HPC Project Report 2025

Eduardo Silva, 12444280
Technische Universität Wien

Felix Gross, 12019865
Technische Universität Wien

***Abstract*—We implement and compare three MPI Allgather-Merge variants: (1) a baseline that calls MPI_Allgather then does a local p-way merge; (2) a Bruck-based merge; (3) a Circulant merge. We measure and compare their performance on the Hydra compute cluster (up to 640 processes). Both Bruck and Circulant outperform the sequential baseline but not significantly.**

***Index Terms*—MPI Allgather, loser-tree merge, Bruck algorithm, Circulant topology, collective communication, high-performance computing**

I. Introduction

The structure of this report is as such. First, we will define the testing environment used to benchmark all algorithms.

Secondly, the sequential baseline and the two parallel algorithms will be discussed on their own. This entails an explanation of how they work, any improvements made to them, pseudocode of their final implementation as well as an analysis of their runtime and memory complexity, followed by an intermediate result analysis within the algorithm approach.

Following this, the three algorithms will be benchmarked against each other and differences in performance will be discussed. Finally, a short summary will be provided.

II. Test Environment

In our experiments, all benchmarks were conducted on the Hydra cluster, a system comprising 36 nodes, each running up to 32 MPI processes (totalling 640 ranks).

For each MPI configuration $N \times n$ (single node process configurations $p = 1 \times 1, p = 1 \times 10, p = 1 \times 32$. Multi-node configurations $p = 10 \times 1, p = 20 \times 1, p = 10 \times 10, p = 20 \times 10,$ and $p = 10 \times 32, p = 20 \times 32$) we measured execution time over six message sizes: 1, 10, 100, 1 000, 10 000 and 100 000 elements with 3 different input types (Type 0, Type 1, Type 2). During development (including intermediate results), we collected the average of ten independent runs. For the final reported results, we increased to thirty independent runs per configuration ensuring a more robust analysis.

III. Baseline

As an initial reference point, we used a p-way merge using C++'s Standard Template Library (STL) heap utilities [1]. This straightforward approach serves as the baseline for performance and correctness, against which subsequent improvements.

While simple and portable, this implementation incurs significant overhead due to frequent heap adjustments. A heap uses approximately $2 \cdot \log(p)$ comparisons in each step because it handles the tree from the root down to the bottom and needs to compare both children of each node. Meanwhile, a tournament tree only needs $\log(p)$ comparisons because it starts on the bottom of the tree and works up to the root, only making a single comparison in each layer.

A. Tournament Loser Tree

We represent the p sorted input runs as the leaves of a binary loser tree stored in an array of size $O(2p)$. During initialization, we play a tournament among the first element of each run. storing the smaller (loser) key at each internal node and keeping the overall minimum in a dedicated winner slot. At each merge step, we output the current winner, then refill its leaf from the same run and replay that element up the tree: at each internal node it competes with the stored loser (the smaller key is promoted upward, the larger is written back), until the new overall winner is found at the root. This yields an $O(m \log p)$ merge with only $\log p$ comparisons per output element [2]. Including the preceding MPI_Allgather this results in a final runtime complexity of $O(m \log p + m + \log p)$.

Furthermore, we applied a Structure of Arrays (SoA) memory layout, keeping *runID* and *value* arrays in separate buffers, instead of grouping them together in an Array of Structs (AoS) format (Section IV.C elaborates this topic more in depth). This layout improves cache efficiency and enables more optimized memory access patterns.

To avoid memory copies we use an an additional buffer of size $O(m)$ to hold the gathered data from MPI_Allgather. From there the p data blocks are merged directly into the receive buffer. Due to the nature of the loser tree, only up to $\log p$ of the $2p$ nodes will be updated within one tournament run. This leads to an additional memory requirement of $O(m + 2p)$ and $O(m \log p)$ memory copies.

B. Intermediate Results

Fig. 1 compares the merge time (μs) of the two p-way merging previously mentioned on the Hydra cluster. These measurements validate our expectation that Tournament Loser Tree outperforms a simple Merge Heap in every regime. The Loser Tree runs about 3.52 times faster than the basic heap-based merge across these six different input sizes, we therefore adopt it as our definitive "Baseline".

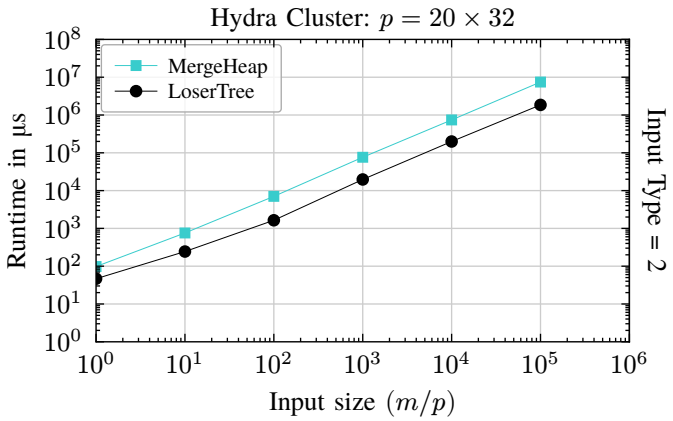


Fig. 1: Performance comparison between MergeHeap and Tournament loser Tree for input type of 2

IV. Algorithm 1: Bruck

The Bruck's algorithm (Fig. 2) completes an all-to-all data exchange in exactly $\lceil \log_2 p \rceil$ rounds. In each round, every process exchanges increasingly more blocks of data with a partner whose rank differs by $2^k \pmod p$. After the send/receive, each process merges the newly acquired blocks into its local buffer–doubling the total amount of blocks held.

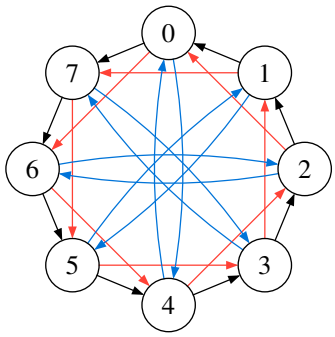
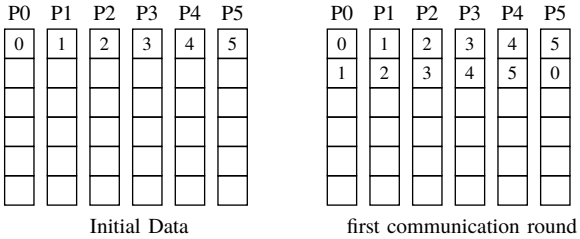


Fig. 2: Example bruck communication rounds for $p = 8$. Black, red, blue arrows represent respectively communication round 1, 2, 3.

However, this algorithm presents an issue when the number of processes is not a power of two. Each round we want to send and receive 2^k data blocks where k is our current round ($0 \leq k < \lceil \log_2 p \rceil$). So when having a non-power-of-two number of processes our last communication round would overflow the data buffer with duplicates. Fig. 3 illustrates this problem clearly for $p = 6$.



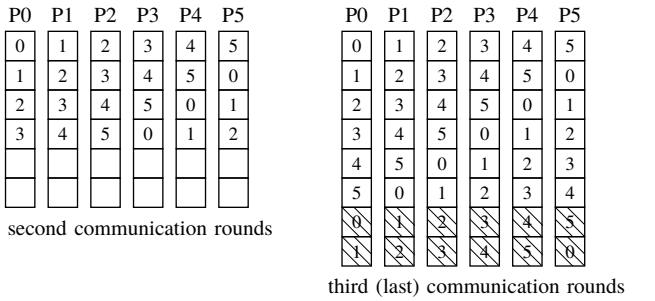


Fig. 3: Bruck allgather for a non-power-of-two number of processes leading to a buffer overflow with duplicates

Therefore, we realized that our last communication round would only need $p - 2^{\lfloor \log_2 p \rfloor}$ blocks sent and received for p not being a power of two.

To fix this issue, we modify the algorithm to precisely calculate the number of blocks needed for the final communication round, as:

$$\text{send\_blocks} = \min\left(2^k, p - 2^{\lfloor \log_2 p \rfloor}\right) \tag{1}$$

Additionally, we ensure that only the missing blocks are sent. To accomplish this precisely we introduce an additional metadata field called *source* rank, which identifies the originating process of each element. We now want to pack exactly those missing blocks whose source rank lies in $[r, r + \text{send\_blocks} - 1] \bmod p$.

A. Source tagging for filtering

To accurately select and send the correct elements in the final partial round, each data element (*value*) is accompanied by a small metadata field (*source*) indicating its original MPI rank. For instance, given our testing environment supports up to 640 ranks ($N \times n, 20 \times 32$), a 16-bit unsigned integer (`uint16_t`) comfortably encodes this information.

In each communication round, processes exchange not only their data but also the corresponding origin rank. This additional data enables processes in the final communication round to filter only those elements whose origins lie in the missing interval, completely avoiding duplicate sends.

B. Communication overhead

While source-rank tagging solves the duplicate transmission problem, it introduces an additional overhead. Each 32 bits (`tutype_t`) is paired with a 16-bit source rank (`rank_t`), effectively increasing the payload by 50%. Therefore, the total cost per message communication becomes:

$$\alpha + \beta\left(m + \frac{m}{2}\right) = \alpha + 1.5\beta m \tag{2}$$

C. Optimizations and their impact

To maximize cache efficiency, memory alignment, and better compiler auto-vectorization, we store *values* and origin tags in two separate, contiguous arrays (Fig. 4, similar to baseline) rather than in a single array of structs (Fig. 5). This reduces unnecessary memory traffic.

