

VIENNA UNIVERSITY OF TECHNOLOGY

184.725 HIGH PERFORMANCE COMPUTING

TU WIEN INFORMATICS

---

## Exercise 2

---

*Authors:*

Camilo TELLO FACHIN

12127084

Manuela Maria RAIDL

01427517

*Supervisor:*

Prof. Dr. Jesper Larsson TRÄFF

January 16, 2023



TECHNISCHE  
UNIVERSITÄT  
WIEN

Vienna University of Technology

## Contents

<b>1</b>	<b>Exercise 1 - Implement A Benchmark Framework</b>	<b>1</b>
<b>2</b>	<b>Exercise 2 - Implement Linear Pipeline for <code>MPI_Bcast</code> and <code>MPI_Reduce</code></b>	<b>3</b>
<b>3</b>	<b>Exercise 3 - Combining <code>MPI</code> Processes</b>	<b>4</b>
<b>4</b>	<b>Exercise 4 - Binary Tree Algorithms for <code>MPI_Bcast</code> and <code>MPI_Reduce</code></b>	<b>6</b>
<b>5</b>	<b>Exercise 5 - Integrated, Improved Binary Tree Algorithm</b>	<b>7</b>
<b>6</b>	<b>Exercise 6 - Improvement with Sibling Leave Communication (BONUS)</b>	<b>8</b>
<b>7</b>	<b>Exercise 7 - Implemenetation and Benchmarking of Improved Version (BONUS)</b>	<b>8</b>

## 1 Exercise 1 - Implement A Benchmark Framework

The main part of task 1 was to implement a benchmarking framework for the following exercises. This was done successfully and all the demanded quantities were computed and gathered in `.txt` files named with their respective configurations and exercise number. This to postprocess and plot the data within python with low effort. The benchmarking framework is used in exercise 1 with a naive implementation of gathering information of all processes, perform an operation on the gathered information, and distribute the result of the operation to all participating processes. In our case we used the `MPI_MAX` operation, which yields the largest element of the buffers. In the following exercises, the same idea is pursued (reduce, perform operation, broadcast operation result) but with more sophisticated message passing approaches.

For all the following exercises we used bash scripts with for loops, for every exercise one bash script that runs all the different configurations. Before compiling the C files with `mpicc .. -lm -O3` one had to first run `module load mpi/openmpiS` on hydra.

For the naive implementation `MY_Allreduce()`, which is essentially `MPI_Reduce()` followed by `MPI_Bcast()`, for powers of 10, we observed the timings shown in figure (1). The configurations for each graph are shown in the legend, where N is the number of Hydra Nodes, T the number of Processes per Node and P is e.g. powers of 10, whereby powers of 10 means that the length of the buffers (Count) are all powers of 10. The measurements for 36 hydra nodes and 32 tasks per node (total of 1152 MPI Processes) were the slowest, while the measurement for 20 nodes and 1 task per node performed best.

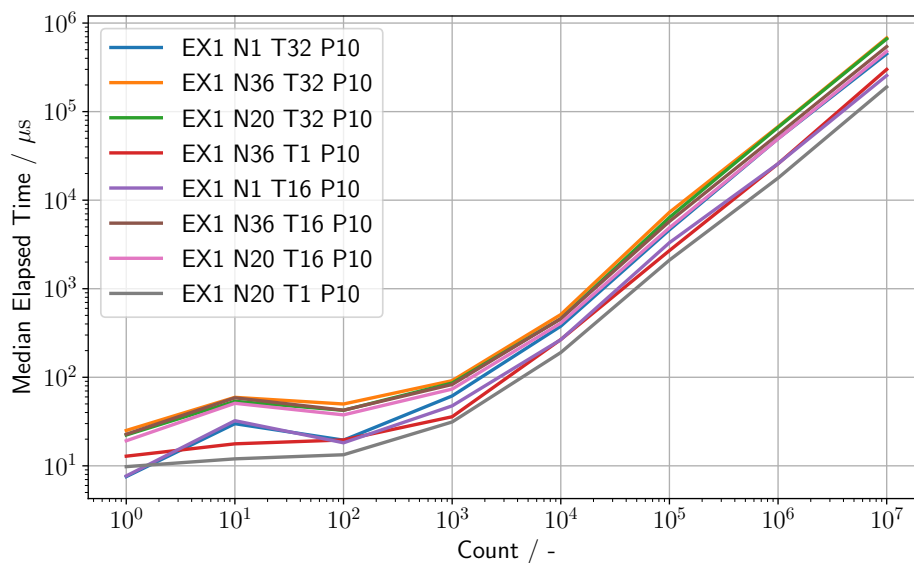


Figure 1: Median Timings for `MPI_Reduce + MPI_Bcast` for all Configurations and powers of 10. Since every MPI process performs its own timing, we use `MPI_Reduce(..., MPI_MAX)` to retrieve the timing of the slowest process. After 10 Warmup rounds in Ex1 we ran a 100 iterations to compute the statistical data asked for in Ex1.

Similar behaviour of the measurements can be observed for the ones with power of 2, see figure (2). Where the total highest number of MPI processes again performed the worst and the one with 20 nodes and 1 task per node did the best. Also visible are jumps in all measurements except two at `count = 32`, the two measurements where the jumps don't occur are the ones with only one task per node. This exact characteristics were also observed in the plot for powers of 10, just at `count = 10`.

