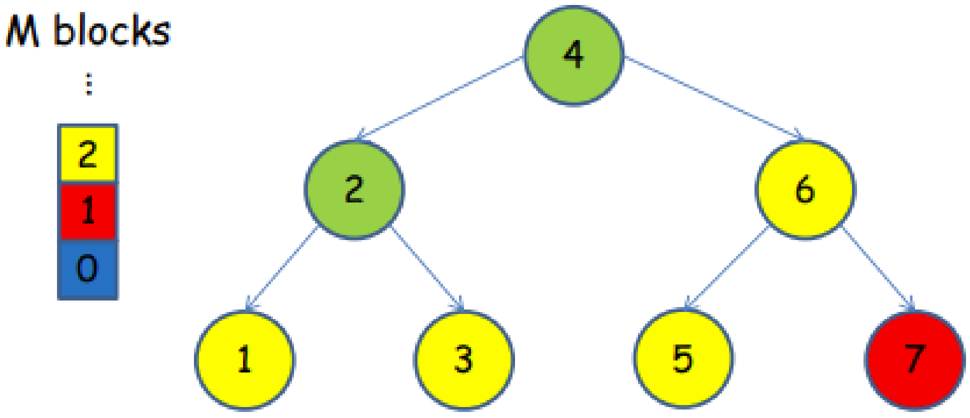
184.725 High Performance Computing

Programming Project Report

Benjamin Gruber (11770987) — Dominik Freinberger (11708140)

October 12, 2023

Implementation, improvement and comparison of different pipelined MPI-Allreduce algorithms for (non-)commutative operators on the TU-Wien 'Hydra' cluster



"The key to understanding the algorithm by Träff lies within the binary-tree-depth" - Dominik Freinberger

Contents

References

[Mes21] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0.* June 2021. URL: `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`.

[Trä21] Jesper Larsson Träff. "A Doubly-pipelined, Dual-root Reduction-to-all Algorithm and Implementation". In: *CoRR* abs/2109.12626 (2021). arXiv: `2109.12626`. URL: `https://arxiv.org/abs/2109.12626`.

[Trä22] Jesper Larsson Träff. *Algorithms for Collective Communication.* 2022. URL: `https://arxiv.org/abs/2109.12626`.

1 Exercise 1 (10 points)

In the first Exercise we implemented a Benchmark that enables us to compare Allreduce implementations of varying complexity to a reference MPI-implementation. This is with regard to computation time and in dependence of the number of nodes utilized, the size of the array underlying the operation and the BlockSize, the latter will be segmented by. We continue by discussing the Benchmark's architecture and then give information about the experimental setup for this and all following exercises and define the machine environment, data types and MPI-libraries. We will also use this chance to highlight the significance of the dual-rail characteristic of the 'Hydra'-system. We then implemented a naive Allreduce operation as combination of the two independent `MPI_Reduce` and `MPI_Bcast` operations which are performed one after another. We bring this section to an end by conducting the runtime experiment with the MPI-Allreduce as reference by running the implementation in `benchmark.c`.

1.1 The Benchmark

The Benchmark for this project is built upon the basis provided via the TUWEL-platform. The latter's statistical output concerning run-times has been expanded by median, standard-deviation, lower and upper-bounds of the confidence interval. All time-based statistical values are cast to MICRO seconds and all experiments are run with the `MPI_FLOAT` data type.

The confidence level can be controlled via run-time variable for the Z-value. The confidence achieved by a certain Z-value is given in table 1. The confidence intervals presented in this report (Appendix A) are all corresponding to confidence 0.95. The number of measurement repetitions for any given setup is determined in dependence of count, such that 21 repetitions are executed if count is very little and at least 2 repetitions are executed for $count_{max}$. There are 2 unmeasured 'warms-up' runs prior to every run. The odd number of runs is deliberate, since it simplifies the median computation.

confidence	Z-value
0.8	1.282
0.85	1.440
0.9	1.645
0.95	1.960
0.99	2.576

Table 1: z-values

The Benchmark also features a correctness test prior to building performance statistics. The test is conducted for the Exercise 1-5 implementations, against the MPI-Allreduce reference and on the maximum vector length (*count*). In case of non-identical results a message is written to the standard error device and abort is called, terminating the program execution.

In order to enable a lean stack of slurm-scripts on the 'Hydra' queue, run-time information corresponding to the Allreduce-implementations from Exercise 1-5, as well as, the MPI-Allreduce (as reference) are gathered within one run of the *benchmark.c* file. The benchmark-run takes time measurements for different array sizes and BlockSizes (where applicable) according to the 'Specification'-section in table 2. However, the configuration of utilized nodes in the 'Hydra' system has to be adjusted in the *mpirun* call.

The machine environment log for the 'Hydra' system can be found in table 2. An important characteristic of the systems is it's dual-rail capability. While many High-Performance-Clusters and TOP500-machines are capable of only either incoming or outgoing communication between any node and it's neighbour within one clock cycle, dual-rail enables simultaneous communication in both directions. The direct implication of this is the utilization of the `MPI_SendReicv` operation, which facilitates simultaneous communication in both ways as compared to separate `MPI_Send` and `MPI_Recv` calls, which would account for two clock cycles.

Benchmark properties				
Machine Env	'Hydra'-system: 36 nodes with dual-socket Intel Xeon Gold 6139 cores; 2.1 GHz clock-speed; dual-rail Intel OmniPath-network			
MPI Library	OpenMPI 4.4			
Compiler	gcc 8.3.0			
Flages	-O3 -lm			
Data Type	`MPI_FLOAT`			
Reduction	`MPI_MAX`			
Specifications	method	nodes x $processes$	count	blockSize
Exercise 1	AR_{Lin}	1x16,32; 20x1,16,32; 36x1,16,32	1,2,1e^2,2^9,2^{12},1e^4,2^{16},2^{19},1e^6, 1e^7	$underthehood$
Exercise 2	$\text{AR}_{Lin\|Pipeline}$	1x16,32; 20x1,16,32; 36x1,16	1,2,1e^2,2^9,2^{12},1e^4,2^{16},2^{19},1e^6, 1e^7	$1e1, 1e2, \ldots, 1e7$
Exercise 3	$\text{AR}_{Lin\|2xPipeline}$	1x16,32; 20x1,16,32; 36x1,16	1,2,1e^2,2^9,2^{12},1e^4,2^{16},2^{19},1e^6, 1e^7	$\frac{count}{4}$, ..., $count$
Exercise 4	$\text{AR}_{Binary\|Pipeline}$	1x16,32; 20x1,16,32; 36x1,16,32	1,2,1e^2,2^9,2^{12},1e^4,2^{16},2^{19},1e^6, 1e^7	$\frac{count}{4}$, ..., $count$
Exercise 5	$\text{AR}_{Binary\|2xPipeline}$	1x16,32; 20x1,16,32; 36x1,16,32	1,2,1e^2,2^9,2^{12},1e^4,2^{16},2^{19},1e^6, 1e^7	$\frac{count}{4}$, ..., $count$

Table 2: Experimental Set-Up for exercises 1-5

1.2 Naive combination of `MPI_Reduce` and `MPI_Bcast`

The MPI-library provides implementations of both non-pipelined Broadcast and Reduction directives (corresponding documentation can be found in the current MPI-Standard [Mes21]).

If one reads the MPI-Standard's [Mes21] description of the Broadcast directive[1] one will quickly conclude that it does not enclose, which communication structure(s) are utilized in the MPI implementations.