$S_v[0]$	…	$S_v[m-1]$

$S_s[0]$	…	$S_s[m-1]$

Fig. 4: Our efficient SoA approach illustrating the overhead tagging. $S_v[i], S_s[i]$ denotes respectively the buffer holding (32-bit) *values* and (16-bit) *source*.

32 bits	16 bits	…	32 bits	16 bits
$S[0].v$	$S[0].s$	…	$S[m-1].v$	$S[m-1].s$

Fig. 5: A possible, but inefficient AoS scheme illustrating the overhead tagging. $S[i]$ denotes the struct holding *source* and *value*.

Furthermore, instead of copying the merged output back into the input array each round, we maintained two full-size buffers (C, and N) and simply swap their roles after every merge [3]. In round k, we merge into N and then swap the pointers in $O(1)$ time.

This approach eliminates an extra $O(m)$ copy per round, cutting the total memory traffic. It also keeps each buffer cache-friendly, since we never move data within the array. Algorithm 1 describes the whole process exactly.

Our Bruck implementation is first initialized by copying its original data block into its current work-buffer and initializing the corresponding tags. It then sends $O\left(1.5 \cdot 2^k \cdot \frac{m}{p}\right)$ and merges $O\left(1.5 \cdot 2^{k+1} \cdot \frac{m}{p}\right)$ integer elements once each round. In the last round for $p \notin \{2^i \mid i \in \mathbb{N}\}$ an additional $O\left(2 \cdot \frac{m}{2}\right)$ filtering operation has to done aswell which moves $O\left(1.5 \cdot \frac{m}{2}\right) = O\left(\frac{3}{4}m\right)$ integer elements. One final copy of the fully merged data into the output buffer is done at the end. This results in the following runtime complexity:

$$\begin{aligned} T_{Bruck} &= O\left(1.5\frac{m}{p} + \sum_{k=0}^{\lceil \log p \rceil - 1} \left(\alpha + 1.5\beta \cdot 2^k \cdot \frac{m}{p} + 2^{k+1} \cdot \frac{m}{p}\right) + 2\frac{m}{2} + m\right) \\ &= O\left(1.5\frac{m}{p} + (\lceil \log p \rceil - 1)\alpha + 1.5\beta m + 4m\right) \end{aligned} \tag{3}$$

$$\begin{aligned} C_{Bruck} &= O\left(1.5\frac{m}{p} + \sum_{k=0}^{\lceil \log p \rceil - 1} \left(2^{k+1} \cdot \frac{m}{p}\right) + \frac{3}{4}m + m\right) \\ &= O\left(1.5\frac{m}{p} + 2m + \frac{7}{4}m\right) = O\left(1.5\frac{m}{p} + 3.75m\right) \end{aligned}$$

Furthermore it entails $O(3 \cdot 1.5m) = O(4.5m)$ extra memory and $O\left(5.25m + 1.5\frac{m}{p}\right)$ copied memory excluding communication.

D. Intermediate Results

When we extended the given Bruck implementation (which only worked when p is a power of two) to handle arbitrary p, performance degraded significantly due to the extra communication previously mentioned. Although, further optimizations, such as, reorganizing into a SoA layout, using non-blocking Isend/Irecv with a ping-pong buffer reduced noticeably execution time (see Fig. 6) but unfortunately they never outperform the original power-of-two code.

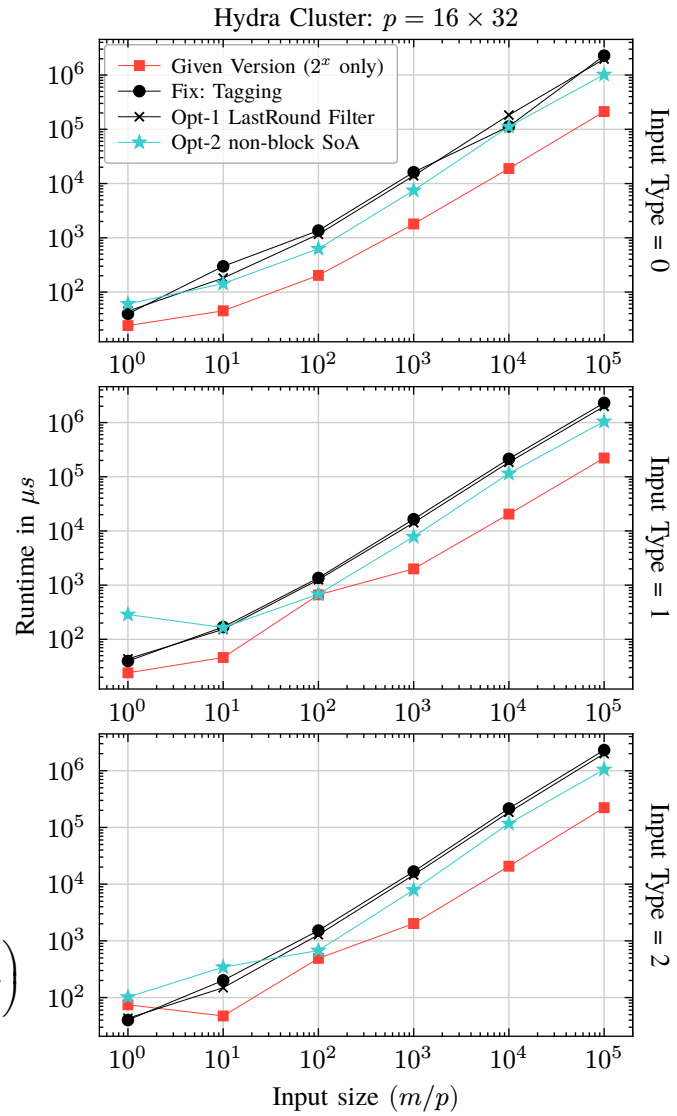


Fig. 6: Runtime comparison between the provided Bruck (red) algorithm and further optimizations for a versioned Bruck that also works for non-power-of-two

Algorithm 1: Generalized Bruck Allgater-Merge for processor $r \in C_p^s$, on the ciruclant graph C_p^s. The block that processor r has initially is $V[s_c]$. C and N are ping-pong buffers. In the final round (when p is not a perfect power of two), it applies a "last-round filter" to pack only necessary elements (PACKPARTIAL). Elements whose tags lies in $\left[r, r + \min\left(2^k, p - 2^{\lfloor \log_2 p \rfloor} - 1\right)\right]$. Additionally, MERGE_RUNS performs a classic two-way merge of two sorted "runs" taking into account respective *source* values. At the end, the result is copied to W.

1: **procedure** ALLGATHER$\left(V[s_c],\, W,\, r \in C_p^s\right)$
2: ▷ Allocate arrays of length m, where rank_t ⇔ uint16_t
3: $m \leftarrow p \cdot s_c$
4: $C_v[m], N_v[m], R_v[m] \leftarrow$ MALLOC($m \cdot$ **sizeof**(tuwtype_t))
5: $C_s[m], N_s[m], R_s[m] \leftarrow$ MALLOC($m \cdot$ **sizeof**(rank_t))
6:
7: ▷ Initialize local block
8: **for** $i = 0, ..s_c - 1$ **do**
9: $C_v[i] \leftarrow V[i]$ ▷ $V[i]$ is sendbuf, where we get our input from
10: $C_s[i] \leftarrow r$
11: **end**
12:
13: *reqs* ← MPI_Request[4]
14: $h_b \leftarrow 1$
15: $s_k \leftarrow 1$
16: $q \leftarrow \lceil \log_2 p \rceil$
17: **for** $k = 0, ..., q - 1 \wedge s_k < p$ **do**
18: ▷ how many blocks this round?
19: **if** $2 \cdot s_k \leq p$ **then**
20: $s_b \leftarrow s_k$
21: **else**
22: $s_b \leftarrow p - s_k$ ▷ partial final round
23: **end**
24: $s_e \leftarrow s_b \cdot s_c$ ▷ total elements to send
25: $t, f \leftarrow (r - s_k + p) \bmod p,\ (r + s_k) \bmod p$
26:
27: MPI_IRECV(R_v, s_e, `MPI_INT`, f, 0,.., *reqs*[0])
28: MPI_IRECV(R_s, s_e, `MPI_UNSIGNED_SHORT`, f, 1,.., *reqs*[1])
29: **if** $s_b < s_k$ **then**
30: ▷ filter to pack only necessary elements
31: PACKPARTIAL($s_b, C_v, C_s, N_v, N_s, r, p, s_c, h_b$)
32: MPI_ISEND(N_v, s_e, `MPI_INT`, t, 0,.., *reqs*[2])
33: MPI_ISEND(N_s, s_e, `MPI_UNSIGNED_SHORT`,..,t, 1,..,*reqs*[3])
34: **else**
35: MPI_ISEND(C_v, s_e, `MPI_INT`, t, 0,.., *reqs*[2])
36: MPI_ISEND(C_s, s_e, `MPI_UNSIGNED_SHORT`,..,t, 1,..,*reqs*[3])
37: **end**
38:
39: MPI_WAITALL(4, *reqs*, `MPI_STATUS_IGNORE`)
40:
41: ▷ Merge two sorted runs A and B into a single sorted output O
42: MERGE_RUNS(C_v, C_s, $h_b \cdot s_c$, R_v, R_s, s_e, N_v, N_s)
43:
44: ▷ Ping Pong
45: SWAP(C_v, N_v)
46: SWAP(C_s, N_s)
47: $h_b \leftarrow h_b + s_b$
48: $s_k \leftarrow 2 \cdot s_k$ ▷ `<<= 1`
49: **end**
50:
51: ▷ Copy the fully-merged result into R
52: **for** $i = 0, ..s_c - 1$ **do**
53: $W[i] \leftarrow C_v[i]$ ▷ $W[m]$ is the recvbuf (where we output our result)
54: **end**
55: FREE($C_v, C_s, N_v, N_s, R_v, R_s$)
56: **end**

V. Algorithm 2: Circulant

This section will explain the rough idea behind Circulant, how er implemented it, our most important improvements and two failed attempts. Their a comparison of their performance is then shown in Fig. 9 and Table II.