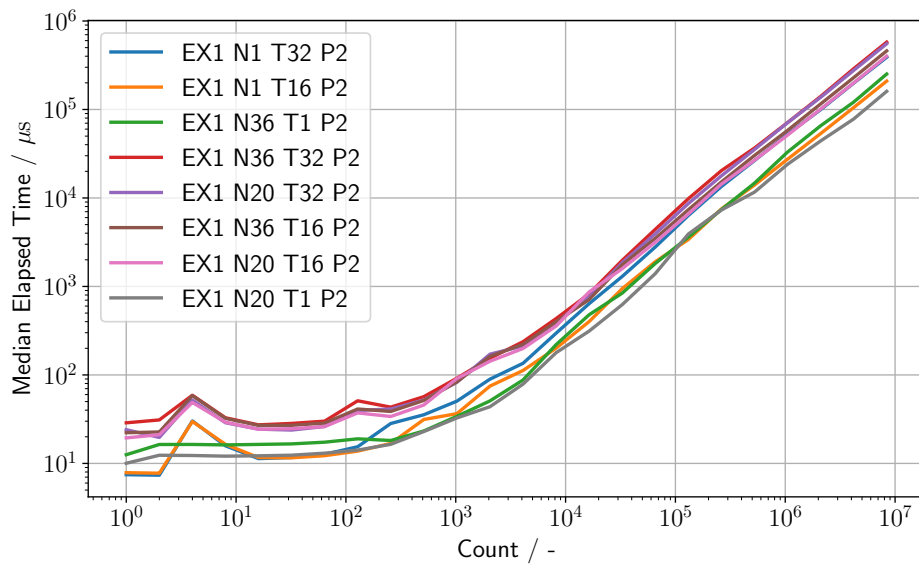


Figure 2: Median Timings for MPI\_Reduce + MPI\_Bcast with all Configurations and powers of 2

In figure (3) we show the best and worst configurations for both powers in the same plot.

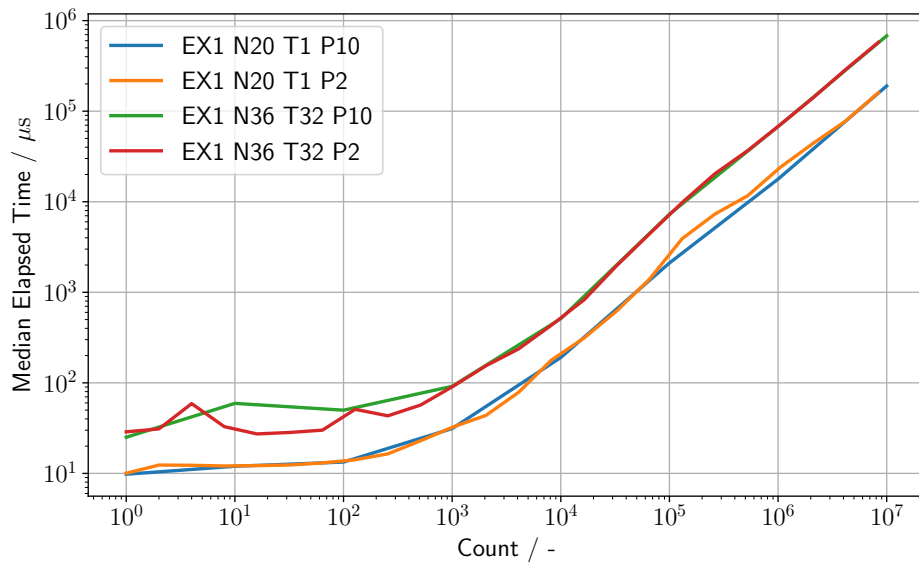


Figure 3: Median Timings for MPI\_Reduce + MPI\_Bcast, best and worst configurations

The timings shown in all figures in the document are all medians, which is an output of the statistical data postprocessing done in each C file to produce all the quantities asked for in this exercise. They contain the average, median, best seen time, standard deviation and confidence intervals.

## 2 Exercise 2 - Implement Linear Pipeline for MPI\_Bcast and MPI\_Reduce

SLIDES ZITIERN, PAPER TRAEFF ZITIERN, REFERENZ ZU NER ANDEREN CHAPTER MACHEN

The goal of exercise 2 is to implement linear pipelined versions of `MPI_Bcast()` and `MPI_Reduce()` based on the algorithms discussed in the lecture (refer to Algorithm 16 and Algorithm 17 of "Algorithms for Collective Communication").

Our implementation for the pipelined reduction – `MY_Reduce_P()` – aligns with the following idea: We distinct between a master process ( $\text{rank} = 0$ ), interior processes ( $0 < \text{rank} < \text{size} - 1$ ) and an end process ( $\text{rank} = \text{size} - 1$ ). As an output of `MY_Reduce_P()`, the master shall possess the entry-wise maximum as reduction result. Additionally, we divide the array that needs to be compared into multiple blocks of size `blockSize` (and probably a smaller block in the end). Goal is to communicate between the processes for each block separately.

The end process ( $\text{rank} = \text{size} - 1$ ) sends blocks – one after the other – to the interior process with  $\text{rank} = \text{size} - 2$ . The end process does not receive data, as there are no further processes. Interior processes first receive a data block from their neighbor with  $\text{rank} + 1$ , perform a local reduction and send this data to their other neighbor with  $\text{rank} - 1$ . This is repeated for all blocks. The master process receives block data from the interior node of  $\text{rank} = 1$ .

For performance estimation of `MY_Reduce_P()`, we take a look on the number of communication rounds. Let  $b$  be the total number of blocks. Then, there will be a need of  $b$  communication rounds in `MY_Reduce_P()`.

Our implementation for the pipelined broadcast – `MY_Bcast_P()` – is quite similar to the `MY_Reduce_P()` implementation. For broadcast the goal is that the data which is I the beginning available for the master process shall be communicated to all the other processes. In order to do so, the master process sends its data block wise to its neighbor with  $\text{rank} = 1$  (interior process). Interior processes receive block data from their predecessor process and immediately communicate this data to their successor process – block by block. The last one to receive the data is the end processor. There is no need to send any further data from here.