However, after some more research, we can establish that both `MPI_Bcast`, `MPI_Reduce` and `MPI_Allreduce` are indeed conglomerates of multiple algorithms, of which the most fitting is chosen depending on a myriad of factors.

1.3 Results and Discussion

Figure 2 in the Appendix A depicts the experimental findings building the basis for this analysis. Both the reference run-times and those corresponding to the Linear, Pipelined Allreduce algorithm perform very similarly. For increasing problem sizes (indicated by $count$) execution time increases exponentially. Considering the greater picture, the reference application performs slightly better than the Naive combination algorithm. This is most likely due to utilization of Doubly Pipelined effects in the $MPI_{Allreduce}$ implementation.

[1] *"`MPI_Bcast` broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of the group **using the same arguments for comm and root**. On return, the content of root's buffer is copied to all other processes."*

2 Exercise 2 (10 points)

In the second Exercise we improve the linear Allreduce implementation presented in Exercise 1 by coming up with pipelined versions for the Reduce and Broadcast operation (this is by introducing the BlockSize as the partition of data to be communicated within one timestep). Firstly, we will give an overview of the algorithm and discuss how the right balance between number of communication rounds and amount of data to be sent per communication round can impact the implementation's runtime. Secondly, we discuss if and when the implementation requires a commutative reduction operation and establish a running time estimate (based on a round-based, linear transmission cost model). We further discuss the implementation runtimes against those from Exercise 1. We will do this with special hindsight to the impact of process configurations and problem sizes.

2.1 Mapping and Pipelining

The characteristic 'linear' refers to the node-mapping the algorithm uses as underlying communication structure, where every node, other than the first and the last, has two communication ways. One to it's predecessor and one to it's successor.

Figure 1: A linear processor array Lp for some number of processors p [..] [Trä22]

One could also argue that linear Reduction and Broadcast act on a linear ring structure (node 0 and node p-1 are connected), however, there are always two consecutive nodes that do not interact with each other, which makes the cases equivalent.

As anticipation to the approach to Exercise 3 presented in this report, one could also interpret the linear communication structure as a tree structure, in which the inner nodes only have one parent and one child. In other words "A linear processor array is isomorphic to a rooted, 1-ary tree." [Trä22]

A directive can be described as pipelined, if the whole information stored in one node is sent to it's communication partner in chunks over multiple rounds rather than in one go. The size of these chunks is determined by the BlockSize variable.

The cost (in regards of time) of a single communication between two nodes depends on the amount of data that is subject to it. For unpipelined algorithms this cost become very high, since all the data is sent in the same round. This can lead to increased idleness in the other nodes, because they have to wait for data. While splitting a large chunk of data into multiple parts introduces overhead by making multiple communication instances necessary, idleness in nodes ahead (in the structure) can be decreased. The advantages of pipelined All-to-One and One-to-All directives becomes apparent if the BlockSize parameter (that is the number of extra communication rounds) is optimized such, that the net decrease in run-time is maximized.

2.2 Linear Pipelined Allreduce

Algorithm A Pipelined Reduce algorithm as performed by processor i. Parent and child will be determined. The length $count$ of the buffer Y and the pipelining blocksize $blockSize$ (which is also the length of the temporary buffer T) are passed as arguments. $Leaf$ corresponds to the last node in the linear communication structure. Note that this description only features the directives that apply to the part of buffer Y that can be segmented without remainder by $blockSize$. The actual implementation features a directive along this algorithm's line that deals with the remaining elements.

```
1:  determine $child, parent$
2:  determine number of full sized blocks $chunk_{idx}$
3:  initiate temporary buffer $T$
4:  if root then
5:      for $j = 0, blockSize, 2blockSize, \ldots, (chunk_{idx} - blockSize)$ do
6:          Recv($T$, child)                                        ▷ Child
7:          $Y[j] \leftarrow T \odot Y[j]$                           ▷ In-order reduction
8:  else if leaf then
9:      for $j = 0, blockSize, 2blockSize, \ldots, (chunk_{idx} - blockSize)$ do
10:         Send($Y[j]$, parent)                                    ▷ parent
11: else
12:     Recv($T, child_0$)                                          ▷ Child
13:     $Y[0] \leftarrow T \odot Y[0]$              ▷ In-order reduction pre SendRecv utilization
14:     for $j = 0, blockSize, 2blockSize, \ldots, (chunk_{idx} - blockSize)$ do
15:         Send($Y[j - blockSize]$, parent) || Recv($T$, child)    ▷ to parent, from child
16:         $Y[j] \leftarrow T \odot Y[j]$
17:     end for
18:     Send($Y[chunk_{idx}]$, child)
    end if
```

Every node in the communication structure has a receive- and temporary-buffer. The latter is of size $BlockSize$, the former of size $count$. Nodes hold their 'own' information in the receive-buffer at round 0 and receive blockwise information into their temporary-buffer, which in turn is then reduced into the receive-buffer. Consequently their current 'reduced knowledge' is held in the receive-buffer, hence this is also from where they send information to their left neighbour.

The first step of the Linear Pipelined Allreduce operation is the reduction. Consider the linear communication structure in 1 and set node 0 as root. All other nodes send their data leftwards from the leaf (node $p - 1$) to the root node, where the result is calculated and saved. Depending on whether a node is root, leaf (node $p - 1$) or an inner node, it will have a different set of instructions: While the root node is only receiving the data, the leaf node is only sending data towards the root. Inner nodes both receive data from the right and send data to their left.[2] This process is pursued blockwise, such that node p obtains block j (from node $p + 1$) during the same round as it sends block $j - 1$ (to node $p - 1$) until the root node holds reduced information about all blocks.

Both operations (Reduction and Broadcast) are designed such that they can deal with $blockSizes$ that do not divide $count$ without reminder. This is achieved by finding out at which position in the array the last block (with $size < blockSize$) starts and sending only the remainder of $\frac{count}{blockSize}$ elements from here. This is an inefficient (at least in terms of code lines) solution compared to the

[2]This is implemented with $MPI_{SendReceiv()}s$ in order to utilize 'Hydra's dual-rail capability

one utilizing the `MPI_Status` and `MPI_Get_count` to find out how many elements were sent[3]).

Algorithm B Pipelined Broadcast algorithm as performed by processor i. Parent and child will be determined. The length $count$ of the buffer Y and the pipelining blocksize $blockSize$ (which is also the length of the temporary buffer T) are passed as arguments. $leaf$ corresponds to the last node in the linear communication structure. Note that this description only features the directives that apply to the part of buffer Y that can be segmented without remainder by $blockSize$. The actual implementation features a directive along this algorithm's line that deals with the remaining elements.

```
1: determine $child, parent$
2: determine number of full sized blocks $chunk_{idx}$
3: if root then
4:     for $j = 0, blockSize, 2blockSize, \ldots, (chunk_{idx} - blockSize)$ do
5:         Send($Y[j - blockSize]$)                                        ▷ Child
6: else if leaf then
7:     for $j = 0, blockSize, 2blockSize, \ldots, (chunk_{idx} - blockSize)$ do
8:         Recv($Y[j]$, parent)                                            ▷ parent
9: else
10:    Recv($Y[0]$, $child_0$)                                             ▷ parent
11:    for $j = 0, blockSize, 2blockSize, \ldots, (chunk_{idx} - blockSize)$ do
12:        Send($Y[j - blockSize]$, child) || Recv($Y[j]$, parent)    ▷ to child, from parent
13:    end for
14:    Send($Y[chunk_{idx} - blockSize]$, child)
   end if
```

Now the Broadcast operation can start spreading the reduced blocks rightwards and all the way to the leaf. This can be interpreted as an inverse Reduction without the round-wise Reduction operations. The set of instructions also turns around: Now the root node is only performing $Send()$ (to the right), while the leaf node only receives data (from the left). The Reduction is implemented in the `reduce()` function, the Broadcast in the `bcast()` function. Both algorithms are combined in the `allReduce2()` function.