A. The idea behind it

The Circulant algorithm provides an alternative solution to the issue of Bruck only working for powers of two processes.

Just like it, Circulant completes an all-reduce data exchange in exactly $\lceil \log_2 p \rceil$ rounds while merging increasingly larger blocks of data in-between communication rounds.

However, unlike Bruck, instead of using doubling skips to decide what process to send data to, Circulant uses halving skips. This combined with it selectively not sending its own initial data block allows it to work for any number of processes (Fig. 7).

However, this algorithm presents an issue when the number of processes is not a power of two. Each round we want to send and receive 2^k data blocks where k is our current round ($0 \leq k < \lceil \log_2 p \rceil$). So when having a non-power-of-two number of processes our last communication round would overflow the data buffer with duplicates. Fig. 3 illustrates this problem clearly for $p = 6$.

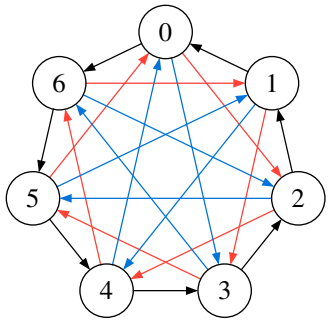


Fig. 7: Example Circulant communication rounds for $p = 7$. Black, red, blue arrows represent respectively communication round 1, 2 and 3.

B. Initial implementation using MPI_Sendrecv

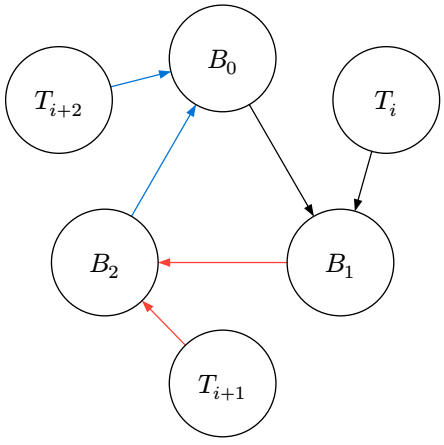


Fig. 8: Depiction of three rotating buffers B_0, B_1, B_2 and a receive buffer T_i at rounds i (black), $i+1$ (red) and $i+2$ (blue). Edges indicate in which array the data in a given buffer is merged into.

Our initial implementation closely follows the pseudocode given in the task description. Sending and receiving data was done using MPI_Sendrecv. To reduce the number of memory copies three rotating buffers of size $O(m)$ were used to merge the received data of a given round i with the most up to date data block held in buffer $i \bmod 3$ into the respective next buffer $i+1 \bmod 3$. This process is depicted in Fig. 8.

In case a processors initial data has to be sent aswell, it will be merged into the buffer at position $i+1 \bmod 3$ and the one at position $i+2 \bmod 3$ will be used to merge the held and received data instead.

C. Switching to non-blocking communication

For the first improvement we switched to using non-blocking communication using MPI_Isend and MPI_Irecv. Due to the three rotating buffers it allowed to postpone waiting for a send operation to finish up until a full rotation was made and a buffer has to be reused.

Its pseudocode is depicted in Algorithm 3.

D. Pipelining communication and merging

When receiving data MPI_Irecv forbids access to its passed buffer until all the data has arrived. This however leads to idle time where a process has to wait, even though some of the data has already arrived and can be merged. To solve this problem we broke each communication in 2^k chunks each of size $\frac{m}{p}$. Now each process only has to wait for the first chunk to arrive before they can start merging the data or if a given chunk is delayed. Incidentally this also means that each process only has to merge $\frac{m}{p}$ elements before it can send a chunk to the next process. The pseudocode of this version is depicted in Algorithm 4.

E. Adding simple merge checks

To accelerate merges we added a check if a given merge for two arrays A and B can be skipped. For that we simply compare the last array of A with the first array of B and vice versa. If one of the two conditions is satisfied it means one array is strictly smaller than the other and they can be copied without any further comparisons. This did lead to further speedup and paired well with the pipelined version since these checks could be made for every chunk (with some minor adjustments).

F. Using a graph communicator

We briefly attempted to gain more performance by optimizing the pattern in which processes communicate with each other. This was done by changing the used communicator object to one made with MPI_Dist_Graph_create_adjacent which was provided with all upcoming communications and their size.

This however did not lead to any speed gains. It seems like the overhead of matching the communication graph to the network topology is not worth any improvements in communication time.

G. Using AVX512 registers to merge

Since Hydra supports AVX512 registers which can do 16 vectorized integer operations at once we tried to exploit to

more efficiently two-way-merge. The idea was to merge two arrays A and B into an array C as such. In each iteration, first the next 16 elements of A are loaded into a vector register and copied to C as is. The elements in the vector register are then all compared with the next element in B to see how many elements were correctly moved. The position pointer for A is then moved by the amount of elements that were correct. If none were correct, then the next element in B is copied instead and the process is repeated. The pseudocode is shown in Algorithm 2.

Algorithm 2: Two-way-merge using AVX registers. Arrays A and B are merged into array C

```
1:  procedure TwoWayMergeAVX(A[n_A], B[n_B], C[n_C])
2:      ▷ Track the position in each array
3:      i_A, i_B, i_C ← 0
4:      while Neither A nor B is exhausted do
5:          C[i_C, ..., i_C + 15] ← A[i_A, ..., i_A + 15]
6:          V_0 ← load(A[i_A, ..., i_A + 15])
7:          V_1 ← V_0 ≤ B[i_B]
8:          n_1 ← count_ones(V_1))
9:          if n_1 = 0 then
10:             C[i_C] ← B[i_B]
11:             i_C ← i_C + 1
12:             i_B ← i_B + 1
13:         else
14:             i_C ← i_C + n_1
15:             i_A ← i_A + n_1
16:         end
17:     end
18:     if A is exhausted then
19:         ▷ Copy the remaining elements in B
20:         Memcpy(C, B)
21:     else
22:         ▷ Copy the remaining elements in A
23:         Memcpy(C, A)
24:     end
25:     return C
26: end
```

H. Further subdividing

Due to the relative success of the pipelined version and because profiling showed that around $\frac{1}{3}$rd of all process time is spent in MPI_Wait it was attempted to further subdivide the chunks of the pipelined version. This turned out only be detrimental, however it suggests that it might be possible to gain more performance using less chunks per message. Due to time constraints we were not able to test this hypothesis. Our evaluation for different subdivisions per chunk can be seen in Table I.

TABLE I: Results of subdivision experiments. Each column shows the runtime when chunks were further split into that many messages

Further subdividing, Hydra p=20x32, m/p=100000							
Type	1	2	5	10	20	50	100
0	1.09*1e6	1.09*1e6	1.08*1e6	1.26*1e6	1.33*1e6	1.36*1e6	1.46*1e6
1	1.15*1e6	1.16*1e6	1.18*1e6	1.31*1e6	1.34*1e6	1.42*1e6	1.54*1e6
2	1.15*1e6	1.16*1e6	1.17*1e6	1.31*1e6	1.34*1e6	1.43*1e6	1.55*1e6

I. Final choice of Circulant implementation

For large message sizes $\frac{m}{p}$ the best version of Circulant was the pipelined one with added checks to potentially skip merges. However for smaller message sizes, it was much worse than the first improvement which only used one MPI_Isend and MPI_Irecv for each communication. To get the best of both worlds, we decided to make one final version which chooses which of the two to use depending on the message size. If $\frac{m}{p} \geq 10000$ the more involved pipelined version is used and the simper one otherwise.

The worst case scenario for Circulant is when it has to merge its own data each round. In that case the simple variant sends one message of size $O\Big(\big(2^k+1\big)\cdot\frac{m}{p}\Big)$, and performs merges of size $O\Big(\big(2^k+1\big)\cdot\frac{m}{p}\Big)$ and $O\Big(2^{k+1}\cdot\frac{m}{p}\Big)$ each round. It uses an extra memory of $O(3.5m)$ for all buffers.