Very similar as for `MY_Reduce_P()` we can see from the implementation, that for `MY_Bcast_P()` there are  $b$  rounds of communication needed as well.

The trivial combination of `MY_Reduce_P()` and `MY_Bcast_P()` can be seen as a pipelined variant of `MPI_Allreduce()`. We end up with a total of  $2b$  communication rounds for this realisation of `MPI_Allreduce()`.

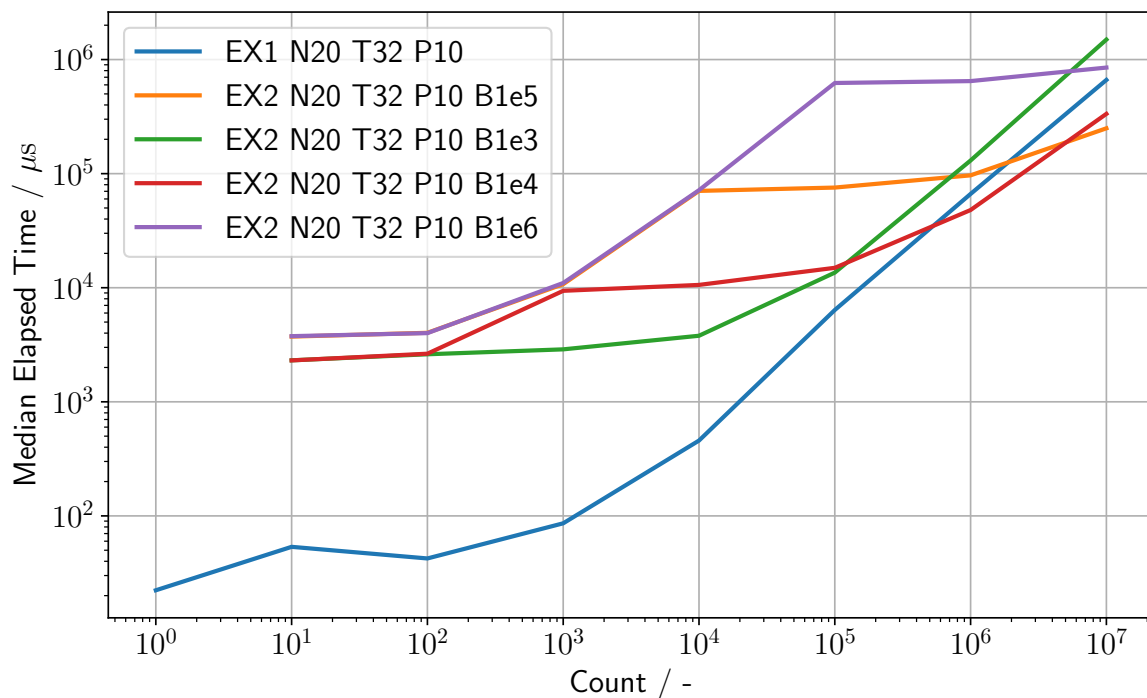


Figure 4: Caption for Ex2 plot 1

### 3 Exercise 3 - Combining MPI Processes

#### PROPER ZITAT ZU AUFGABE 5

With the goal of saving slightly on communication rounds by combining a pipelined reduction and a pipelined broadcast more tightly to keep the processes “more busy”, we end up with the function implementation of `MY_Allreduce_P()`. This implementation is completely based on the pipelined tree implementation for exercise 5 (REFER HERE TO EX5). For better understanding, refer to exercise 5 first.

Hence, we interpret the “lined up” processes as a tree, where each node except one leaf has exactly one child and all nodes except the root have a parent node. With this understanding, we reuse the implementation of `MY_Allreduce_T()` from exercise 5 by getting rid of the communication between a “right child” and its parent as in our setting for exercise 3 only “left children” exist.

We can expect an improvement compared to the trivial combination of `MY_Reduce_P()` and `MY_Bcast_P()`, as in `MY_Allreduce_P()` the reduction already gets started as soon as the data of the first block received the master process – the root node. Therefore, number of processes/nodes – 1 + number of blocks are needed in `MY_Allreduce_P()`, whereas `MY_Reduce_P()` and `MY_Bcast_P()` use number of blocks communication rounds each. For a small numbers of processes/nodes, the number of communication rounds can be reduced based on the number of blocks and therefore the blocksize.

For performance estimation let us consider the number of communication rounds between processes in our pipelined `MY_Allreduce_P()`. Again let  $b$  be the total number of blocks and  $n$  the number of processes. Then, there is the need for a total of  $b + n - 1$  rounds of communication. As we assume  $n$  to be smaller than the numbers of blocks, we end up with less communication rounds compared to `MY_Reduce_P()` and

