Modified Bruck/Dissemination Algorithm for Commutative Reduction in MPI

Jeremias Lang

July 19, 2025

1 Overview

This document describes a custom implementation of a modified Bruck/Dissemination-style algorithm to support commutative reduction operations in the Message Passing Interface (MPI). The implementation includes:

- A baseline algorithm using `MPI_Allgather` and pairwise merging.
- A modified version of the Bruck/Dissemination algorithm to allow partial aggregation in each round with the same communication pattern.
- An alternative circulant graph based variant.

All implementations assume that each process begins with a sorted block of integers and aim to produce a globally sorted sequence using only commutative operations.

2 Algorithm 0 - Baseline Implementation

The baseline implementation uses the `MPI_Allgather` function to gather the sorted blocks from all processes into a single array of size m. A pairwise merge tree is then performed in a ping-pong manner using one additional buffer of the same size. One array serves as the input, while the other holds the merged output after each round. The pointers to these arrays are swapped for the next round, so no in-place copying is required in each merging round. The final data array is copied back once into the original `recvbuf` (allocated by the calling process) for correctness checks.

2.1 Time And Space Complexities

Each merge round doubles the block size and merges adjacent pairs of blocks via a custom simple merge function `mergeInts`. The merging proceeds in a bottom-up tree fashion, leading to $\mathcal{O}(\log p)$ rounds for p processes, as the block (and step) size doubles each round. Once the last round completes, the elements are in the globally sorted order. The initial cost of `MPI_Allgather` is assumed to be $\mathcal{O}(n + \log p)$, where n is the total message size.

The merging of p sorted blocks, each of size $\frac{n}{p}$, has the optimal time complexity of $\mathcal{O}(n \log p)$. In each round, all n elements are merged via pairwise calls to `mergeInts`, which is linear in n. Because the number of rounds is in $\mathcal{O}(\log p)$, the total merging cost is $\mathcal{O}(n \log p)$. After each round, the pointers to the input and output arrays are swapped in constant time, and a final copy of size $\mathcal{O}(n)$ is performed back into the original array. Therefore, the overall time complexity is $\mathcal{O}(n \log p)$.

Each round performs merges on contiguous blocks in a cache-friendly manner. The idea of merging the data in place, without allocating a temporary buffer, was discarded as there is a possibility of a higher time complexity $\mathcal{O}(n \log n)$ when calling `std::inplace_merge` instead of `mergeInts`. The extra space required is exactly n for the temporary buffer. While doing a direct p-way merge via min-heaps for example also achieves $\mathcal{O}(n \log p)$ asymptotically, they could bring additional overhead from creating and managing the min-heaps and would not use contiguous blocks of data.

3 Algorithm 1 - Custom Bruck's Implementation

The implementation of the custom Bruck's algorithm [2] is outlined in Algorithm 1.
The primary issue with applying the Bruck's algorithm to the AllReduce problem arises when the number of processes, p, is not a power of two. Specifically, during the final round, processes need to send only the first $l = p - 2^{\lfloor \log_2 p \rfloor}$ blocks from their received order to their communication partner. Sending the entire merged list causes receivers to obtain redundant data, resulting in duplicates upon merging. Normally, this issue does not occur when blocks are simply redistributed without an aggregation operation. However, when merging is involved, information about the initial ownership of data blocks is lost. Therefore, to preserve the original communication logic of Bruck's algorithm and retain the same communication partners across rounds, additional information must sometimes be transmitted in specific rounds. This ensures the construction of a separate partially merged list sent only in the final round.

Luckily, it is unnecessary to send all original unmerged blocks at each stage in addition to the merged data. Given the aggregating nature of Bruck's algorithm, the necessary additional data to send and save at each stage can be identified by iteratively subtracting the largest power of two less than or equal to the current remainder until zero is reached, starting with l. At each iteration, the number of additional blocks that must be sent is given by the remainder, and the round in which this occurs corresponds to the exponent of the nearest power of 2. For instance, with $p = 9$ follows $l = 1$. The nearest power of two is 1 with remainder 0 with round 0, meaning that the block received in the first communication round can simply be copied directly into the `partial` buffer for sending in the final round. This recursive subtraction process effectively retraces the communication steps of Bruck's algorithm to determine the minimum additional data necessary to be sent separately for each round.