The runtime and copy complexity of the simple variant using is therefore:

$$\begin{aligned} T_{simple} &= O\left(\sum_{k=0}^{\lceil \log p\rceil -1}\left(\frac{3}{2}\alpha+\big(2^k+1\big)\cdot\frac{m}{p}\cdot\beta+\big(2^{k+1}+2^k+1\big)\cdot\frac{m}{p}\right)+m\right) \\ &= O\left(\frac{3}{2}(\lceil\log p\rceil-1)\alpha+(p+\lceil\log p\rceil-1)\cdot\frac{m}{p}\cdot\beta+(3p+\lceil\log p\rceil-1)\frac{m}{p}+m\right) \\ &= O\left(\frac{3}{2}(\lceil\log p\rceil-1)\alpha+(p+\lceil\log p\rceil-1)\cdot\frac{m}{p}\cdot\beta+(\lceil\log p\rceil-1)\frac{m}{p}+4m\right) \end{aligned} \tag{4}$$

$$\begin{aligned} C_{simple} &= O\Big(\textstyle\sum_{k=0}^{\lceil\log p\rceil-1}\Big(\big(2^{k+1}+2^k+1\big)\cdot\frac{m}{p}\Big)+m\Big) \\ &= O\Big(\big(3p+\lceil\log p\rceil-1\big)\frac{m}{p}+m\Big) \\ &= O\Big(\big(\lceil\log p\rceil-1\big)\frac{m}{p}+4m\Big) \end{aligned}$$

The complex variant sends $O\big(2^k+1\big)$ message of size $O\Big(\frac{m}{p}\Big)$, and performs merges of size $O\Big(\big(2^k+1\big)\cdot\frac{m}{p}\Big)$ and $O\Big(2^{k+1}\cdot\frac{m}{p}\Big)$ each round. It also uses an extra memory of $O(3.5m)$ for all buffers and an additional $O\big(4\cdot\frac{m}{2}\big)$ memory to store all the send and receive requests for a total of $O(5.5m)$. While the amount of copies are the same its runtime complexity is much more complicated, since the merging of data is interleaved with sending messages.

$$\begin{aligned} T_{Pipelined} = O(&\sum_{k=0}^{\lceil\log p\rceil-1}\left(\frac{m}{p}+\frac{\alpha}{2}+\max\left(2^k\cdot\left(\frac{m}{p}+\frac{\alpha}{2}\right),\frac{m}{p}\cdot\beta\right)\right. \\ &\left.+\frac{\alpha}{2}+2^k\cdot\max\left(\frac{\alpha}{2},2\frac{m}{p}\right)+m\right) \end{aligned} \tag{5}$$

$$\begin{aligned} C_{Pipelined} &= O\Big(\textstyle\sum_{k=0}^{\lceil\log p\rceil-1}\Big(\big(2^{k+1}+2^k+1\big)\cdot\frac{m}{p}\Big)+m\Big) \\ &= O\Big(\big(\lceil\log p\rceil-1\big)\frac{m}{p}+4m\Big) \end{aligned}$$

The final implementations complexities are then the ones of whichever version is picked for a specific message size.

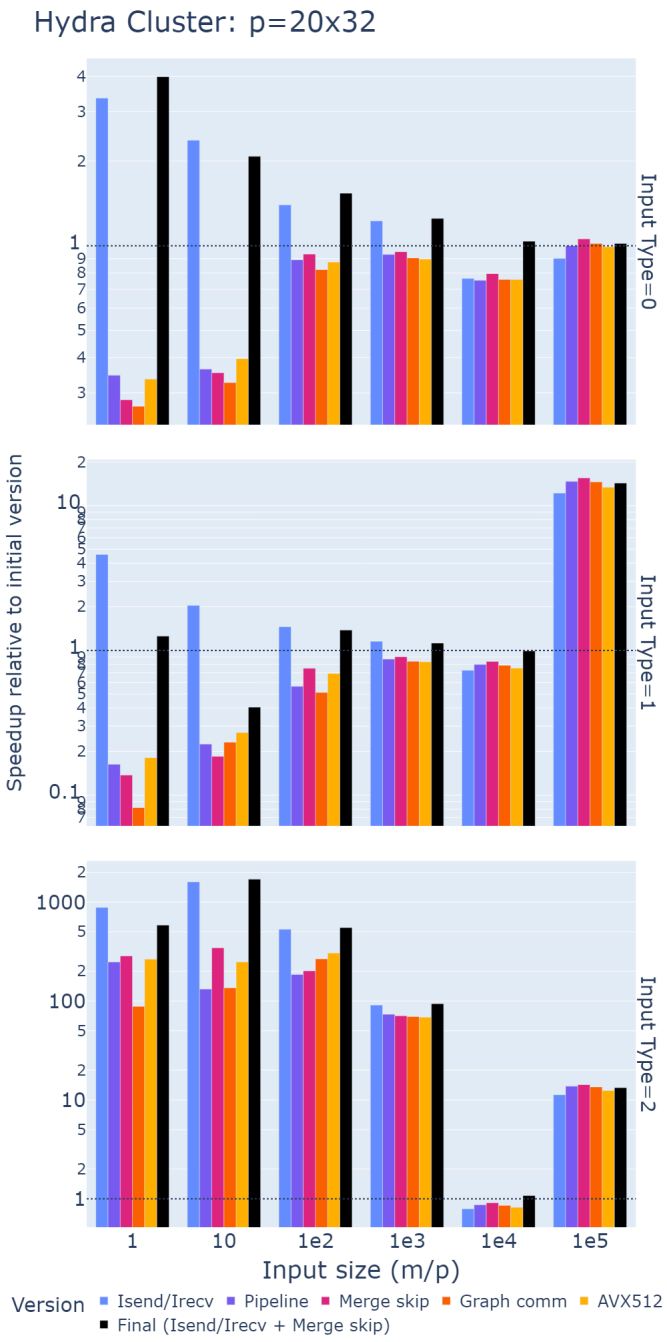


Fig. 9: Comparison of the speedup of different Circulant implementations relative to the initial version using MPI_Sendrecv.

20x32 hydra benchmark							
Impl.	Type	1	10	100	1000	10000	100000
Initial	0	1.62e+02	2.68e+02	9.62e+02	9.50e+03	9.94e+04	1.10e+06
	1	1.83e+02	2.39e+02	1.06e+03	9.66e+03	1.04e+05	1.66e+07
	2	1.95e+05	1.89e+05	3.97e+05	8.00e+05	1.13e+05	1.54e+07
	0	4.84e+01	1.13e+02	6.89e+02	7.75e+03	1.30e+05	1.22e+06
	1	3.98e+01	1.17e+02	7.28e+02	8.37e+03	1.43e+05	1.36e+06
Isend/Irecv	2	2.21e+02	1.18e+02	7.47e+02	8.79e+03	1.43e+05	1.37e+06
Pipeline	0	4.68e+02	7.36e+02	1.08e+03	1.02e+04	1.32e+05	1.10e+06
	1	1.12e+03	1.06e+03	1.88e+03	1.11e+04	1.30e+05	1.13e+06
	2	7.86e+02	1.43e+03	2.14e+03	1.09e+04	1.30e+05	1.12e+06
Merge skip	0	5.72e+02	7.58e+02	1.03e+03	9.98e+03	1.25e+05	1.04e+06
	1	1.33e+03	1.29e+03	1.41e+03	1.07e+04	1.24e+05	1.07e+06
	2	6.83e+02	5.49e+02	1.97e+03	1.13e+04	1.24e+05	1.08e+06
Graph comm	0	6.03e+02	8.22e+02	1.17e+03	1.05e+04	1.31e+05	1.08e+06
	1	2.23e+03	1.03e+03	2.07e+03	1.15e+04	1.32e+05	1.14e+06
	2	2.21e+03	1.39e+03	1.49e+03	1.15e+04	1.32e+05	1.14e+06
AVX512	0	4.83e+02	6.76e+02	1.10e+03	1.06e+04	1.31e+05	1.11e+06
	1	1.01e+03	8.84e+02	1.53e+03	1.16e+04	1.38e+05	1.24e+06
	2	7.37e+02	7.61e+02	1.30e+03	1.17e+04	1.38e+05	1.24e+06

Algorithm 3: Simple Circulant implementation using MPI_Isend/Irecv for processor $r \in C_p^s$ with adjusted roughly halving skips $s_k, k = 0, 1, ..., q-1$ and $s_q = p$ and $s_k = \left\lfloor \frac{s_{\{k+1\}}}{2} \right\rfloor$, where $q = \lfloor \log_2 p \rfloor$

```
1:  procedure AllGatherCirculantSimple(V[s_c], W, r ∈ C_p^s)
2:    ▷ Early return for single process
3:    if p = 1 then
4:      W ← V
5:      return
6:    end
7:    m ← p · s_c
8:    ▷ Allocate 3 buffers of length m to store merged data
9:    B_0[m], B_1[m], B_2[m] ← Malloc(m · sizeof(tuwtype_t))
10:
11:   ▷ Tracks which buffer B_i contains the most up to date data
12:   i_B ← 0
13:   ▷ Tracks the amount of data in the most up to date buffer B_i
14:   n_B ← s_c
15:
16:   ▷ Allocate buffer of length ⌊p/2⌋ · s_c to store incoming data
17:   T[⌊p/2⌋ · s_c] ← Malloc(⌊p/2⌋ · s_c · sizeof(tuwtype_t))
18:
19:   ▷ Tracks if the data in a given buffer B_i has been sent
20:   send_reqs ← MPI_Request[3]
21:   ▷ Tracks if the incoming data is fully received
22:   recv_req ← MPI_Request
23:
24:   ▷ Round 0
25:   ε ← s_1 ∧ 0x1
26:   t, f ← (r − s_0 + ε + p) mod p, (r + s_0 − ε) mod p
27:   MPI_Irecv(B_{i_B}, s_c, MPI_INT, f, 0,.., recv_req)
28:   MPI_Isend(V, s_c, MPI_INT, t, 0,.., IGNORE_REQUEST)
29:   MPI_Wait(recv_req, MPI_STATUS_IGNORE)
30:
31:   ▷ The remaining rounds
32:   for k = 1, ..., q − 1 ∧ s_k < p do
33:     ε ← s_{k+1} ∧ 0x1
34:     t, f ← (r − s_k + ε + p) mod p, (r + s_k − ε) mod p
35:     if ε = 1 then
36:       MPI_Irecv(T, n_B, MPI_INT, f, k,..,recv_req)
37:       MPI_Isend(B_{i_B}, n_B, MPI_INT, t, k,..,send_Req[i_B])
38:
39:       MPI_Wait(recv_req, MPI_STATUS_IGNORE)
40:       MPI_Wait(send_req[(i_B + 1) mod 3], MPI_STATUS_IGNORE)
41:       twoWayMerge(B_{i_B}, n_B, T, n_B, B_{(i_B+1) mod 3})
42:       i_B ← (i_B + 1) mod 3
43:       n_B ← 2 · n_B
44:     else
45:       ▷ Post recv for T early
46:       MPI_Irecv(T, n_B + s_c, MPI_INT, f, k,..,recv_req)
47:
48:       ▷ Merge V into the next buffer
49:       MPI_Wait(send_req[(i_B + 1) mod 3], MPI_STATUS_IGNORE)
50:       twoWayMerge(V, s_c, B_{i_B}, n_B, B_{(i_B+1) mod 3})
51:       MPI_Isend(B_{{(i_B+1) mod 3}}, n_B + s_c, MPI_INT, t, k, comm, send_req[(i_B + 1) mod 3])
52:
53:       ▷ Merge the incoming data T into the buffer two ahead
54:       MPI_Wait(send_req[i_B + 2] mod 3, MPI_STATUS_IGNORE)
55:       MPI_Wait(recv_req, MPI_STATUS_IGNORE)
56:       twoWayMerge(B_{i_B}, n_B, T, n_B + s_c, B_{(i_B+2) mod 3})
57:
58:       i_B ← (i_B + 2) mod 3
59:       n_B ← 2 · n_B + s_c
60:     end
61:     twoWayMerge(V, s_c, B_{i_B}, n_B, W)
62:
63:     ▷ Cleanup resources
64:     MPI_Waitall(3,send_req, MPI_STATUSES_IGNORE)
65:     free(B_0, B_1, B_2, T)
66:   end
67: end
```