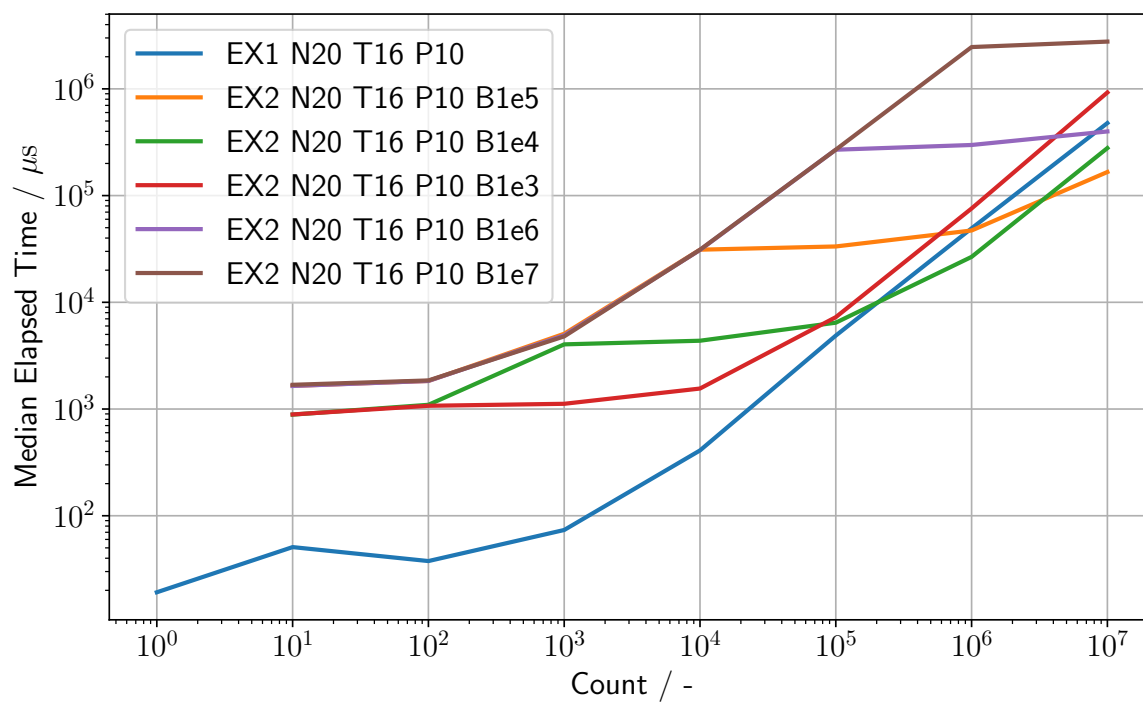


Figure 5: Caption for Ex2 plot 2

MY\_Bcast\_P() from exercise 2.

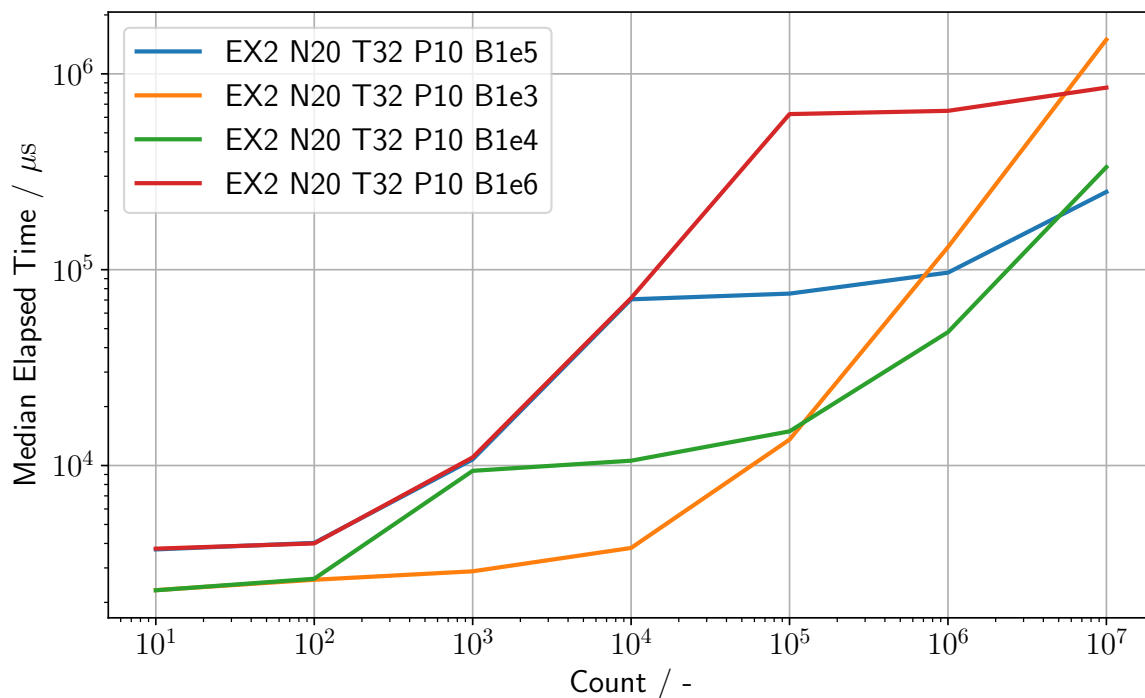


Figure 6: Caption for Ex2 plot 3

## 4 Exercise 4 - Binary Tree Algorithms for MPI\_Bcast and MPI\_Reduce

Instead of “lined up” processes we now want to use a binary tree structure and according algorithms `MY_Reduce_T()` and `MY_Bcast_T()` for reduction and broadcasting. As we understand each process as a node, we will use the wording node from now on. For indexing of the nodes, we use preorder traversal. We start very similar to the implementations of `MY_Reduce_P()` and `MY_Bcast_P()` from exercise 2. For the reduction `MY_Reduce_P()` the root node (rank = 0) on level 0 should gain the reduced result. Hence, the leaves start by sending the data block by block to their parents. All interior nodes receive exactly two data blocks per communication round. One from their left and one from their right child. After performing a local reduction, the data is sent to the nodes parent. In the end, the root node receives the data from its two children and ends up with the reduction result. For the broadcast `MY_Bcast_P()`, the root node sends the data block wise to its two children. A child receives the data and immediately forwards this data to its children, in case it is not a leaf.

The trivial combination of `MY_Reduce_T()` and `MY_Bcast_T()` can be seen as a pipelined variant of `MPI_Allreduce()`.