In the worst case, when p is one below the nearest power of two, the number of special rounds requiring additional data transmission is in $\mathcal{O}(\log p)$ as the remainder after subtracting powers of two decreases then by half each round. Furthermore, the additional data sent each round cannot exceed half the size of the normal merged list transmitted in the standard algorithm. Hence, the total extra data sent is at most half the original data sent during Bruck's algorithm.

Initially, the computation of the special rounds and respective block sizes is performed in $\mathcal{O}(\log p)$. Each processor holds its own initial data of size $m = n/p$ at the start. Three additional buffers are needed: the `local` and `recvblocks` buffers, each preallocated to size $m \cdot p$, and the `partial` buffer, preallocated to $\lceil \frac{m \cdot p}{2} \rceil$. The `merged` buffer can reuse the `recvbuf` provided by the MPI function call. To avoid costly memory reallocations, these buffers are preallocated using `malloc()`, and buffer indices determining the needed elements are tracked manually.
Additionally, this approach could potentially require less memory when handling problems where the lists contain larger data types, by managing and sending an index array that indicates which processor each element of the data list belongs to. In this way, only

the index array needs to be additionally communicated, rather than the full data lists. However, since this introduces at least linear overhead for managing the index arrays when new data gets merged, it was not implemented, as the data type used was `int`, making this approach bring no benefit compared to just sending the data itself.

Algorithm 1 ALLGATHERMERGEBRUCK executed by processor P out of p. This algorithm solves the problem of the standard Bruck's algorithm when applying a commutative operation like merging in each round by sending additional data.

```
procedure AllGatherMergeBruck($V[m]$, $P$, $p$)    ▷ $V$ - sorted local block
    $\ell \leftarrow p - 2^{\lfloor \log_2 p \rfloor}$, $R \leftarrow \ell$, $L \leftarrow [\,]$
    while $R > 0$ do
        $n \leftarrow \lfloor \log_2 R \rfloor$    ▷ Round number
        $r \leftarrow R - 2^n$    ▷ Number of blocks
        append $(n, r)$ to $L$
        $R \leftarrow r$
    end while

    $q \leftarrow \lfloor \log_2 p \rfloor$
    $local \leftarrow [m * p]$
    copy $V$ into $local$    ▷ copy initial block
    $recvblocks \leftarrow [m * p]$
    $merged \leftarrow [m * p]$
    $partial \leftarrow [\lceil m * p/2 \rceil]$
    $idx \leftarrow 0$
    for $k = 0$ to $q - 1$ do
        $r \leftarrow -1$
        $s_k \leftarrow 2^k$
        $t, f \leftarrow (P - s_k + p) \bmod p,\ (P + s_k) \bmod p$
        if $k == L[idx].n$ then    ▷ check if special round with extra send blocks r
            $r \leftarrow L[idx].r$
            $idx \leftarrow idx + 1$
        end if
        if $r > 0$ then    ▷ check if extra blocks have to be sent this round
            append $partial$ to $local$
        end if
        Send($local, t$) || Recv($recvblocks, f$)
        $merged \leftarrow$ Merge($local, recvblocks$)
        if $r > 0$ then
            $partial \leftarrow$ Merge($local, recvblocks[partial]$)    ▷ merge with $partial$ part at end of $recvblocks$
        else if $r == 0$ then
            copy $local$ into $partial$ at front    ▷ preparing $partial$ being sent at a later round
        end if
        swap($local$, $merged$)    ▷ pointer swap — no copy
    end for
    if $\ell \neq 0$ then    ▷ final additional round
        $s_q \leftarrow 2^q$
        $t, f \leftarrow (P - s_q + p) \bmod p,\ (P + s_q) \bmod p$
        Send($partial, t$) || Recv($recvblocks, f$)
        $merged \leftarrow$ Merge($local, recvblocks$)    ▷ $merged$ holds the final sorted global list
    end if
end procedure
```

3.1 Time And Space Complexities

The main loop executes exactly $\lfloor \log_2 p \rfloor$ rounds as in the standard Bruck's algorithm, differing only when a special round requires additional block transmissions. In such rounds, if extra blocks must be sent, the `partial` buffer is appended to the `local` buffer before the combined data is transmitted using `MPI_Sendrecv` to communication partners. After receiving the data, the original `local` buffer (excluding the appended `partial` buffer) is merged with the received buffer into the `merged` buffer. Then, if additional data was transmitted, it is handled separately by merging only the extra received data at the end of the receive buffer with the original `local` buffer into the `partial` buffer. If a special round requires no extra block transmissions, the `local` buffer is simply copied into the `partial` buffer. At the end of each iteration, pointers to the `local` and `merged` buffers are swapped, eliminating unnecessary copying for later rounds. In a final additional round when p is not a power of two, the `partial` buffer alone is transmitted and merged with `local` data into the final sorted list.