Algorithm 4: Pipelined Circulant implementation for processor $r \in C_p^s$ with adjusted roughly halving skips $s_k, k = 0, 1, ..., q-1$ and $s_q = p$ and $s_k = \left\lfloor \frac{s_{\{k+1\}}}{2} \right\rfloor$, where $q = \lfloor \log_2 p \rfloor$.

```
1:  procedure AllGatherCirculantPipelined(V[s_c], W, r ∈ C_p^s)
2:    ▷ Early return for single process
3:    if p = 1 then
4:      W ← V
5:      return
6:    end
7:    m ← p · s_c
8:    ▷ Allocate 3 buffers of length m to store merged data
9:    B_0[m], B_1[m], B_2[m] ← Malloc(m · sizeof(tuwtype_t))
10:
11:   ▷ Tracks which buffer B_i contains the most up to date data
12:   i_B ← 0
13:   ▷ Tracks the amount of data in the most up to date buffer B_i
14:   n_B ← 0
15:   ▷ Tracks the number of chunks in a buffer B_i
16:   c_0, c_1, c_2 ← 0
17:
18:   ▷ Allocate buffer of length ⌊p/2⌋ · s_c to store incoming data
19:   T[⌊p/2⌋ · s_c] ← Malloc(⌊p/2⌋ · s_c · sizeof(tuwtype_t))
20:
21:   ▷ Tracks if the data in a given buffer B_i has been sent
22:   send_reqs_0, send_reqs_1, send_reqs_2 ← Malloc[floor(p/2) · sizeof(MPI_Request)]
23:   ▷ Tracks if the incoming data is fully received
24:   recv_req ← Malloc[floor(p/2) · sizeof(MPI_Request)]
25:
26:   ▷ Round 0
27:   ε ← s_1 ∧ 0x1
28:   t, f ← (r − s_0 + ε + p) mod p, (r + s_0 − ε ) mod p
29:   MPI_Irecv(B_{i_B}, s_c, MPI_INT, f, 0,.., recv_req[0])
30:   MPI_Isend(V, s_c, MPI_INT, t, 0,.., IGNORE_REQUEST)
31:   MPI_Wait(recv_req[0], MPI_STATUS_IGNORE)
32:   n_B, c_0 ← s_c, 1
33:
34:   ▷ The remaining rounds
35:   for k = 1, ..., q − 1 ∧ s_k < p do
36:     ε ← s_{k+1} ∧ 0x1
37:     t, f ← (r − s_k + ε + p) mod p, (r + s_k − ε) mod p
38:     if ε = 1 then
39:       i_N ← (i_B + 1) mod 3
40:       ▷ Prepare for the arrival of c_B messages and store them in B_{i_N}
41:       chunkwise_MPI_Irecv(T, f, c_{i_B}, recv_req)
42:       ▷ Send the data in B_{i_B} using c_{i_B} messages
43:       chunkwise_MPI_Isend(B_{i_B}, t, c_{i_B}, send_reqs_{i_B})
44:
45:       MPI_Waitall(c_{i_N}, send_reqs_{i_N})
46:       ▷ Merge the incoming chunks as they arrive
47:       chunkwise_recv_and_twoWayMerge(B_{i_B}, n_B, T, c_{i_B}, B_{i_N})
48:       n_B, c_{i_N} ← 2 · n_B, 2 · c_{i_B}
49:       i_B ← i_N
50:     else
51:       chunkwise_MPI_Irecv(T, c_{i_B} + 1, f, recv_reqs)
52:
53:       ▷ The temporary array is stored in buffer B_{i_T}
54:       i_T ← (i_B + 1) mod 3
55:       MPI_Waitall(c_{i_T}, send_reqs_{i_T})
56:       ▷ Send chunks away as soon as they are ready
57:       chunkwise_twoWayMerge_and_send(V, s_c, B_{i_N}, n_B, B_{i_T}, c_{i_B} + 1)
58:       c_{i_T} ← c_{i_B} + 1
59:
60:       i_N ← (i_B + 2) mod 3
61:       MPI_Waitall(c_{i_N}, send_reqs_{i_N})
62:       chunkwise_recv_and_twoWayMerge(B_{i_B}, n_B, T, c_{i_T}, B_{i_N})
63:       n_B, c_{i_N} ← 2 · n_B + s_c, 2 · c_{i_B} + 1
64:       i_B ← i_N
65:     end
66:     twoWayMerge(V, s_c, B_{i_B}, n_B, W)
67:
68:     ▷ Cleanup resources
69:     MPI_Waitall(3 × ⌊p/2⌋,send_reqs_0, send_reqs_1, send_reqs_2)
70:     free(B_0, B_1, B_2, T, send_reqs_0, send_reqs_1, send_reqs_2)
71:   end
72: end
```

VI. Results

In this section, we present and compare the running times of our three Allgather-Merge implementations (Baseline, Bruck, Circulant) for the nine different MPI configurations, six messages sizes, and three input types (Section II). Each of the nine MPI configurations gets its own table (from Table III to Table XI), with one column per message size and one block of three rows per implementation (Type 0, Type 1, Type 2). Moreover, we also compare Bruck's and Circulant's speedup relative to the sequential Baseline (Fig. 10 to Fig. 12) for each configuration and input type, and Fig. 13 visualizes the overall distribution of speedups. Results are averaged over 30 runs.

Across all configurations, we observe that Baseline often meets or outperforms both Bruck and Circulant whenever process message sizes m/p are very small (1 or 10 elements) regardless of process distribution or input type. In those cases, initial communication overhead dominates total time. Furthermore, for Type 0 inputs, where there are long stretches of identical elements, the Baseline's loser-tree barely does any work since the same datablock will keep winning until the values change. Thus, branch prediction is nearly perfect.

Interestingly, the performance of Bruck and Circulant seems to hit a wall for, $p = 10 \times 1$ and $p = 20 \times 1$ once m/p exceeds roughly 10^4.

In Circulant's case, this possibly arises primarily because OpenMPI stops sending messages eagerly, once the payload exceeds 4096 *bytes*; below that threshold small sends complete immediately, but above it each transfer incurs a rendezvous handshake.

Moreover, when each of the 10 or 20 processes finishes its local merge at nearly the same time, all ranks attempt to send simultaneously and must wait together for their data to arrive. Therefore no process can really get ahead, causing a synchronization bottleneck.

For $N > 1$ (multi-node cases), we notice that Bruck tends to fall off earlier than Circulant as m/p increases. This likely stems from Circulant sending data in chunked segments. The pipeline approach used in Circulant seems to prove efficient, as it allows earlier access to buffers, ending up being more helpful as messages grow larger.

Bruck's performance remains nearly constant whether communication is intra or inter node, whereas Circulant shows a slightly better performance on inter-node runs due to the large amount of chunks it sends for larger message sizes. The difference however is rather small and could simply be due to noise. Regardless, both algorithms continue to improve as we increase process count, indicating they have not yet fully saturated any shared resource.

All in all, the speedup of our parallel algorithms is relatively modest. It rarely exceeds 3x compared to Baseline. The combined effects of communication-handshake overhead, eager threshold transitions, network saturation seem to keep observed speedups mostly around 1.5x-2x. Given that in largest configuration there are 640 processes collaborating, the speedup does not seem to be worth the additional compute resources and network infrastructure. However this is likely to change as the amount of data to merge increases.

VII. Conclusion

In this project we explored multiple implementations for three Allreduce algorithms. For the sequential Baseline, Bruck and Circulant, we experimented with different approaches before settling on their final versions. The final Baseline uses a sequential p-way loser-tree merge. For Bruck, we addressed the issue of it failing if the number of processes is not a power of two by tagging each element with their source rank and filtering them in the last round. To get the most out of Circulant we used two versions which are dispatched depending on the amount of data: a simple Isend/Irecv-based implementation for small message sizes, and a more complex pipelined variant for larger messages.