2.3 Assessing the ideal BlockSize

As demonstrated in Figure 4 in the Appendix A there is a clear optimum for the BlockSize parameter. For a count of $1 \cdot 10^7$ elements the BlockSize was varied in powers of ten, going from 10 to $1 \cdot 10^7$. The upper limit corresponds to sending all of the data in one run, which results in no pipelining being done. For all intermediate configurations (meaning no extremes, such as 1 node multiple processes or vice-versa), the ideal BlockSize is found to be $1 \cdot 10^4$. In the edge cases of the $20x1$ and $36x1$ configurations a BlockSize of one order smaller seems to be the optimum, whereas for the other extreme of $1x16$ and $1x32$ the ideal BlockSize is of one order higher than the intermediate one. Note, that we were not able to test the $36x32$ configuration, as the job exceeded the timelimit.

[3]We only found out about this at a later stage in the progress, so this is utilized in Exercise 3 and 5

2.4 Requirements to the commutativity of the Reduction-Operator

Commutativity regarding the Reduction-Operator of a Reduction algorithm in MPI-context can be assessed by a simple condition: If the algorithm can guarantee that 1) elements are reduced in the same order every time and 2) elements are reduced in the order of rank, we can deduce commutativity of the algorithm[4]

In a linear communication structure there is never doubt about which node is sending information 'first', since there is only one possible sender (predecessor). This is different for tree structures (f.e. 2-ary tree: 2 possible senders). Condition 1) is therefore fulfilled.

It should not pose a problem to implement f.e. a minus-operator that will assign the status of minuend and subtrahend to the correct member of the operation. However, this is just an example for a non-commutative Reduction-Operator and there are others that demand Reduction in rank order.

This second condition is not necessarily met by every linear-communication-structure-based Reduction algorithm, since $Reduction(p-1, p-2, .., p1, p0)$ might yield a different result from $Reduction(0, 1, 2, .., p-2, p-1)$. Since for our algorithm node 0 is set as root and communication starts from $p-1$ towards root the latter Reduction-order is ensured and condition 2) fulfilled.

2.5 Running time estimate

We are now estimating the running time for the pipelined Allreduce operation by using a round-based, linear transmission cost model. As a first step, let us introduce the pipelining lemma as presented in [Trä22]:

Lemma 1 (Pipelining Lemma)**.** With latency of d rounds to deliver the first block and a new block every s rounds (with $d > s$), the best time T under the linear cost model (that is: maximum time cost $\alpha + \beta * BlockSize$ and delay $d * (\alpha + \beta * BlockSize)$ for any block) for a pipelined algorithm that divides the message m into M blocks is

$$T(m) = (d-s)\alpha + 2\sqrt{s(d-s)\alpha\beta m} + s\beta m$$

Proof. See [Trä22] on page 76. ■

We first propose an estimated running time for the Broadcast step (second step) of the algorithm.

Corrollary 1 (Linear Pipeline Broadcast Running Time Estimate)**.** The best possible time T_{BCAST} for the pipelined, linear broadcast operation is (with $d = p-1$ and $s = 1$)

$$T_{\text{B}}(m) = (p-2)\alpha + 2\sqrt{(p-2)\alpha\beta m} + \beta m$$

Proof. See [**slides3**] on slide 186-187. ■

We then propose an estimated running time for the Reduction step (first step) of the algorithm:

Corrollary 2 (Linear Reduction Running Time Estimate)**.** The best possible time T_{R} for the pipelined,linear reduction operation is the same as for the pipelined, linear broadcast operation

$$T_{\text{R}}(m) = (p-2)\alpha + 2\sqrt{(p-2)\alpha\beta m} + \beta m$$

Proof. This is because the Reduction is the inverse of the Broadcast and more specifically both operations share the same values for latency k and block frequency s ■

And finally obtain the total estimated running time by summation:

[4]which is equivalent to stating no requirements to the commutativity of the Reduction-Operator

Corrollary 3 (Linear Pipelined Allreduce Running Time Estimate)**.** The best possible time T_{A} for the pipelined, linear Allreduce operation is the sum of the running time estimates for the reduction and broadcast operations

$$T_{\text{A}}(m) = T_{\text{R}}(m) + T_{\text{B}}(m) = \ (2p-4)\alpha + 4\sqrt{(p-2)\alpha\beta m} + 2\beta m$$

In terms of running time this implicates: $\mathcal{O}\big(p + \sqrt{mp}\big) + 2\beta m$. With introduction of binary tree mappings for Allreduce algorithms in Exercise 4 and 5, the order term will be reduced.

2.6 Results and Discussion

Figure 3 in the Appendix A depicts the experimental findings building the basis for this analysis.

Linear communication structures configurations featuring high numbers of nodes (such as configurations $20x32, 20x16, 36x16$, not to mention $36x32$) cause huge extra cost compared to low node number configurations, because block-transport latency increases linearly. In comparison the same latency for tree based structures is in logarithmic dependence of the number of nodes. This effect can most precisely be pointed out based by comparison of equation 4.3 (k) vs. d, the equivalent for linear structures introduced in the Run-time Estimate Section of Exercise 2.

However, the latency is only in dependence of the number of nodes therefore causes a constant overhead on the run-time. For large problem sizes the effect decreases as the overhead in rounds becomes small compared to the number of rounds where all nodes are active. Accordingly, the run-times for low $counts$ to deviate from the reference to a greater extend then those for high $counts$.

It is, after all, surprising that the implementation yields faster run-times for all configurations then the reference implementation for $count > 10^6$. As pointed out in the Appendix A the plots hold the time-data-point corresponding to the $blockSize$ yielding the fastest run-time.

3 Exercise 4 (10 points)

In the fourth Exercise we map the pipelined Allreduce algorithm onto a (BFS ordered) binary tree communication structure, as opposed to Exercise 1 and 2, where we used a linear communication structure. We will give an overview of the algorithm and discuss how a binary tree's flattened hierarchy (compared to a linear structure) can impact the number of necessary communication rounds. Afterwards we will discuss the implementation run-times against those from the previous Exercises and the MPI-Allreduce reference application.