The algorithm needs the same number of communication rounds as the standard Bruck's algorithm, namely $\mathcal{O}(\log p)$. Computing special rounds and block sizes to be sent occurs once initially in $\mathcal{O}(\log p)$. Merging is executed in linear time and manual buffer management ensures constant-time index tracking. Each special round involves at most one additional, smaller merge operation, therefore merging complexity remains $\mathcal{O}(m_r)$ per round, where m_r is the size of the merged list in each round. Each round involves a single `MPI_Sendrecv` call with a cost of $\alpha + \beta m_r$, with at most an additional $\frac{m_r}{2}$ data sent in special rounds, preserving the asymptotic complexity. In the last step, it could be necessary to copy the final list to the correct address `recvbuf` (allocated from the calling process). This is avoided by first computing if the number of rounds is even or odd, and then determining on that if `recvbuf` should be the `local` buffer or the `merged` buffer, ensuring that the final merge operation always copies into `recvbuf` automatically. As all send and receive operations share the same asymptotic complexity as the standard Bruck's algorithm (send at worst 1.5 times the data), the total communication complexity remains the same: $\mathcal{O}(\lceil \log p \rceil \cdot \alpha + (p-1) \cdot 1.5 \cdot m \cdot \beta) = \mathcal{O}(\lceil \log p \rceil \cdot \alpha + (p-1) \cdot m \cdot \beta)$. The merging computations also match the complexity of the original Bruck's algorithm (assuming $p =$ power of 2). Specifically, the merge cost doubles each round, starting from m in the first round up to $m \cdot p$ in the final $(\log p)$ round. Summing these across all rounds reveals the total merge cost:

$$m + 2m + 4m + \cdots + m \cdot p = m \cdot p + m \cdot (p-1) = m \cdot (2p-1)$$

Thus, the total merging complexity is $\mathcal{O}(m \cdot p)$, which is equivalent to $\mathcal{O}(n)$. Combining both communication and merging costs, the total runtime complexity of the algorithm is:

$$\mathcal{O}(\lceil \log p \rceil \cdot \alpha + (p-1) \cdot 1.5 \cdot m \cdot \beta + 1.5 \cdot m \cdot p) = \mathcal{O}(\lceil \log p \rceil \cdot \alpha + (p-1) \cdot m \cdot \beta + m \cdot p)$$

The total space complexity is $\mathcal{O}(3 \cdot m \cdot p) = \mathcal{O}(m \cdot p) = \mathcal{O}(n)$, requiring the `recvbuf` provided by the calling process plus three additional buffers—`local`, `recvblocks`, and `partial`—each preallocated with a maximum size of $m \cdot p$.

4 Algorithm 2 - Circulant Implementation

Algorithm 2 outlines the implementation of the custom `Allreduce` algorithm.
This algorithm addresses the problem for commutative operations when p is not a power of two, by managing a different communication pattern and separating the sending of local inclusive and exclusive data. In contrast to the algorithm defined in the previous section,

this algorithm does not require sending additional information in some rounds. Instead, it maintains the local data exclusive of the initial value separately from the initial value itself. In certain rounds, only the data without the initial value is sent to a nearer process.

To determine for each round whether the data exclusive or inclusive of the initial value must be sent, $p - 1$ is represented in binary form. Starting from the most significant bit, each bit corresponds to a round. If the bit is 1, the data must be sent inclusive of the initial value. The skipping step, which determines the communication partner in each round, is always equal to the number of blocks being sent, ensuring that no process receives duplicate data across all rounds. The skipping steps s_k can be computed more practically by iteratively halving p and applying the ceiling function, which is used in Algorithm 2. Each round's behavior can then be determined by inspecting the next skipping step s_{k+1}: if it is odd, the initial value is not sent in that round; if it is even, the initial value is included in the data sent. A special case occurs in the first round, where the initial value is always sent.