Performance results show that both parallel algorithms outperform the Baseline, achieving speedups of 1.5x to 2x on average. Circulant tends to be 10-30% faster than Bruck across most configurations. However, these gains came at the cost of an increased extra memory usage. While this trade-off proved beneficial in our settings, it may become problematic for larger amounts of data.

References

[1] Wikipedia contributors, "K-way merge algorithm — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/K-way_merge_algorithm

[2] K. Knapp, "The loser tree data structure: How to optimize merges and make your programs run faster." Apr. 23, 2024.

[3] K. Swaminathan, G. Lakshminarayanan, and S. Ko, "High Speed Generic Network Interface for Network on Chip Using Ping Pong Buffers," in *2012 International Symposium on Electronic System Design (ISED)*, 2012. doi: 10.1109/ISED.2012.11.

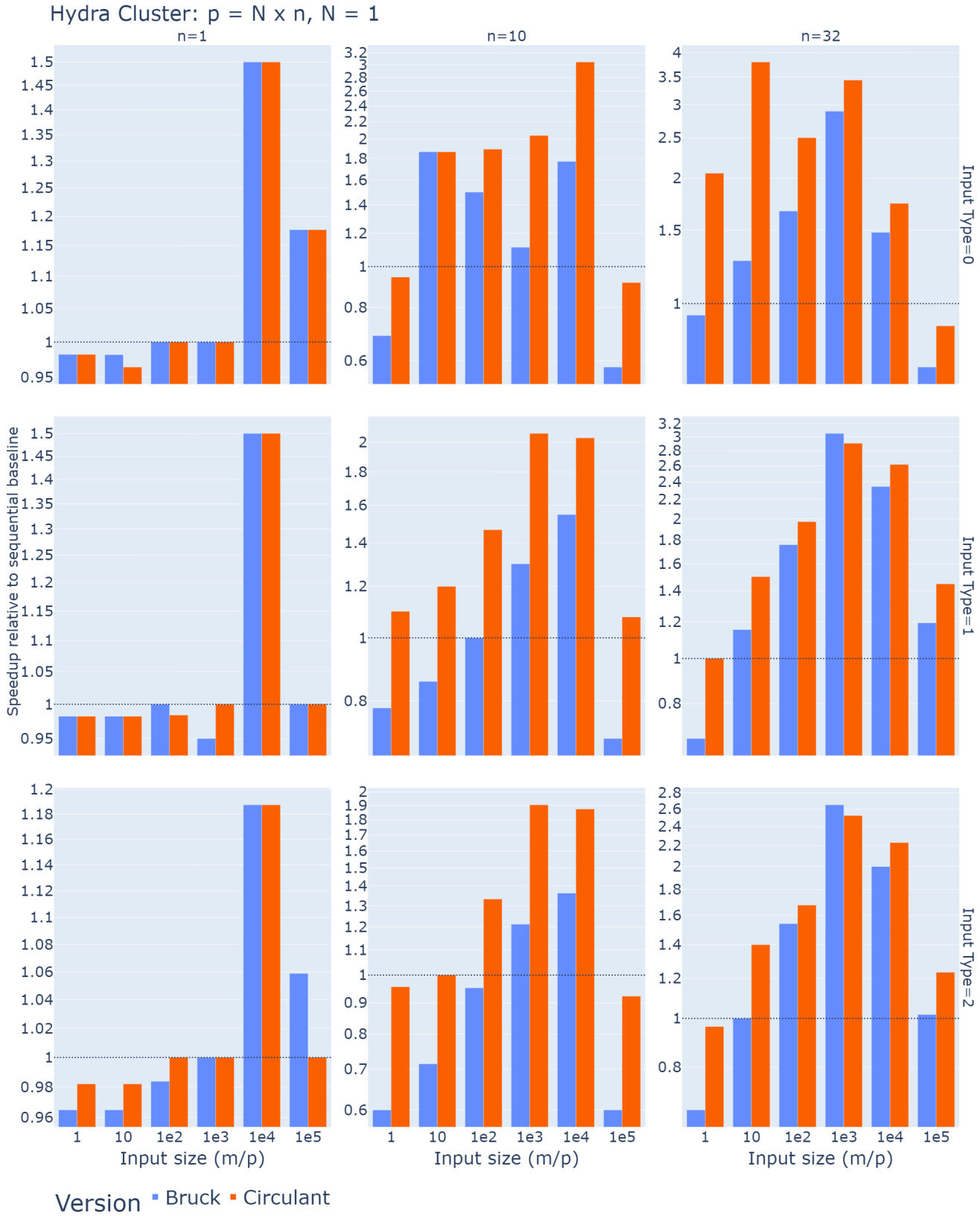


Fig. 10: Comparison of the speedup of the final Bruck and Circulant implementations relative to the sequential Baseline version for a single node.

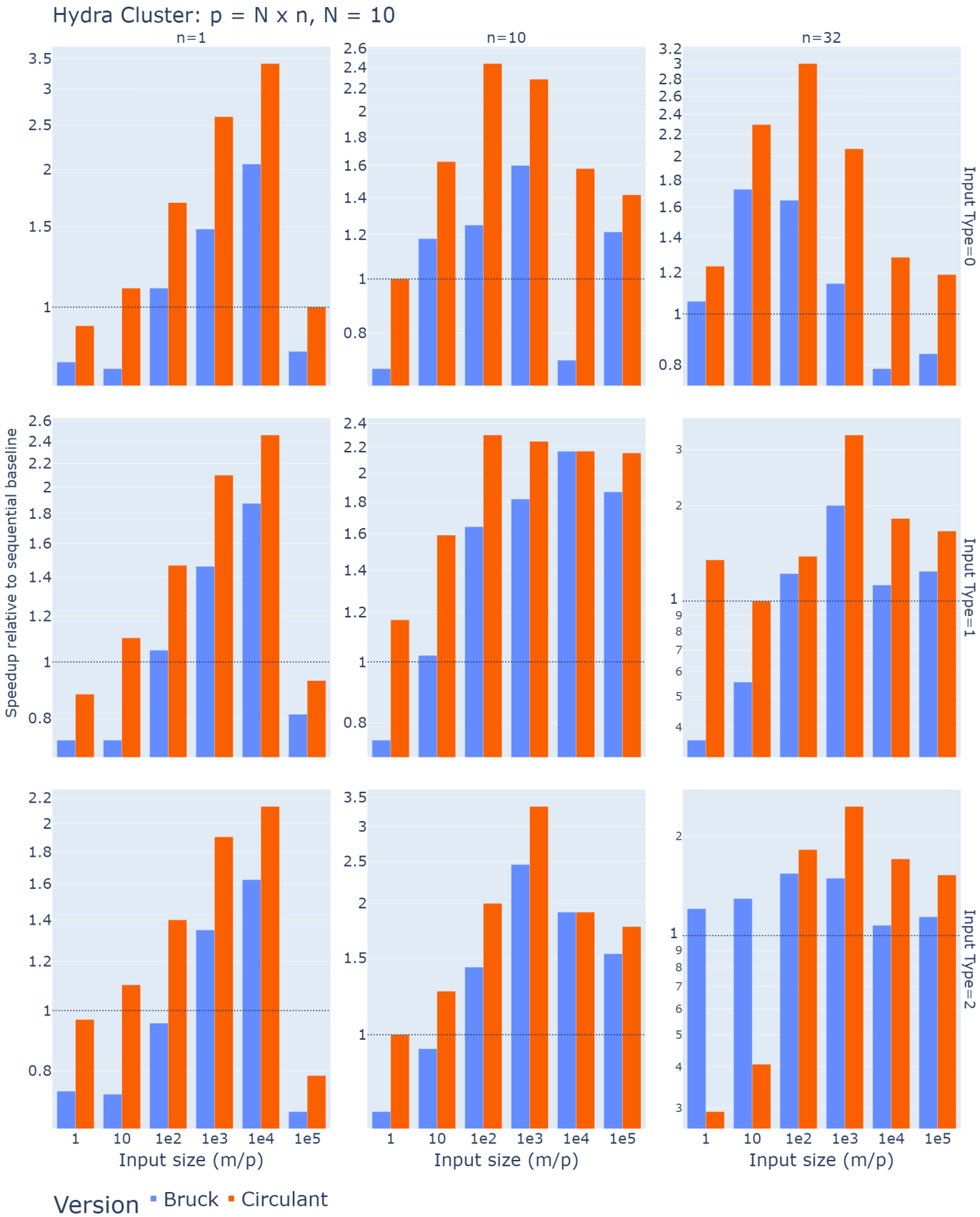


Fig. 11: Comparison of the speedup of the final Bruck and Circulant implementations relative to the sequential Baseline version for 10 nodes.