For performance estimation – again – let us consider the number of communication rounds between processes in the just discussed implementations of `MY_Reduce_T()` and `MY_Bcast_T()`. Again let  $b$  be the total number of blocks, then there is the need for  $b$  MPI communication rounds per function call. Therefore, we end up with a total of  $2b$  rounds of communication for the combination of `MY_Reduce_T()` and `MY_Bcast_T()`.



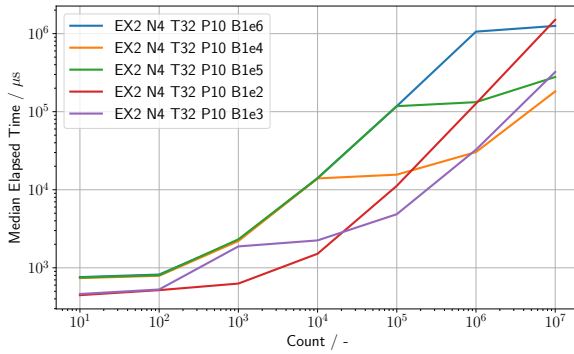


Figure 7: Caption for Ex2 plot 4

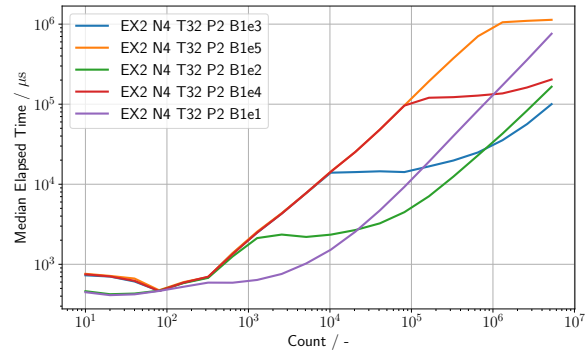


Figure 8: Caption for Ex2 plot 5

## 5 Exercise 5 - Integrated, Improved Binary Tree Algorithm

PROPER ZITAT VON TRAEFF

We aim to devise an integrated, improved binary tree algorithm for `MPI_Allreduce()`. Our implementation `MY_Allreduce_T()` is based on the algorithm for a doubly pipelined binary tree described in [Jesper Larsson Träff. A doubly-pipelined, dual-root reduction-to-all algorithm and implementation. arXiv:2109.12626, 2021.]. Note that we use only one tree, so there will never happen any communication between roots of different trees, as there is no. Per communication round, the reduction of one block can be performed. Hence, for complete reduction as much communication rounds are needed, as there are blocks. The idea is, that the root node starts with its broadcast-like send operations as soon as it received the data of the first block. The root node performs a local reduction against its own value and then the broadcasting starts while other blocks are still (or not yet) in their reduction phase.

For performance estimation let us consider the number of communication rounds one last time. As before, let  $b$  be the total number of blocks and (different to before)  $n$  the number of processes. Then, there is the need for a total of  $b + \lfloor \log_2(n) \rfloor$  rounds of communication. The number  $\lfloor \log_2(n) \rfloor$  can be understood as the maximum level of our binary tree. As we assume  $n$  to be smaller than the numbers of blocks, we end up with less communication rounds compared to `MY_Reduce_T()` and `MY_Bcast_T()` from exercise 4.

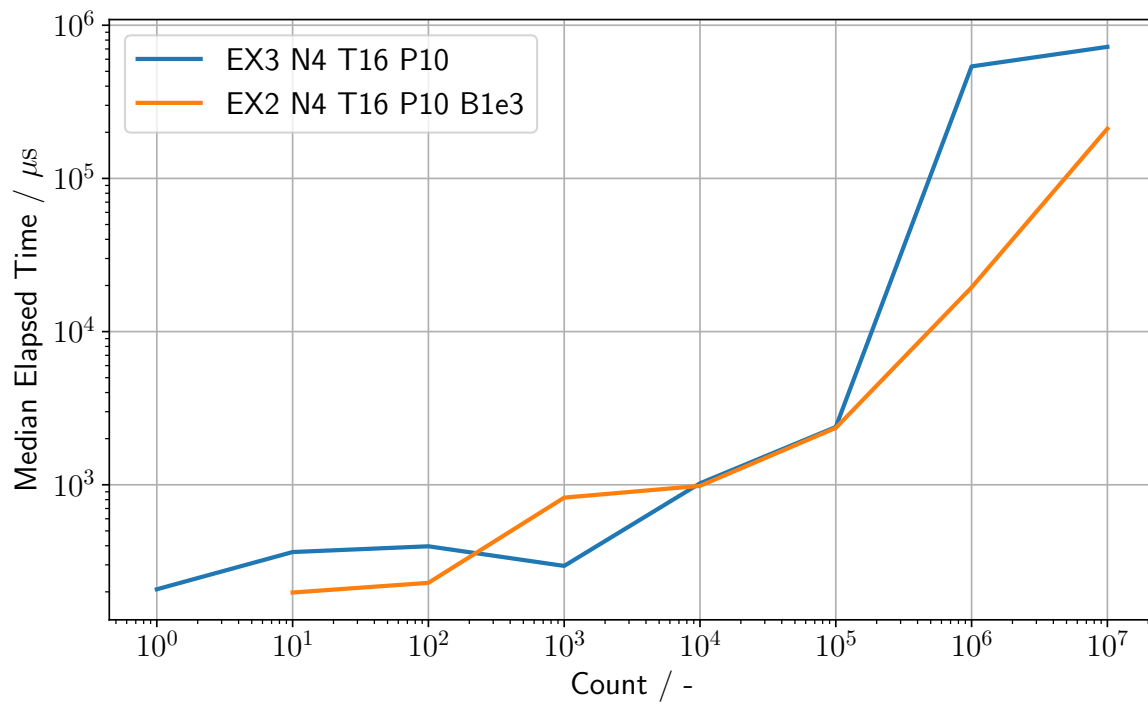


Figure 9: Caption for Ex3 plot 1

## 6 Exercise 6 - Improvement with Sibling Leave Communication (BONUS)

Not implemented.