The buffers for merging are managed in the same way as in Algorithm 1, using pointer swaps between two arrays, `W` and `merged`, to avoid unnecessary copying, and an additional array, `recvblocks`, for receiving the data in each round. There is also an extra array, `V`, which permanently stores the initial value. This is necessary because `W` only holds the data exclusive of the initial value. When data including the initial value must be sent, `W` and `V` are first merged into the `merged` array before transmission. Further, the received data is always merged with `W` (or simply copied during round 0). At the end of each iteration, the pointers of `merged` and `W` are swapped. As in Algorithm 1, there is no final copy into the `recvbuf` allocated by the calling process, because the correct starting buffer `W` or `merged` can be determined in advance by checking whether the number of rounds is even or odd.

Algorithm 2 AllGatherMergeCirculant executed by rank P out of p. This algorithm solves the problem by using a different communication pattern and selecting when to send the initial data.

```
procedure AllGatherMergeCirculant($V[m]$, $P$, $p$)          ▷ $V$ – sorted local block
    $q \leftarrow \lceil \log_2 p \rceil$                                ▷ number of rounds
    $s_q \leftarrow p$                                         ▷ roughly–halving skips
    for $k \leftarrow q-1$ downto $0$ do
        $s_k \leftarrow \lceil s_{k+1}/2 \rceil$
    end for
    $recvblocks \leftarrow [m * p]$
    $merged \leftarrow [m * p]$
    $W \leftarrow [m * p]$
    for $k \leftarrow 0$ to $q-1$ do                            ▷ circulant rounds
        $s \leftarrow s_k$; $s' \leftarrow s_{k+1}$; $\varepsilon \leftarrow s' \bmod 2$
        $t \leftarrow (P - s + \varepsilon + p) \bmod p$                       ▷ send to
        $f \leftarrow (P + s - \varepsilon) \bmod p$                         ▷ recv from
        if $\varepsilon = 1$ then                                  ▷ $s'$ odd
            Send($W, t$) || Recv($recvblocks, f$)
            $merged \leftarrow$ Merge($W, recvblocks$)
        else
            if $k = 0$ then                                  ▷ first round
                Send($V, t$) || Recv($recvblocks, f$)
                copy $recvblocks$ into $merged$
            else
                $merged \leftarrow$ Merge($W, V$)
                Send($merged, t$) || Recv($recvblocks, f$)
                $merged \leftarrow$ Merge($W, recvblocks$)
            end if
        end if
        swap($W$, $merged$)                          ▷ pointer swap — no copy
    end for
    $merged \leftarrow$ Merge($W, V$)          ▷ $merged$ holds the final sorted global list
end procedure
```

4.1 Time and Space Complexities

The algorithm closely follows the Bruck's algorithm and Algorithm 1, maintaining the same asymptotic complexities. Exactly $\lceil \log p \rceil$ rounds are required. The computation of doubling steps for communication partners is performed once in $\mathcal{O}(\log p)$. In each round, either the entire data or the data excluding the initial block is sent using again a single `MPI_Sendrecv` call, resulting in equal or lower data transmission compared to the standard Bruck's algorithm. Each round (except round 0 - only one copy) involves one or two merging operations: when sending the entire data, the initial block `V` is merged into `W` before sending, and in every round, the received data is merged into `W`. Thus, the total merge cost remains $\mathcal{O}(m \cdot p)$, identical to Algorithm 1. The rest of the loop is done the same way as in Algorithm 1 , avoiding copies by swapping pointers between two buffers. Consequently, the overall runtime complexity is the same:

$$\mathcal{O}(\lceil \log p \rceil \cdot \alpha + (p-1) \cdot m \cdot \beta + m \cdot p)$$

Similar to Algorithm 1, three buffers are preallocated, each of size at most $m{\cdot}p$, resulting

in the same space complexity of $\mathcal{O}(m \cdot p)$.
Although the asymptotic complexity matches, it has to be noted that Algorithm 1 is guaranteed to require in the worst case roughly 1.5 times more data transfer compared to this algorithm.

References

[1] Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. Edgar Gabriel et al, September 2004. URL: https://www.open-mpi.org/

[2] Efficient algorithms for all-to-all communications in multiport message-passing systems. J. Bruck et al, November 1997. DOI: 10.1109/71.642949