3.1 BinaryTree Pipelined Allreduce

Algorithm C Pipelined Reduce algorithm as performed by processor i. The corresponding level d_i from below in which the processor is sitting in the binary tree, as well as parents and children will be determined. The length $count$ of the buffer Y and the pipelining blocksize $blockSize$ (which is also the length of the temporary buffer T) are passed as arguments. Note that this description only features the directives that apply to the part of buffer Y that can be segmented without remainder by $blockSize$. The actual implementation features a directive along this algorithm's line that deals with the remaining elements.

```
1: determine $child_R, child_L, parent$
2: determine tree depth $d$
3: determine number of full sized blocks $chunk_{idx}$
4: initiate temporary buffer $T$
5: **if** root **then**
6:     **for** $j = 0, blockSize, 2blockSize, \dots, (chunk_{idx} - blockSize)$ **do**
7:         Recv($T$, child$_0$)                                   ▷ Left Child
8:         $Y[j] \leftarrow T \odot Y[j]$                          ▷ In-order reduction
9:         Recv($T$, child$_1$)                                   ▷ Right Child
10:        $Y[j] \leftarrow T \odot Y[j]$                          ▷ In-order reduction
11: **else if** leaf **then**
12:     **for** $j = 0, blockSize, 2blockSize, \dots, (chunk_{idx} - blockSize)$ **do**
13:         Send($Y[j]$, parent)                                  ▷ Parent
14: **else**
15:     **for** $j = 0, blockSize, 2blockSize, \dots, (chunk_{idx} - blockSize)$ **do**
16:         Recv($T$, $child_0$)                                  ▷ Left Child
17:         Recv($T$, $child_1$)                                  ▷ Right Child
18:         Send($Y[j]$, parent)                                  ▷ Parent
```

Similar to the linear communication structure from Exercise 2 every node in the tree communication structure has an equally configured receive- and temporary-buffer (the latter is of size $BlockSize$, the former of size $count$). Again nodes hold their 'own' information in the receive-buffer at round 0 and receive blockwise information into their temporary-buffer, which in turn is then reduced into the receive-buffer. While the linear structure algorithm only featured one Receive-Reduction operation, the binary tree based one features two of these in sequence (one per child). Only after both the node's current 'reduced knowledge' (held in the receive-buffer) is up-to-date. 'Upwards' propagation to the parent remains the same as with the linear structure, since the number of parents has not changed (1).

While the root node is only receiving the data, the leaf nodes are only sending data. Inner nodes pursue communication both up- and downwards. The segmentation of the total data into chunks complies with the one described in Exercise 2.

Algorithm D Pipelined broadcast algorithm as performed by processor i. The corresponding level d_i from below in which the processor is sitting in the binary tree, as well as parents and children will be determined. The length $count$ of the buffer Y and the pipelining blocksize $blockSize$ (which is also the length of the temporary buffer T) are passed as arguments. Note that this description only features the directives that apply to the part of buffer Y that can be segmented without remainder by $blockSize$. The actual implementation features a directive along this algorithm's line that deals with the remaining elements.

```
 1: determine $child_R, child_L, parent$
 2: determine tree depth $d$
 3: determine number of full sized blocks $chunk_{idx}$
 4: **if** root **then**
 5:     **for** $j = 0, blockSize, 2blockSize, \ldots, (chunk_{idx} - blockSize)$ **do**
 6:         Send($Y[j]$, child$_0$)                                  ▷ Left Child
 7:         Send($Y[j]$, child$_1$)                                  ▷ Right Child
 8: **else if** leaf **then**
 9:     **for** $j = 0, blockSize, 2blockSize, \ldots, (chunk_{idx} - blockSize)$ **do**
10:         Recv($Y[j]$, parent)                                     ▷ Parent
11: **else**
12:     **for** $j = 0, blockSize, 2blockSize, \ldots, (chunk_{idx} - blockSize)$ **do**
13:         Recv($Y[j]$, parent)                                     ▷ Parent
14:         Send($Y[j]$, $child_0$)                                  ▷ Left Child
15:         Send($Y[j]$, $child_1$)                                  ▷ Right Child
```

Now the Broadcast operation can start spreading the reduced blocks downwards and all the way to the leafs. As with the linear structure Broadcast this can be interpreted as an inverse Reduction without Reduction operations (this makes the temporary buffer obsolete). Again the root node is only performing $Send()$ (downwards), while the leaf nodes only receive data. The Reduction is implemented in the `treeduce()` function, the Broadcast in the `treecast()` function. Both algorithms are combined in the `allReduce4()` function.

3.2 Results and Discussion

Functionally the introduced algorithm pair works exactly as the linear structure one, however, it shall be mentioned that the reduced number of hierarchy levels causes less idle time for higher-up nodes during the Reduction and for closer-to-leaf nodes during the Broadcast. It is therefore reasonable to expect run-time improvements (compared to Exercise 2) to manifest themselves proportional to the tree depth. As such the results should at least in there tendencies align with the benchmark implementation. From Figure 6 in the Appendix A we can see that while the tendencies are indeed similar, the Naive Combination implementation from Exercise 1 is faster. This is due to a more refined model of algorithm choosing in the MPI implementation.

Additionally, as the Exercise 4 implementation does not intertwine Reduce and Broadcast steps of the algorithm. This implicates more idleness of nodes closer-to-leaf nodes during the Reduction (after the first block has been reduced in the root and would be available for downwards propagation) and lower efficiency of the higher-up nodes during the Broadcast (because they could start earlier and start propagating downwards) and overall more communication rounds compared to the reference. This is why our Allreduce implementation is slower by a factor more or less constant over $count$. This factor should, at least theoretically, be diminished in Exercise 5.

4 Exercise 5 (10 points)

In the fifth Exercise we implement the doubly-pipelined, reduction-to-all algorithm along the lines of Träff[Trä21], who suggests such an algorithm for a dual-rooted binary tree mapping. We start with an overview of the algorithm based on the Pseudo-Code, followed by comments on the adaptions we made to make it suitable to a non-dual-rooted binary tree structure. We further discuss how overlapping reduction and broadcast pipelines can achieve an improved number of communication rounds and use the pipelining-lemma to establish a running time estimate (again based on a round-based, linear transmission cost model). We close by discussing the implementation's results against the reference.

4.1 Doubly-pipelined, Reduction-to-all Algorithm

Algorithm E Doubly pipelined reduction-to-all algorithm as performed by processor i. The corresponding level d_i from below in which the processor is sitting in the binary tree, as well als parents and children will be determined. The length $count$ of the buffer Y and the pipelining blocksize $blockSize$ (which is also the length of the temporary buffer T) are passed as arguments. The symbol ZERO inside a Send() or Recv() operation indicates that a virtual block with **ONE** zero element is sent or received.

```
1:  determine node depth $d_i$, $d$
2:  determine number of blocks $b$
3:  determine $child_L, child_R, parent$
4:  initiate temporary buffer $T$
5:  for $j = 0, 1, \ldots, b + d$ do
6:      if not leaf then
7:          if $j - (d_i + 1) < 0$ or $j - (d_i + 1) \geq b$ then
8:              Send(ZERO, $child_0$) || Recv($T$, $child_0$)                          ▷ First child
9:              $Y[j * blockSize] \leftarrow T \odot Y[j * blockSize]$   ▷ In-order reduction (for Recv$\neq 0$)
10:             Send(ZERO, $child_1$) || Recv($T$, $child_1$)                          ▷ Second child
11:             $Y[j * blockSize] \leftarrow T \odot Y[j * blockSize]$   ▷ In-order reduction (for Recv$\neq 0$)
12:         else
13:             Send($Y[j - (d_i + 1) * blockSize]$, $child_0$) || Recv($T$, $child_0$)   ▷ First child
14:             $Y[j * blockSize] \leftarrow T \odot Y[j * blockSize]$   ▷ In-order reduction (for Recv$\neq 0$)
15:             Send($Y[j - (d_i + 1) * blockSize]$, $child_1$) || Recv($T$, $child_1$)   ▷ Second child
16:             $Y[j * blockSize] \leftarrow T \odot Y[j * blockSize]$   ▷ In-order reduction (for Recv$\neq 0$)
17:     if not root then
18:         if $j > b + d_i$ then
19:             Send(ZERO, parent) || Recv(ZERO, parent)                     ▷ Parent; 'do nothing'
20:         else
21:             if $j \geq b$ then
22:                 Send(ZERO, parent) || Recv($T$, parent)                              ▷ Parent
23:             else
24:                 Send($Y[j * blockSize]$, parent) || Recv(ZERO, parent)             ▷ Parent
25:             if $j - d_i \geq 0$ and $j - d_i < b$ then
26:                 $Y[(j - d_i) * blockSize] \leftarrow T \odot Y[(j - d_i) * blockSize]$ ▷ In-order reduction
27:
```

The idea of this algorithm builds upon the procedure of the tree-based algorithm introduced in 3. However, the core advancement in this algorithm lies in the overlapping of the reduction pipeline and the broadcast pipeline, resulting in a simultaneous bi-directional MPI_Sendrecv communication. Each process starts by computing the total depth d of the tree based on the $size$ parameter and further computes it's individual depth d_i in the tree structure. .