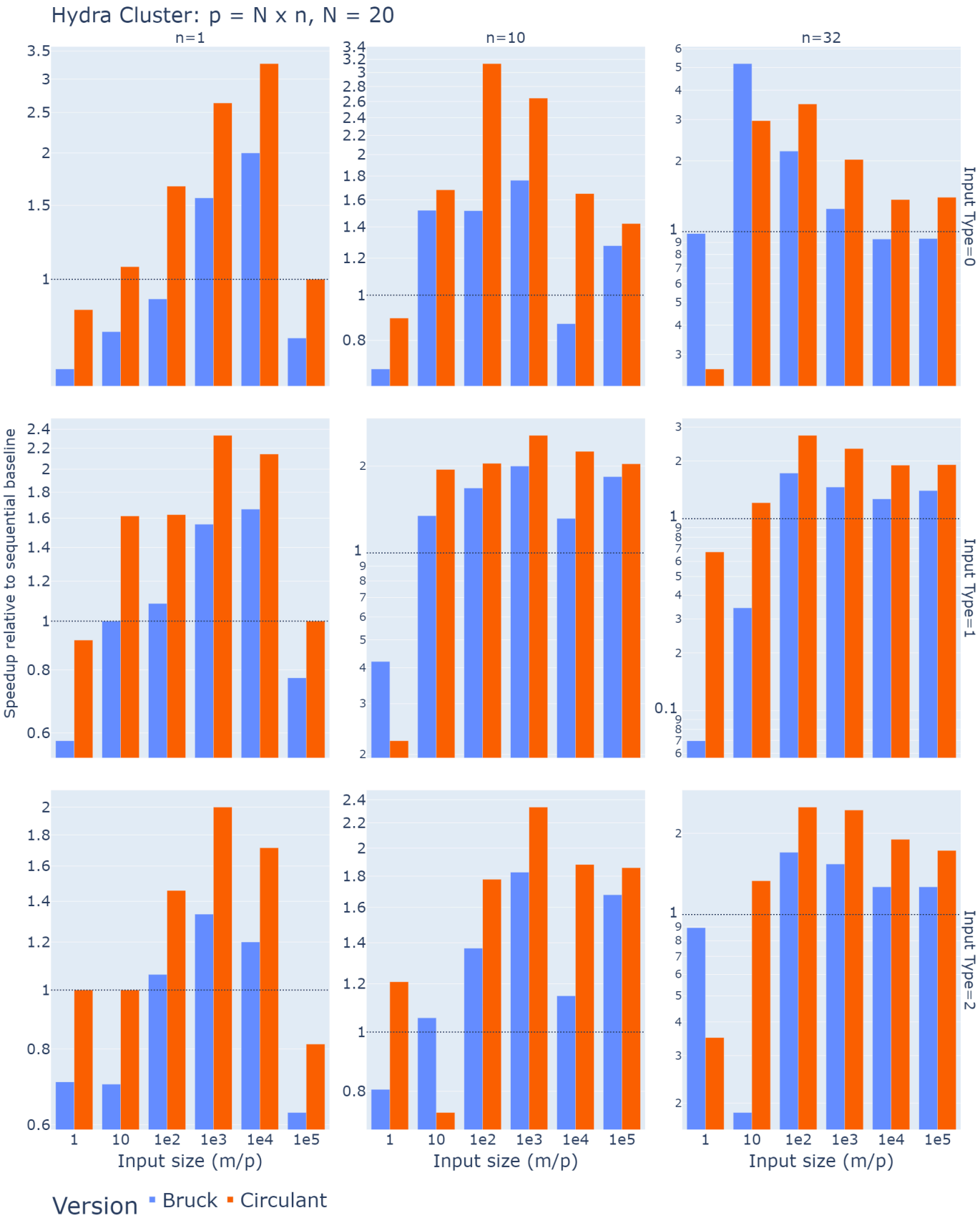


Fig. 12: Comparison of the speedup of the final Bruck and Circulant implementations relative to the sequential Baseline version for 20 nodes.

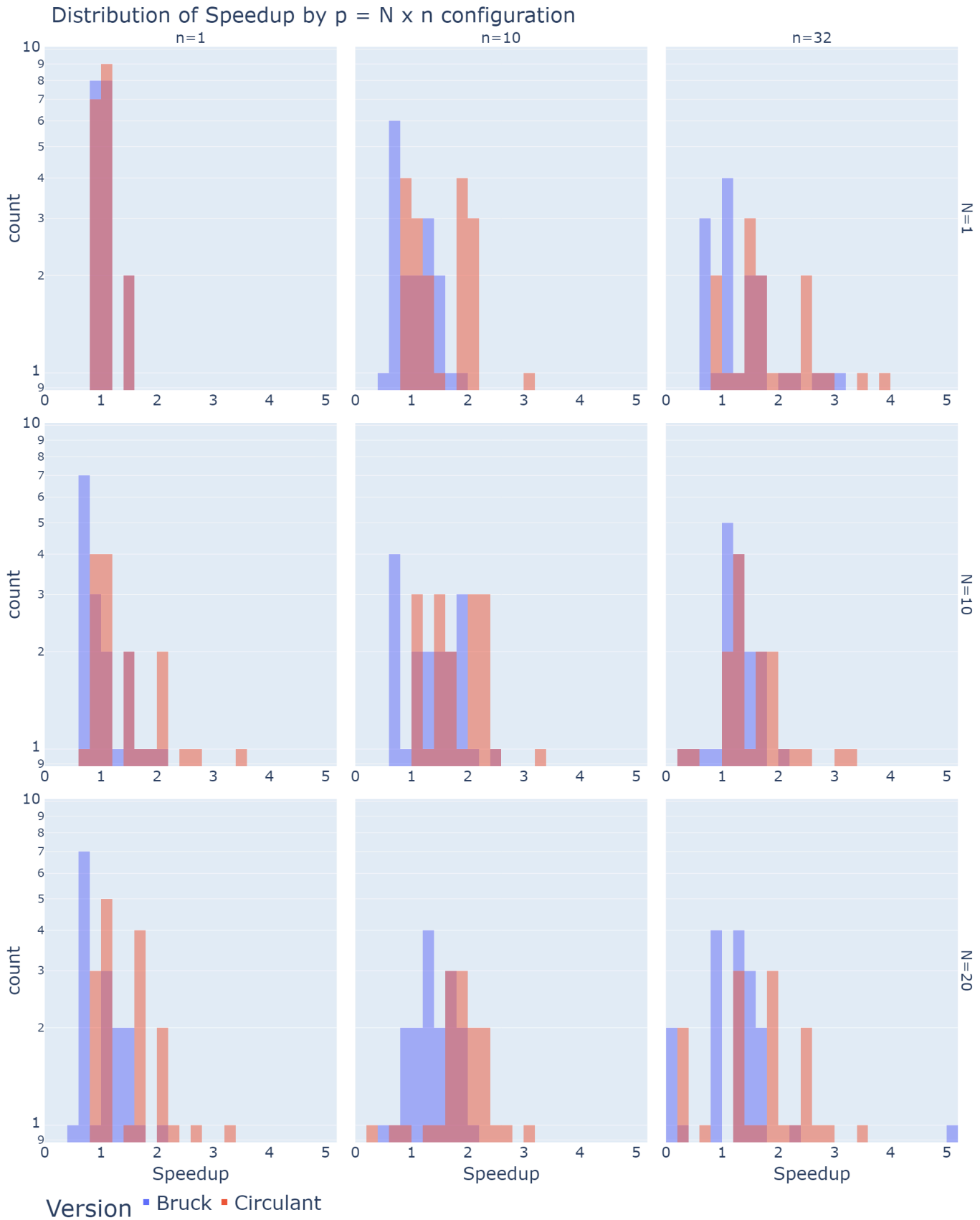


Fig. 13: Distribution of speedup of the final Bruck and Circulant implementations relative to the sequential Baseline by node configuration.

TABLE III: Performance of algorithms (execution time in microseconds). Configuration with $p = N \times n$ MPI processes, where N is the number of nodes and n the number of processes(tasks) per node.

Configuration $p = 1 \times 1$							
Algorithm	**Type**	**Message size (number of elements)**					
		1	**10**	**100**	**1000**	**10000**	**100000**
Baseline	0	5.58e-02	5.49e-02	6.23e-02	1.97e-01	2.45e+00	2.08e+01
	1	5.55e-02	5.50e-02	6.12e-02	1.97e-01	2.49e+00	1.78e+01
	2	5.57e-02	5.51e-02	6.11e-02	1.98e-01	1.94e+00	1.81e+01
Bruck	0	5.68e-02	5.57e-02	6.21e-02	1.98e-01	1.68e+00	1.72e+01
	1	5.68e-02	5.65e-02	6.16e-02	2.00e-01	1.68e+00	1.73e+01
	2	5.79e-02	5.70e-02	6.26e-02	1.98e-01	1.69e+00	1.78e+01
Circulant	0	5.67e-02	5.62e-02	6.21e-02	1.99e-01	1.69e+00	1.75e+01
	1	5.65e-02	5.64e-02	6.20e-02	1.97e-01	1.64e+00	1.74e+01
	2	5.69e-02	5.67e-02	6.11e-02	1.97e-01	1.67e+00	1.80e+01

TABLE IV: Performance of algorithms (execution time in microseconds). Configuration with $p = N \times n$ MPI processes, where N is the number of nodes and n the number of processes(tasks) per node.

Configuration $p = 1 \times 10$							
Algorithm	**Type**	**Message size (number of elements)**					
		1	**10**	**100**	**1000**	**10000**	**100000**
Baseline	0	6.84e+00	2.89e+01	3.64e+01	1.05e+02	1.42e+03	1.10e+04
	1	7.82e+00	1.22e+01	2.22e+01	1.37e+02	1.34e+03	1.43e+04
	2	6.64e+00	1.05e+01	2.03e+01	1.22e+02	1.22e+03	1.26e+04
Bruck	0	9.99e+00	1.51e+01	2.45e+01	9.07e+01	7.92e+02	1.90e+04
	1	1.08e+01	1.49e+01	2.23e+01	1.02e+02	8.48e+02	2.00e+04
	2	1.10e+01	1.48e+01	2.19e+01	9.95e+01	8.89e+02	2.01e+04
Circulant	0	7.26e+00	1.54e+01	1.90e+01	4.94e+01	4.68e+02	1.20e+04
	1	7.14e+00	1.01e+01	1.56e+01	6.30e+01	6.44e+02	1.37e+04
	2	6.91e+00	1.04e+01	1.53e+01	6.34e+01	6.45e+02	1.35e+04

TABLE V: Performance of algorithms (execution time in microseconds). Configuration with $p = N \times n$ MPI processes, where N is the number of nodes and n the number of processes(tasks) per node.

