operation like a collective, we simulate the communication functionality of MPI. We maintain thread-private variables to simulate the local memory spaces of every thread

Algorithm 2 Binomial Tree Scatter

function BINOMIALTREESCATTER(buf, cnt)
  rank ← MPI_COMM_RANK
  size ← MPI_COMM_SIZE
  for $i \leftarrow \log_2(\texttt{size}) - 1$ down to 0 **do**
    if rank mod $2^{i+1} == 0$ **then**
      $\texttt{sndbuf} = \texttt{buf} + \texttt{rank} + \texttt{cnt} \cdot 2^i$
      SEND(sndbuf, cnt · 2^i, rank + 2^i)
    else if rank mod $2^i == 0$ **then**
      $\texttt{rcvbuf} = \texttt{buf} + \texttt{rank}$
      RECV(rcvbuf, cnt · 2^i, rank − 2^i)
    end if
  end for
end function

Algorithm 3 Butterfly AllGather

function BUTTERFLYALLGATHER(buf, cnt)
  rank ← MPI_COMM_RANK
  size ← MPI_COMM_SIZE
  for $i \leftarrow 0$ to $\log_2(\texttt{size}) - 1$ **do**
    if rank mod $2^{i+1} < 2^i$ **then**
      $\texttt{dest} = \texttt{rank} + 2^i$
      $\texttt{sndbuf} = \texttt{buf} + \lfloor \texttt{rank}/2^{i+1} \rfloor \cdot 2^{i+1} \cdot \texttt{cnt}$
      $\texttt{rcvbuf} = \texttt{sndbuf} + 2^i \cdot \texttt{cnt}$
    else
      $\texttt{dest} = \texttt{rank} - 2^i$
      $\texttt{rcvbuf} = \texttt{buf} + \lfloor \texttt{rank}/2^{i+1} \rfloor \cdot 2^{i+1} \cdot \texttt{cnt}$
      $\texttt{sndbuf} = \texttt{rcvbuf} + 2^i \cdot \texttt{cnt}$
    end if
    EXCHANGE(sndbuf, rcvbuf, $2^i \cdot$ cnt, dest)
  end for
end function

(MPI process), and we introduce two shared arrays to simulate the fully connected network. These variables are a $P \times P \times N$ array called `network`, in which every processor has a buffer of size N used to send data to every other processor, and a $P \times P$ array of flags, so that threads can synchronize when sending and receiving data.

We implement `Send` and `Recv` functions that mimic the blocking send and receive functions in MPI. Simplified versions of these functions are given in Fig. 3. As shown in Fig. 3a, a sending thread copies data from a local buffer to the shared network in a location dedicated to the source/destination pair. Then, it sets a flag to indicate to the receiving thread that it can read from the shared network. To simulate a blocking send, the sending thread waits until the receiving thread has copied the data from the network buffer and resets the flag before returning. Figure 3b shows the code for the receiving thread: it waits until the sending thread fills the network buffer, sets the flag, copies the data to a local buffer, and resets the flag. We indicate where our implementation drives the visualization within the `Send` function.

On the TSGL canvas, we create a background where we arrange the local memories of the processors horizontally and

```
// Function to send data
void Send(int* buf, int cnt, int dest,
          int** flag, int*** network) {

    int src = omp_get_thread_num();
    int received = 0;

    // Copy data from local buffer to network
    memcpy(network[src][dest], buf, cnt*sizeof(int));

    /* animate message using TSGL subroutines */

    // Mark data as sent
    flag[src][dest] = 1;

    // Wait until data is received
    while (!received) {
        #pragma omp atomic read
        received = flag[src][dest];
    }
}
```

(a) Send

```
// Function to receive data
void Recv(int* buf, int cnt, int src,
          int** flag, int*** network) {

    int dest = omp_get_thread_num();
    int sent = 0;

    // Wait until data is sent
    while (!sent) {
        #pragma omp atomic read
        sent = flag[src][dest];
    }

    // Copy data from network to local buffer
    memcpy(buf, network[src][dest], cnt*sizeof(int));

    // Mark data as received
    flag[src][dest] = 0;
}
```