## 7 Exercise 7 - Implementation and Benchmarking of Improved Version (BONUS)

Not implemented.

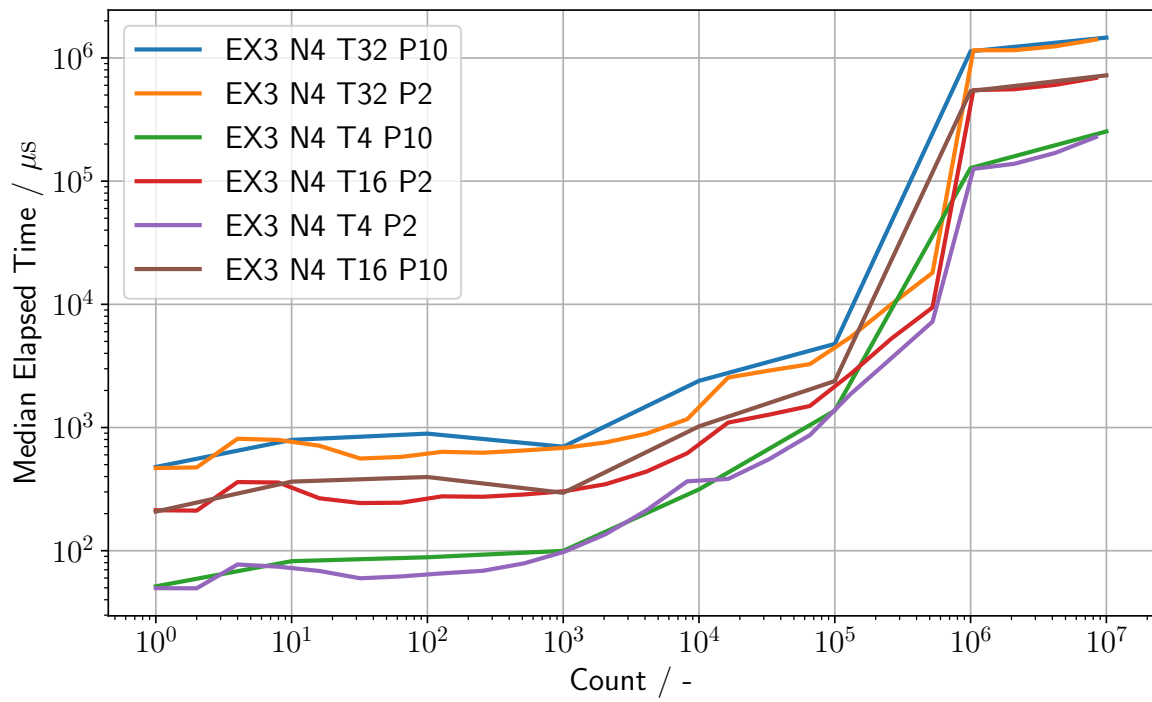


Figure 10: Caption for Ex3 plot 2

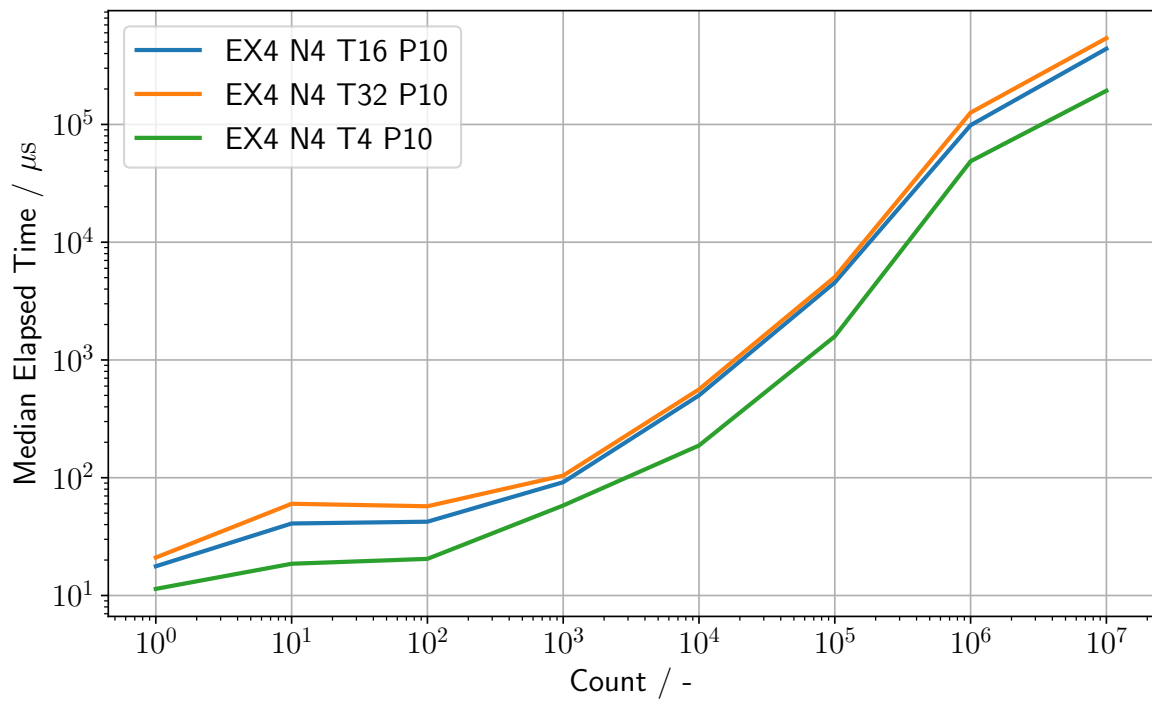


Figure 11: Caption for Ex3 plot 3