Each process runs through a total of $b + d$ rounds, where in each round j, a non-leaf process i first receives a block $Y[j * blockSize]$ from it's first child into it's temporary buffer T then sends back to this child an earlier block $Y[(j - (d_i + 1)) * blockSize]$ from it's receive buffer and finally issues an element-wise reduction of the received block in T with the according block in it's receive buffer $Y[j * blockSize]$ (lines 13 and 14). This procedure is immediately repeated with the second child process (lines 15 and 16). If the process is also non-root, then after reduction with the data from both children is completed, the process will send the reduced block up to it's parent process, from which it receives the already reduced block $Y[(j - d_i) * blockSize]$ and thus ends one iteration in the loop (lines 20 to 22). Processes that are leaves in the tree just send and receive blocks from their parent processes.
Since each process starts simultaneously and will run through all $b + d$ loop iterations and the number of `MPI_Send` and `MPI_Recv` operations have to match, there are special situations where certain processes will not send or receive meaningful data (yet). The first such situation is captured by the *out-of-bounds* condition (line 7), which takes care of the case when non-leaf nodes would try to access and send data outside of the range of Y, i.e. when either $j - (d_i + 1)$ is negative or exceeds the number of blocks. In these situations, the process will just send $ZERO$ messages, containing no data. However, it is possible, that actual data is already received, therefore the receive into the temporary buffer and the reduction step is conducted as above.
Another extraordinary situation occurs in the communication with the parent processes, when $j > b + d_i$, which is captured by the *do-nothing* condition (line 18), where a process sends and receives $ZERO$ from and to it's parent, which happens when no all data was already send to the parent and no useful data is received anymore. The last distinction to be made is when $j > b$, which is again a *out-of-bounds* situation (line 21) and stops tell the to process to stop sending data to it's parent and start sending $ZERO$. Again, useful data will be received into the temporary buffer. The other option is that the process already sends useful data to it's parent but still receives un-reduced data from it (line 24). Finally, if $0 \leq j - d_i < b$ (i.e. index is *within-bounds*), then the data is written into the receive buffer $Y[(j - d_i) * blockSize]$.

4.2 Requirements to the commutativity of the Reduction-Operator

Since the $MPI_{Allreduce}$ is based on tree structure mappings, we can obtain some inspiration from the according documentation [Mes21] in order to gain an idea of how our Doubly-pipelined Reduction-to-all algorithm. The standard indicates that it is unclear, in which order the nodes communicate with the root and therefore some requirements to the reduction operator have to be made:
"The "canonical" evaluation order of a reduction is determined by the ranks of the processes in the group. However, the implementation can take advantage of associativity, or associativity and commutativity in order to change the order of evaluation. This may change the result of the reduction for operations that are not strictly associative and commutative, such as floating point addition"

As opposed to linear communication structure based Allreduce implementations, it cannot be guaranteed thatelements are summed in the order of rank for non commutative operations.

4.3 Running time estimate[5]

In Corrollary 3 we were able to establish a running-time estimate for the Linear Pipelined Allreduce directive (where the Broadcast only starts once the Reduction is finished).

[5]This analysis only accounts for communication cost

$$\mathcal{O}\big(p + \sqrt{mp}\big) + 2 * 1\beta m$$

In a first step, we assemble the general differences between the case described above and the one in question. Most importantly the binary tree mapping leads to a reduced latency d of $d = \log_2(p)$, parameter s (a new block arrives at a non-root node every s rounds), however, increases since we need to establish connection to two children ($s = 2$). For the BinaryTree Pipelined Allreduce directive where the Broadcast only starts once the Reduction is finished (this correlates to Exercise 4), we would therefore expect the following running-time (shown in [**slides3**]):

$$\mathcal{O}\big(\log_2(p) + \sqrt{m\log_2(p)}\big) + 2 * 2\beta m.$$

It is evident that the Doubly Pipelined counterparts to these algorithms should facilitate decreased run-time by starting the reduction process at an earlier stage. First of all, algorithm E indicates that inner nodes receive a new block every 3 rounds (since every round features 3 $SendRecv()$-pairs). Therefore $s = 3$. The number of rounds until the the first block reaches the root (equivalent to the first reduced block reaching the last node) stays the same compared to the Singly Pipelined case, which is:

$$k = \text{ceil}\big\{\log_2\big((p+1)/4\big)\big\}$$

Proof. See [**slides3**] on slide 349-363. ■

Hence, we can state a run-time estimation for the Doubly Pipelined case without as:

$$\mathcal{O}\big(\log_2(p) + \sqrt{m\log_2(p)}\big) + 1 * 3\beta m.$$

This also aligns with the estimation presented in the paper [Trä21].

4.4 Results and Discussion

At first glance, this algorithm falls short of expectations, since no performance gain as compared to algorithm from Exercie 4 could be achieved. At the lower end of number of elements, the algorithm indeed is slower than the standard pipelined tree algorithm for most $node \times process$ configurations. For all problem sizes, the worst performance was achieved for high $node \times process$ counts, whereas configurations of $1 \times processes$ and $nodes \times 1$ performed better by almost up to an order (see 36×32 vs. 20×1 line) being on a par with the reference implementation. On the one hand, this contradicts the expectation, that tree-based algorithms will benefit from a high total node count, however, on the other hand we also expected an overhead caused by too many `MPI_Sends` and `MPI_Recvs` issued. This is, of course, even more pronounced for a high total node count. Additionally, we state here, that the code base for the algorithms in Exercise 4 and Exercise 5 differ significantly, as these algorithms were developed independently from another. Especially, the computation of the individual height in the tree d_i by each process introduces an overhead in Exercise 5. Of course, this has an effect on the performance of this algorithm and could explain to some extent, why we do not see a significant performance increase.

5 Exercise 3 (10 points)

We realized the integrated, pipelined, Allreduce-implementation required for Exercise 3 as modification of the doubly-pipelined, reduction-to-all algorithm from the Exercise 5: By reducing the binary (2-ary) tree structure to a 1-ary structure we obtain a 'more tightly combined' == doubly pipelined linear Allreduce-implementation. We start with an overview of the algorithm and discuss how the algorithm differentiates from the one presented in Exercise 5. We close by answering whether this implementation can achieve better results against our benchmark then it's less optimized counterpart from Exercise 2.