Configuration $p = 1 \times 32$							
Algorithm	**Type**	**Message size (number of elements)**					
		1	**10**	**100**	**1000**	**10000**	**100000**
Baseline	0	1.54e+01	3.86e+01	6.51e+01	5.52e+02	4.08e+03	3.85e+04
	1	7.40e+00	1.50e+01	6.54e+01	6.16e+02	6.80e+03	6.84e+04
	2	7.90e+00	1.41e+01	5.72e+01	5.38e+02	5.80e+03	5.85e+04
Bruck	0	1.68e+01	3.07e+01	3.93e+01	1.97e+02	2.78e+03	5.47e+04
	1	1.14e+01	1.38e+01	3.71e+01	2.07e+02	2.96e+03	5.71e+04
	2	1.22e+01	1.42e+01	3.74e+01	2.07e+02	2.99e+03	5.71e+04
Circulant	0	7.38e+00	1.09e+01	2.69e+01	1.67e+02	2.32e+03	4.37e+04
	1	7.44e+00	1.06e+01	3.34e+01	2.10e+02	2.67e+03	4.78e+04
	2	8.23e+00	1.04e+01	3.41e+01	2.10e+02	2.67e+03	4.77e+04

TABLE VI: Performance of algorithms (execution time in microseconds). Configuration with $p = N \times n$ MPI processes, where N is the number of nodes and n the number of processes(tasks) per node.

Configuration $p = 10 \times 1$							
Algorithm	**Type**	**Message size (number of elements)**					
		1	**10**	**100**	**1000**	**10000**	**100000**
Baseline	0	9.18e+00	1.14e+01	2.21e+01	1.28e+02	1.55e+03	1.20e+04
	1	8.85e+00	1.12e+01	2.22e+01	1.39e+02	1.52e+03	1.32e+04
	2	8.99e+00	1.10e+01	2.12e+01	1.28e+02	1.32e+03	1.13e+04
Bruck	0	1.26e+01	1.52e+01	2.09e+01	8.12e+01	7.37e+02	1.51e+04
	1	1.20e+01	1.55e+01	2.17e+01	8.92e+01	8.00e+02	1.61e+04
	2	1.22e+01	1.57e+01	2.27e+01	8.94e+01	8.09e+02	1.62e+04
Circulant	0	1.01e+01	1.03e+01	1.37e+01	4.69e+01	4.42e+02	1.26e+04
	1	1.02e+01	1.08e+01	1.59e+01	6.28e+01	6.18e+02	1.46e+04
	2	9.21e+00	1.06e+01	1.53e+01	6.34e+01	6.19e+02	1.47e+04

TABLE VII: Performance of algorithms (execution time in microseconds). Configuration with $p = N \times n$ MPI processes, where N is the number of nodes and n the number of processes(tasks) per node.

Configuration $p = 10 \times 10$							
Algorithm	**Type**	**Message size (number of elements)**					
		1	**10**	**100**	**1000**	**10000**	**100000**
Baseline	0	2.01e+01	5.26e+01	2.09e+02	1.64e+03	1.50e+04	1.70e+05
	1	2.13e+01	4.31e+01	2.30e+02	2.04e+03	2.62e+04	2.82e+05
	2	2.08e+01	3.93e+01	2.08e+02	2.74e+03	2.12e+04	2.37e+05
Bruck	0	2.95e+01	4.41e+01	1.66e+02	1.02e+03	2.14e+04	1.46e+05
	1	2.89e+01	4.20e+01	1.43e+02	1.14e+03	1.21e+04	1.51e+05
	2	3.04e+01	4.21e+01	1.43e+02	1.11e+03	1.19e+04	1.51e+05
Circulant	0	2.02e+01	3.26e+01	8.23e+01	7.08e+02	9.50e+03	1.23e+05
	1	1.83e+01	2.79e+01	1.00e+02	8.94e+02	1.20e+04	1.38e+05
	2	2.00e+01	3.11e+01	1.00e+02	8.19e+02	1.10e+04	1.38e+05

TABLE VIII: Performance of algorithms (execution time in microseconds). Configuration with $p = N \times n$ MPI processes, where N is the number of nodes and n the number of processes(tasks) per node.

Configuration $p = 10 \times 32$							
Algorithm	**Type**	**Message size (number of elements)**					
		1	**10**	**100**	**1000**	**10000**	**100000**
Baseline	0	3.76e+01	1.44e+02	8.44e+02	6.41e+03	5.91e+04	6.33e+05
	1	3.10e+01	1.02e+02	8.33e+02	1.25e+04	9.10e+04	9.85e+05
	2	4.13e+01	1.16e+02	9.11e+02	9.16e+03	8.77e+04	9.09e+05
Bruck	0	3.58e+01	8.15e+01	5.10e+02	5.67e+03	7.53e+04	7.54e+05
	1	8.51e+01	1.81e+02	6.81e+02	6.05e+03	8.12e+04	7.91e+05
	2	3.47e+01	8.58e+01	5.96e+02	6.10e+03	8.18e+04	7.95e+05
Circulant	0	3.05e+01	6.13e+01	2.84e+02	3.11e+03	4.62e+04	5.31e+05
	1	2.37e+01	1.02e+02	6.07e+02	3.67e+03	5.09e+04	5.90e+05
	2	1.48e+02	2.79e+02	5.04e+02	3.71e+03	5.13e+04	5.90e+05

TABLE IX: Performance of algorithms (execution time in microseconds). Configuration with $p = N \times n$ MPI processes, where N is the number of nodes and n the number of processes(tasks) per node.

Configuration $p = 20 \times 1$							
Algorithm	**Type**	**Message size (number of elements)**					
		1	**10**	**100**	**1000**	**10000**	**100000**
Baseline	0	1.19e+01	1.54e+01	3.54e+01	2.59e+02	3.21e+03	2.16e+04
	1	1.17e+01	2.15e+01	3.92e+01	2.82e+02	3.08e+03	2.71e+04
	2	1.24e+01	1.40e+01	3.55e+01	2.46e+02	2.47e+03	2.28e+04
Bruck	0	1.85e+01	2.07e+01	3.90e+01	1.67e+02	1.66e+03	2.97e+04
	1	1.94e+01	2.17e+01	3.65e+01	1.81e+02	1.82e+03	3.52e+04
	2	1.72e+01	2.07e+01	3.33e+01	1.82e+02	2.05e+03	3.54e+04
Circulant	0	1.37e+01	1.49e+01	2.17e+01	9.59e+01	9.84e+02	2.14e+04
	1	1.20e+01	1.32e+01	2.47e+01	1.28e+02	1.45e+03	2.77e+04
	2	1.20e+01	1.46e+01	2.44e+01	1.20e+02	1.43e+03	2.74e+04

TABLE X: Performance of algorithms (execution time in microseconds). Configuration with $p = N \times n$ MPI processes, where N is the number of nodes and n the number of processes(tasks) per node.

Configuration $p = 20 \times 10$							
Algorithm	**Type**	**Message size (number of elements)**					
		1	**10**	**100**	**1000**	**10000**	**100000**
Baseline	0	2.57e+01	7.97e+01	4.76e+02	3.75e+03	3.32e+04	3.78e+05
	1	2.98e+01	7.44e+01	4.77e+02	4.61e+03	5.43e+04	5.72e+05
	2	2.95e+01	9.65e+01	4.80e+02	4.25e+03	4.79e+04	5.22e+05
Bruck	0	3.61e+01	5.22e+01	3.17e+02	2.12e+03	3.81e+04	2.93e+05
	1	6.98e+01	5.54e+01	2.84e+02	2.37e+03	4.14e+04	3.16e+05
	2	3.67e+01	9.13e+01	3.51e+02	2.38e+03	4.14e+04	3.15e+05
Circulant	0	2.83e+01	4.71e+01	1.50e+02	1.46e+03	2.09e+04	2.61e+05
	1	1.35e+02	3.89e+01	2.35e+02	1.82e+03	2.47e+04	2.87e+05
	2	2.42e+01	1.39e+02	2.70e+02	1.82e+03	2.50e+04	2.86e+05

TABLE XI: Performance of algorithms (execution time in microseconds). Configuration with $p = N \times n$ MPI processes, where N is the number of nodes and n the number of processes(tasks) per node.

Configuration $p = 20 \times 32$							
Algorithm	**Type**	**Message size (number of elements)**					
		1	**10**	**100**	**1000**	**10000**	**100000**
Baseline	0	5.23e+01	8.37e+02	2.29e+03	1.51e+04	1.34e+05	1.44e+06
	1	5.35e+01	2.32e+02	1.91e+03	1.98e+04	1.92e+05	2.11e+06
	2	4.26e+01	2.47e+02	1.77e+03	2.01e+04	1.94e+05	1.99e+06
Bruck	0	5.38e+01	1.67e+02	1.01e+03	1.29e+04	1.45e+05	1.52e+06
	1	7.60e+02	6.76e+02	1.18e+03	1.35e+04	1.50e+05	1.58e+06
	2	4.78e+01	1.36e+03	1.08e+03	1.30e+04	1.50e+05	1.59e+06
Circulant	0	2.06e+02	2.89e+02	6.36e+02	7.49e+03	9.52e+04	1.09e+06
	1	7.90e+01	1.94e+02	7.09e+02	8.24e+03	1.04e+05	1.18e+06
	2	1.29e+02	1.80e+02	6.83e+02	8.21e+03	1.05e+05	1.18e+06