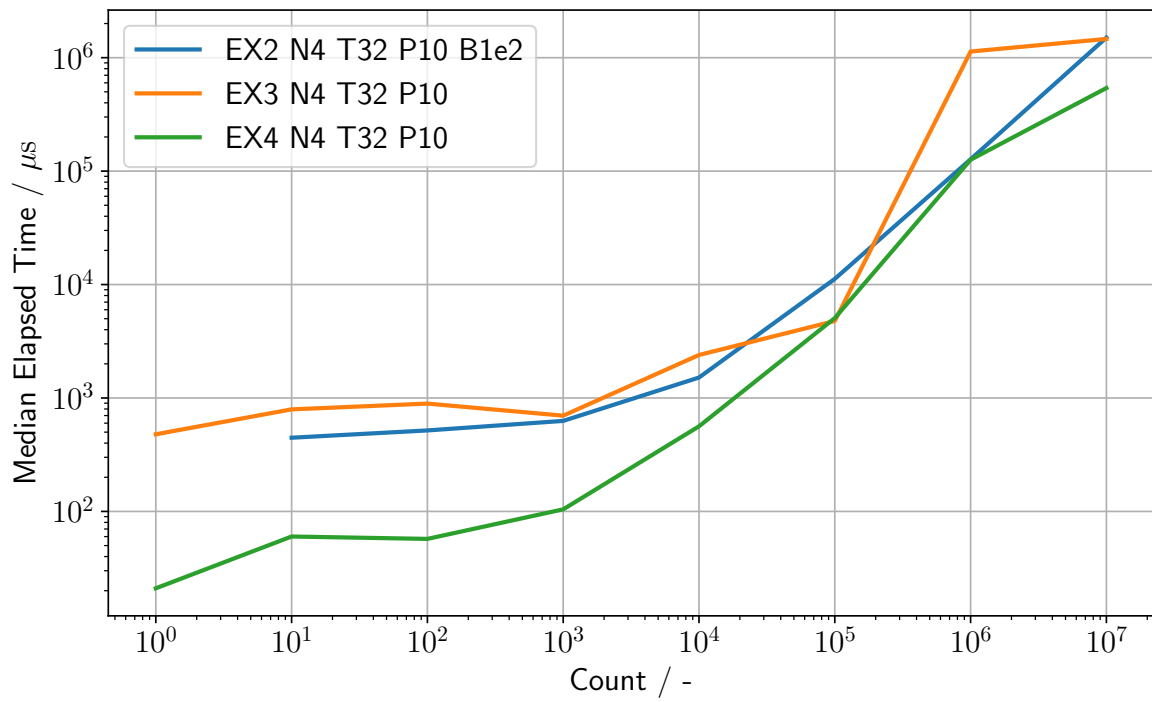


Figure 12: Caption for Ex4 plot 1

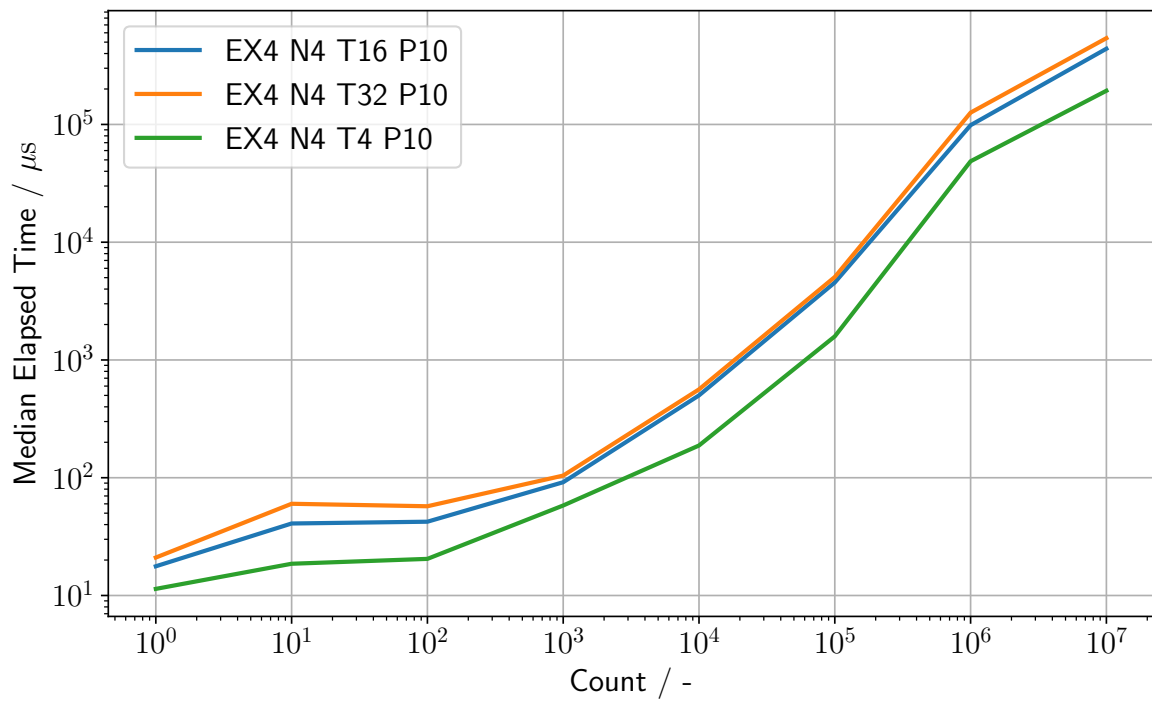


Figure 13: Caption for Ex4 plot 2

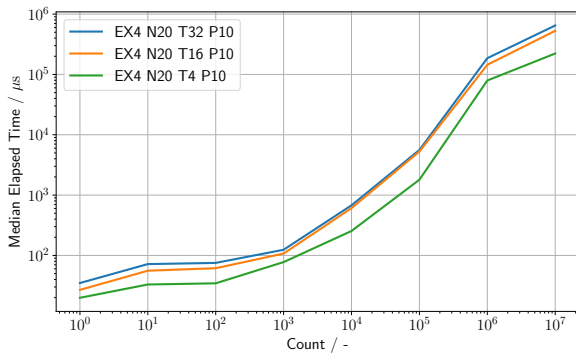


Figure 14: Caption for Ex2 plot 3

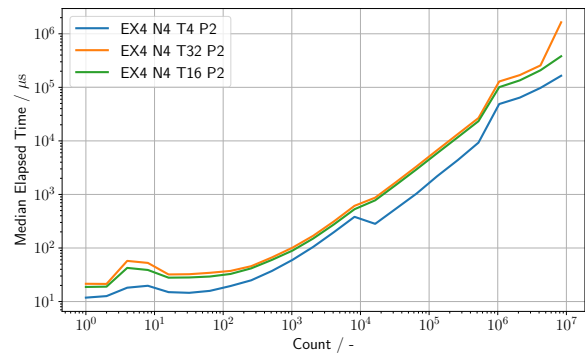