5.1 Linear Doubly-Pipelined Allreduce Algorithm and Implementation

Algorithm F Doubly pipelined reduction-to-all algorithm as performed by processor i. The corresponding level d_i from below in which the processor is sitting in the binary tree, as well als parents and children will be determined. The length $count$ of the buffer Y and the pipelining blocksize $blockSize$ (which is also the length of the temporary buffer T) are passed as arguments. The symbol ZERO inside a Send() or Recv() operation indicates that a virtual block with **ONE** zero element is sent or received.

```
1: determine node depth $d_i$, $d$
2: determine number of blocks $b$
3: determine $child, parent$
4: initiate temporary buffer $T$
5: for $j = 0, 1, \dots, b + d$ do
6:     if not leaf then
7:         if $j - (d_i + 1) < 0$ or $j - (d_i + 1) \geq b$ then
8:             Send(ZERO, child) || Recv($T$, child$_0$)                          ▷ Child
9:             $Y\big[j * blockSize\big] \leftarrow T \odot Y\big[j * blockSize\big]$   ▷ In-order reduction (for Recv$\neq 0$)
10:        else
11:            Send($Y\big[j - (d_i + 1) * blockSize\big]$, child) || Recv($T$, child$_1$)   ▷ Child
12:            $Y\big[j * blockSize\big] \leftarrow T \odot Y\big[j * blockSize\big]$   ▷ In-order reduction (for Recv$\neq 0$)
13:    if not root then
14:        if $j > b + d_i$ then
15:            Send(ZERO, parent) || Recv(ZERO, parent)              ▷ Parent; 'do nothing'
16:        else
17:            if $j \geq b$ then
18:                Send(ZERO, parent) || Recv($T$, parent)                      ▷ Parent
19:            else
20:                Send($Y\big[j * blockSize\big]$, parent) || Recv(ZERO, parent)   ▷ Parent
21:            if $j - d_i \geq 0$ and $j - d_i < b$ then
22:                $Y\big[(j - d_i) * blockSize\big] \leftarrow T \odot Y\big[(j - d_i) * blockSize\big]$ ▷ In-order reduction
23:
```

As was already stated above, this algorithm works in the exact same way as the one of Exercise 5 does. The only difference is the absence of each second child in the tree, so each process has at maximum one child and one parent (exceptions being the root and leaf processes, with no parent and no child respectively). Other than that, we refer to Section 4 for a detailed explanation of the algorithm.

5.2 Results and Discussion

As was already the case in Exercise 5, the performance of this algorithm did not increase with respect to it's non-overlapping pipelined counterpart in Exercise 2, which is in contrast to our expectations. Again we claim, that the difference in the implementation details make a direct comparison harder, although in this case, the argument of finding the depth in the tree vanishes as the depth in the tree here simple is the rank of the process. Overall, the algorithm shows similar performance as in Exercise 2 for the lower end of counts of elements (up to $1 \cdot 10^4$). For larger counts, the following observation can be made: For $node \times process$ configurations that are either heavy on the node-side or on the process-side (i.e. $1 \times processes$ or $nodes \times 1$), the performance is compareable to the one of Exercise 2. But for large total amounts of $node \times process$, the performance worsens drastically, especially for the cases 20×32 and 36×16, where the performance is roughly an order worse. This is a similar picture as we highlighted in the discussion of Exercise 5 and it could be argued that too large an amount of nodes results in disadvantageous amounts of communication. The MPI reference is better in most cases, with some overlap in the best cases of our implementation.

A Appendix

NOTE 1 (regarding plots corresponding to Exercise 1, 3, 4 and 5):
As described in the Experimental Setup (table 2) we ran these exercises for multiple *blockSize* configurations. As discussed in Exercise 2, the most performant *blockSize*-configuration is a heuristic and as such should be optimized for the array length *count* as well as different communication structure configurations (number of nodes and processes per node). As such the runtime-value for each communication structure configuration and count in the concerning blocks corresponds to the *blockSize* with the **lowest** runtime-median.

NOTE 2 (regarding graph linestyle):
In all plots, **dotted** lines correspond to the `MPI_Allreduce`-reference and **solid** lines represent our custom implementations.

NOTE 3 (regarding raw data and extended statistics):
Data on which all plots of this section are based can be found in the project-associated submission folder (for details on the latter's structure, see README.txt). In order not to compromise on readability, we refrained from plotting extended statisics (such as std-dev and confidence intervals)

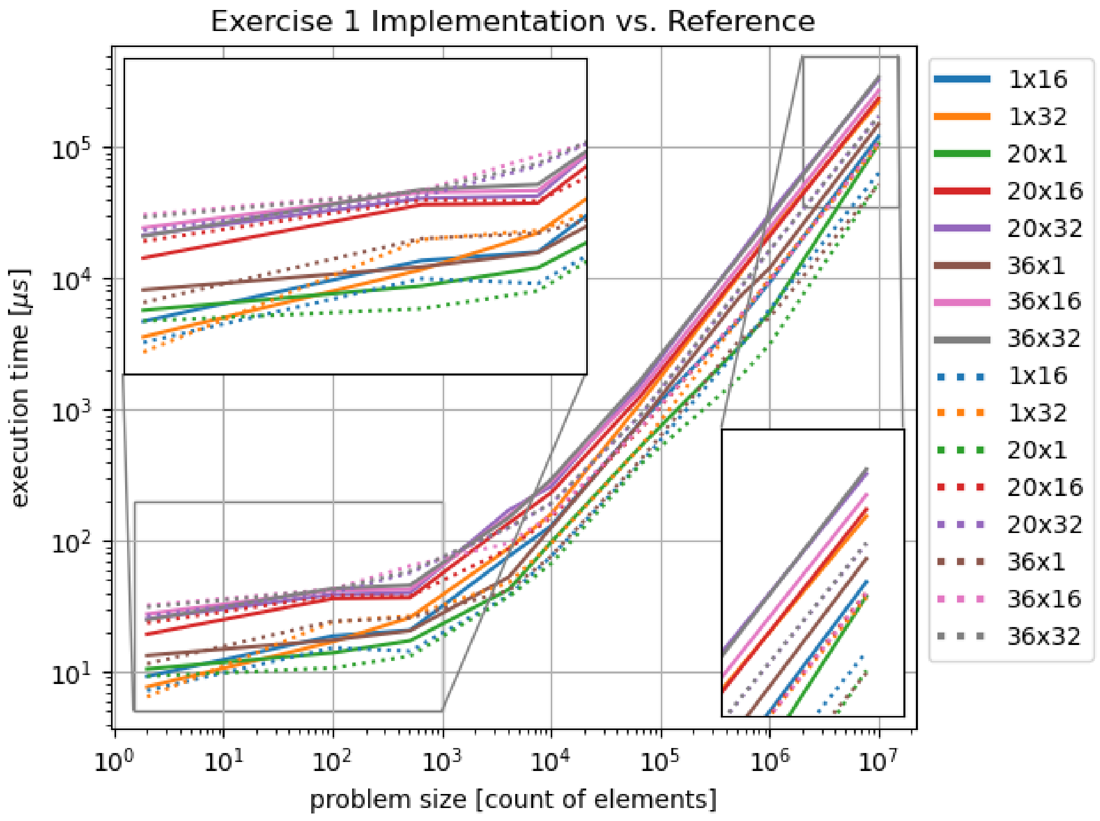


Figure 2: Comparison of execution time of `MPI_Reduce` + `MPI_Bcast` implementation vs. `MPI_Allreduce`