(b) Receive

Fig. 3. Shared memory simulation of MPI_Send and MPI_Recv

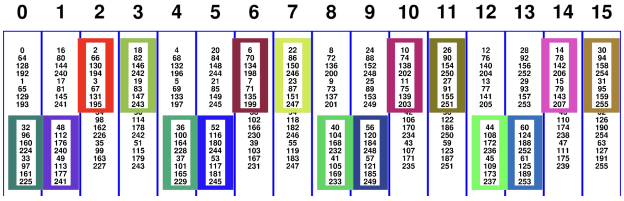


Fig. 4. Example canvas during AlltoAll using butterfly pattern

list their local data vertically in columns. Figure 4 shows an example visualization for 16 processors performing an AlltoAll collective. During the animation, messages, depicted as data contained in colored rectangles, move horizontally between processors from sender to receiver. At this step of the algorithm shown in Fig. 4, processors 0 and 2 exchange data, processors 1 and 3 exchange data, and so on. In order for messages to avoid crossing paths, messages moving left to right are drawn higher than messages moving right to left. After the data transfer is complete, the animation displays the resulting data distribution across the processors.

We now demonstrate how the visualization tool can be used to compare the binomial tree and bandwidth-efficient Broadcast algorithms. Figure 5 shows screen captures of each step of the algorithms run with $P = 8$ processors and $n = 8$ elements. The binomial tree algorithm is shown on the left in Figs. 5a, 5b and 5c. The alternative algorithm is shown in two phases: the Scatter is depicted in the middle in Figs. 5d, 5e

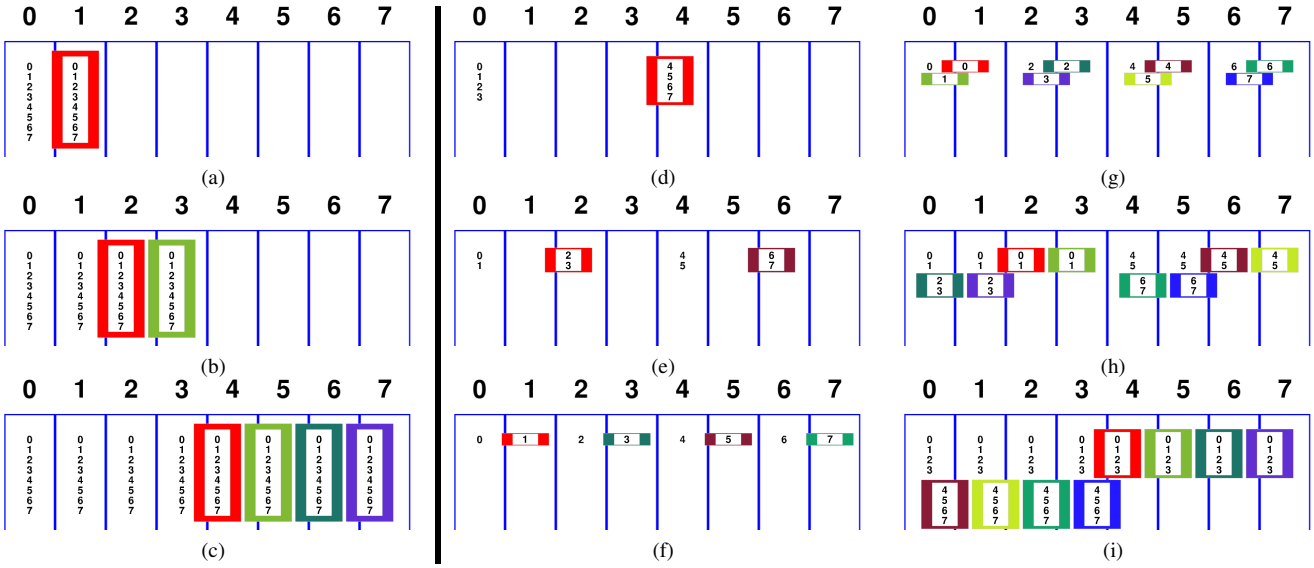


Fig. 5. Visualization of Broadcast algorithms. Left: Binomial-tree Broadcast; Right: Broadcast using Scatter/AllGather.

and 5f, and the AllGather is given on the right in Figs. 5g, 5h and 5i. Each of the three collectives require $\log_2 P = 3$ steps.