Figure 15: Caption for Ex4 plot 4

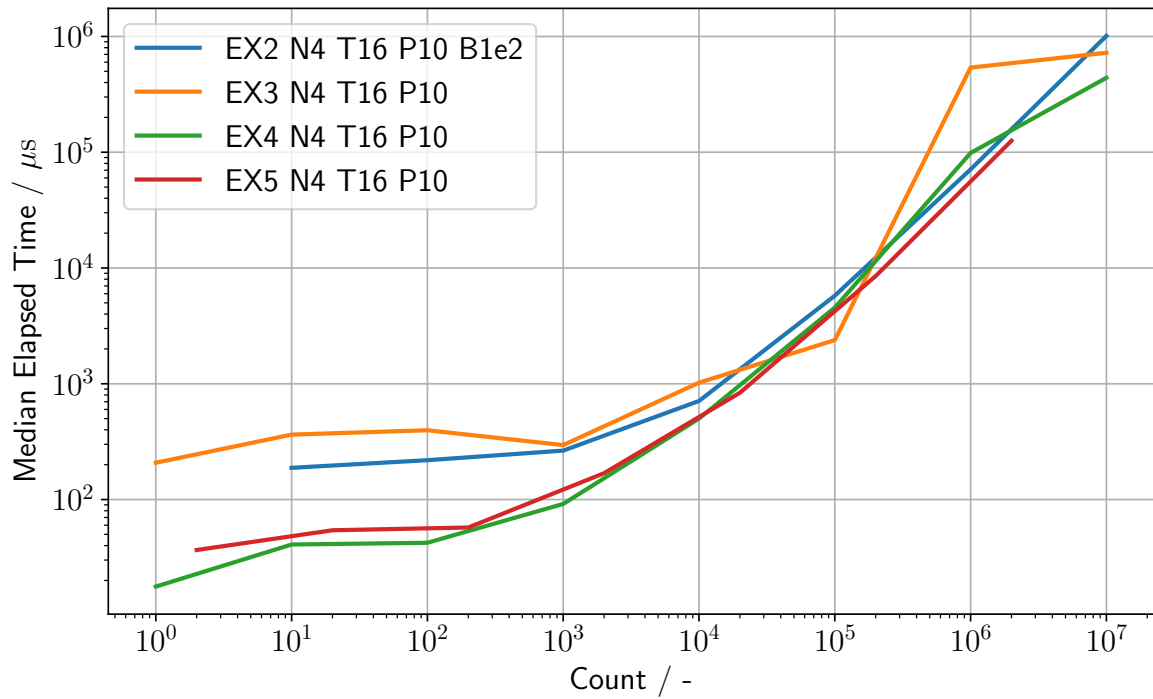


Figure 16: Caption for Ex5 plot 3

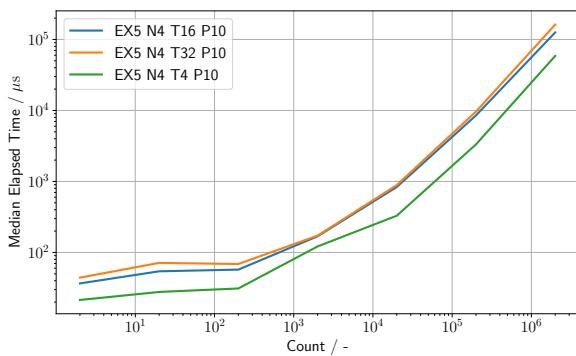


Figure 17: Caption for Ex5 plot 1

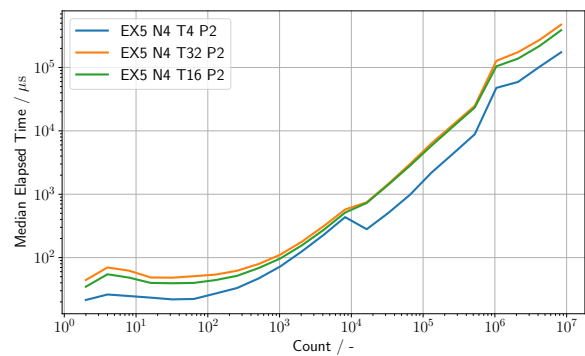


Figure 18: Caption for Ex5 plot 2