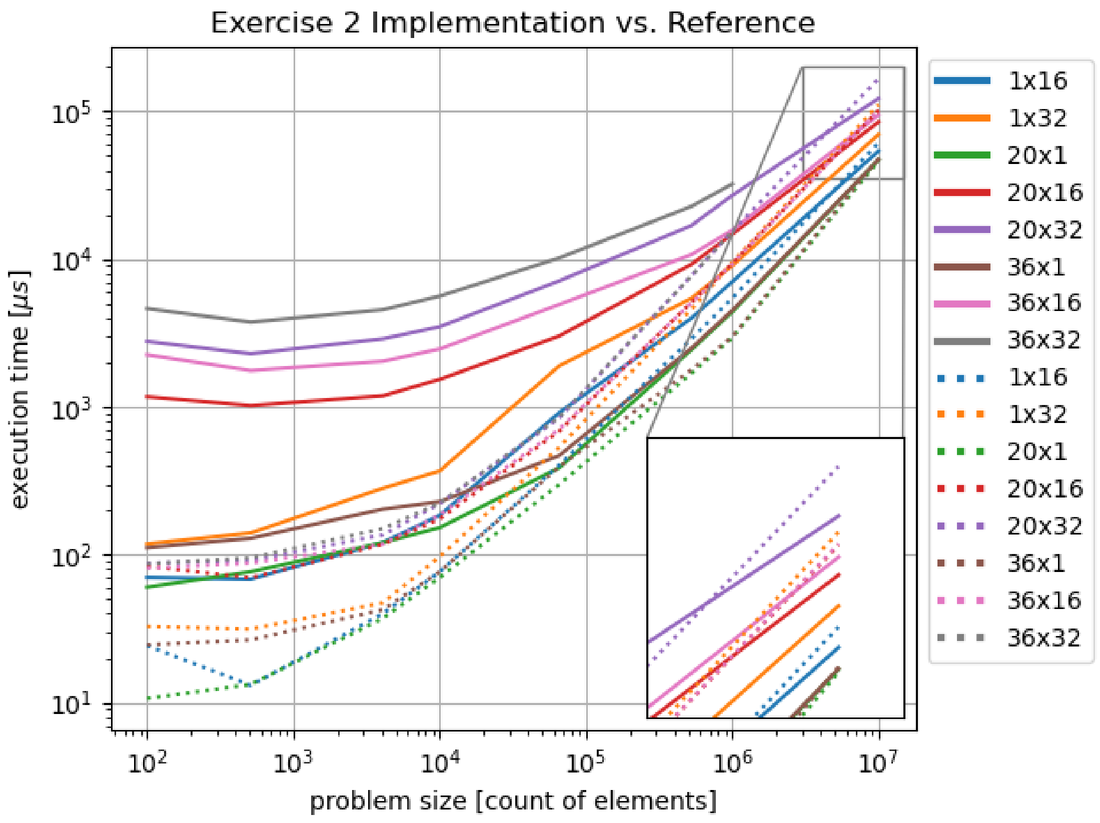


Figure 3: Comparison of execution time of Exercise 2 implementation vs. `MPI_Allreduce`

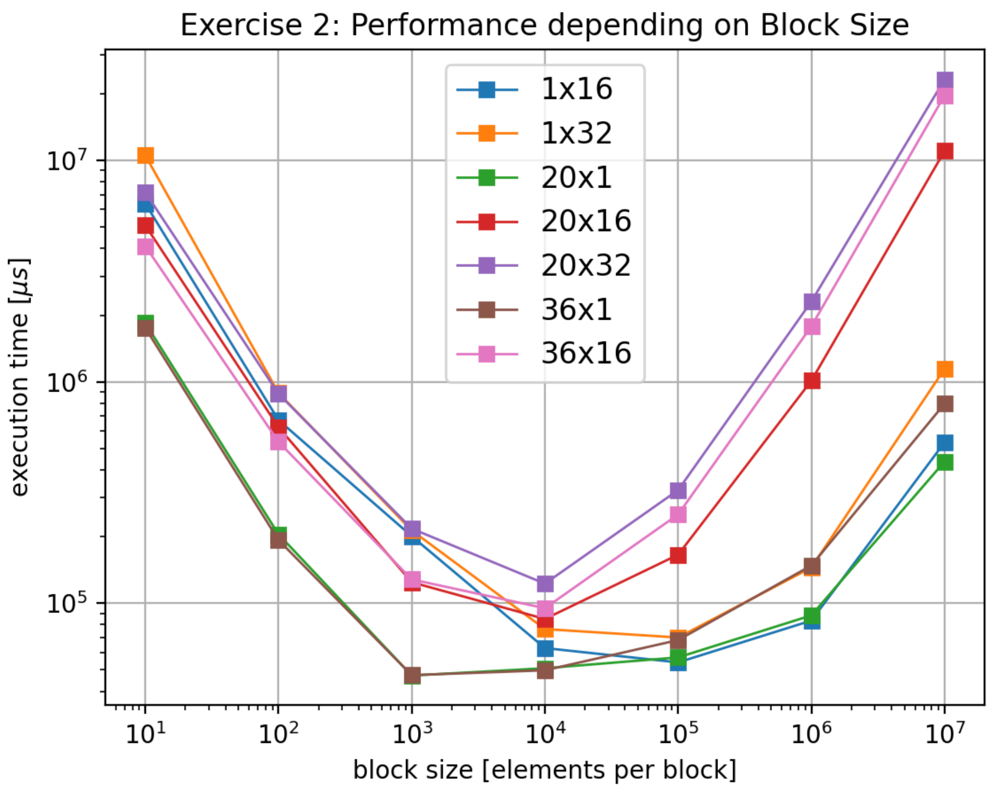


Figure 4: Effect of block size on performance in Exercise 2 implementation for array length $1e7$