Note that in the case of the binomial tree algorithm, the message size is 8 at each step. In the first step only processor 0 sends data to processor 1, in the second step processors 0 and 1 simultaneously send to processors 2 and 3, and in the third step the first 4 processors simultaneously send messages to the last 4 processors. The cost is thus $3\alpha + 24\beta$.

The Scatter phase of the bandwidth-efficient algorithm is given in the middle column of Fig. 5. At each step, we see that each processor with data sends half of its data to a partner processor, so the sizes of the messages decrease from 4 to 2 to 1 in the so-called recursive halving pattern. After the last step, the data to be broadcast is distributed across the processors. The AllGather phase is given in the right column. Here, partner processors exchange data at each step, so that the size of the local data doubles. In this recursive doubling process, the message sizes increase from 1 to 2 to 4. The cost of this Broadcast algorithm is thus $6\alpha + 14\beta$.

This example illustrates the bandwidth-latency trade-off between the two algorithms as analyzed in Sec. II. While not demonstrable in screenshots, the visualization sets the speed of the messages based on a configurable ratio between α and β. In this way, students can observe the difference in time: when latency cost dominates, the binomial tree algorithm is faster by a factor of approximately 2, and when bandwidth cost dominates, the Scatter/AllGather approach is faster by a factor of approximately $\frac{1}{2}\log P$.

IV. Conclusion

The communication collectives visualization tool we present here can be used in parallel computing classes that include distributed-memory parallel algorithms and/or MPI programming. Using the tool will help students to understand, analyze, and compare different algorithms for implementing the most fundamental collectives. While we focus on the Broadcast collective in this paper (as well as the use of Scatter and AllGather), the tool also includes visualizations of Gather, ReduceScatter, Reduce, AllReduce, and AlltoAll. It is configurable, so that instructors can solicit suggestions from students during a demonstration to experiment with changes in data size, number of processors, and ratio of α and β parameters. The code is also extensible, and we plan to use it in the future as a basis for student projects. For example, the implementation is currently limited to powers-of-two number of processors and data sizes that are divisible by the number of processors. Our future work is to generalize the code to remove many of these limitations, and we plan to involve new parallel computing students in these efforts.

References

[1] J. C. Adams, P. A. Crain, and M. B. Vander Stel, "TSGL: A Thread Safe Graphics Library for visualizing parallelism," *Procedia Computer Science*, vol. 51, pp. 1986–1995, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915012715

[2] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005. [Online]. Available: https://doi.org/10.1177/1094342005051521

[3] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007. [Online]. Available: http://dx.doi.org/10.1002/cpe.1206

[4] J. C. Adams, P. A. Crain, and C. P. Dilley, "Seeing multithreaded behavior using TSGL," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 972–977. [Online]. Available: https://www.computer.org/csdl/proceedings-article/ipdpsw/2016/3682a972/12OmNyaoDyH

[5] G. Ballard and S. Parsons, "Visualizing parallel dynamic programming using the Thread Safe Graphics Library," in *Proceedings of the IEEE/ACM Ninth Workshop on Education for High Performance Computing*, ser. EduHPC '21, 2021, pp. 24–31. [Online]. Available: https://doi.org/10.1109/EduHPC54835.2021.00009

[6] C. Wiley and G. Ballard, "Visualizing PRAM algorithm for mergesort," in *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, ser. EduPar '24, 2024, pp. 365–368. [Online]. Available: https://doi.org/10.1109/IPDPSW63119.2024.00083