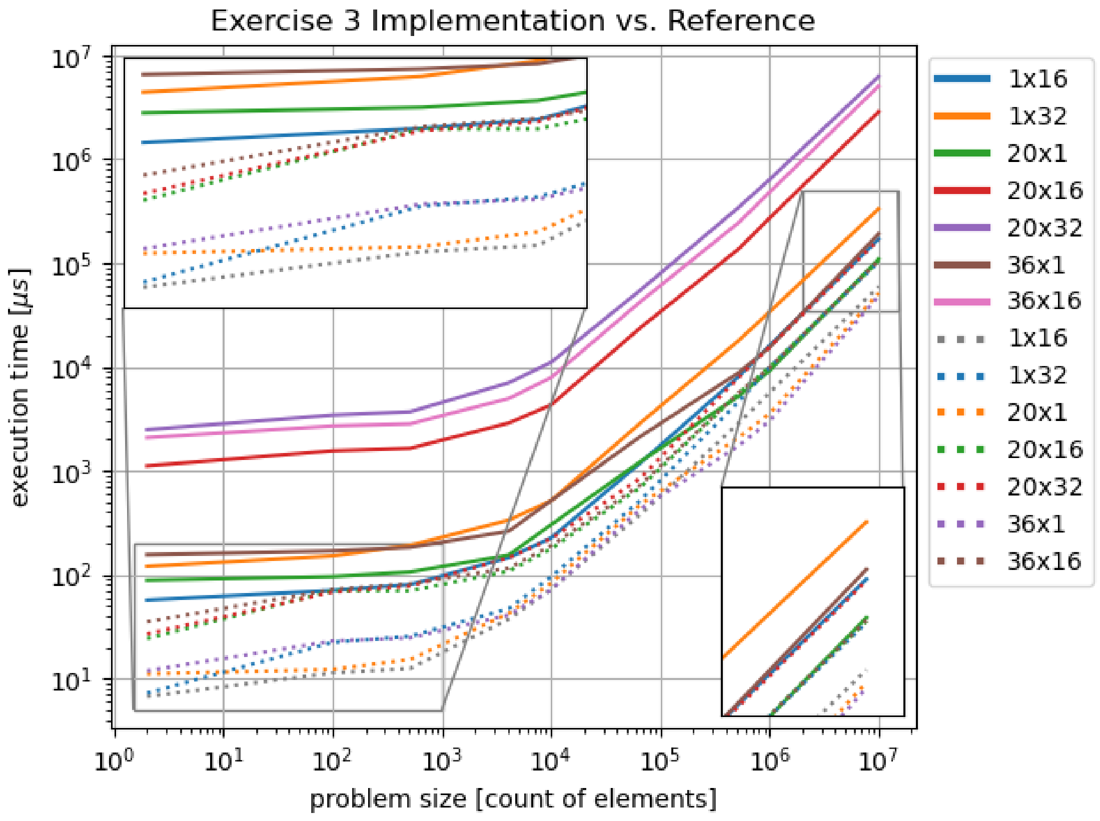


Figure 5: Comparison of execution time of Exercise 3 implementation vs. `MPI_Allreduce`

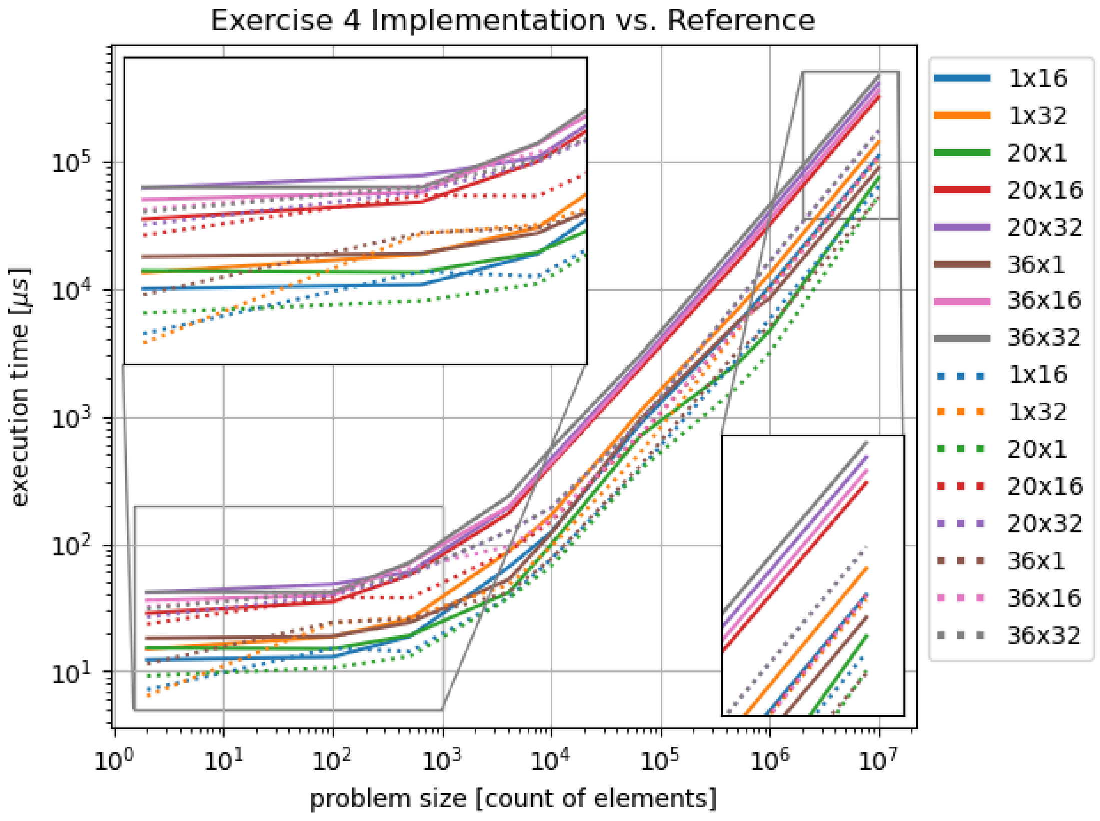


Figure 6: Comparison of execution time of Exercise 4 implementation vs. `MPI_Allreduce`

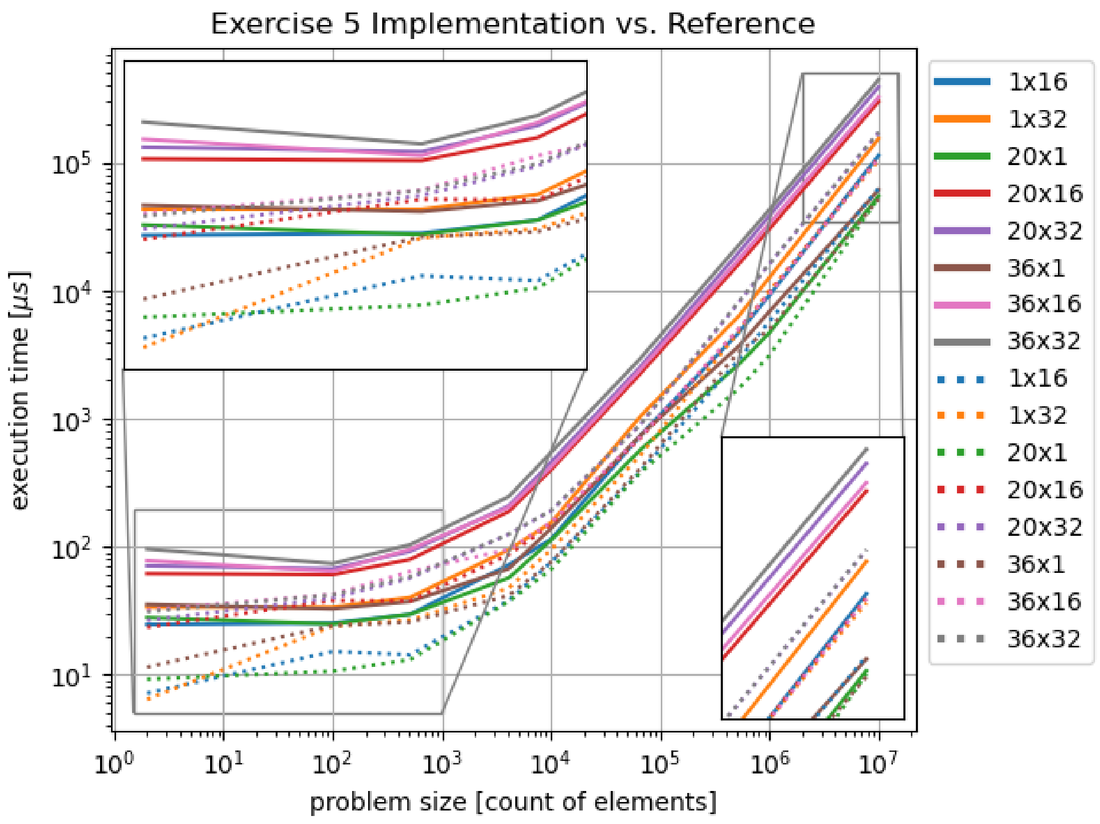


Figure 7: Comparison of execution time of Exercise 5 implementation vs. `MPI_Allreduce`