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

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 succesfully and all the demanded quantities where 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 persued (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.

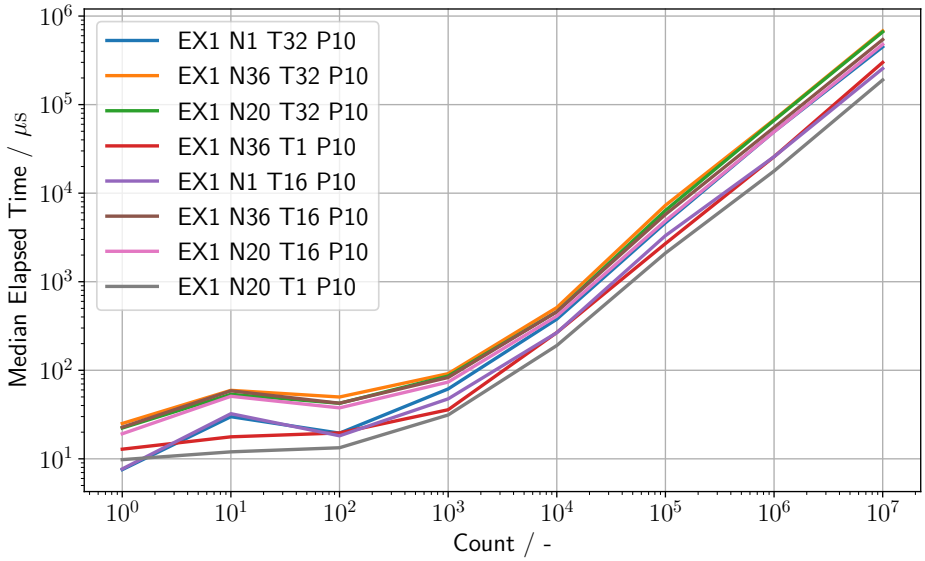

Figure 1: Median Timings for `MPI_Reduce + MPI_Bcast` for all Configurations and powers of 10. Since every `MPI` process performs its own timing, we use `MPI_Reduce(..,MPI_MAX)` to retrieve the timing of the slowest process. After 10 Warmup rounds in Ex1 we ran a 100 iterations to compute the statistical 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 dont occur are the ones with only one task per node. This exact characteristics where also observed in the plot for powers of 10, just at count = 10

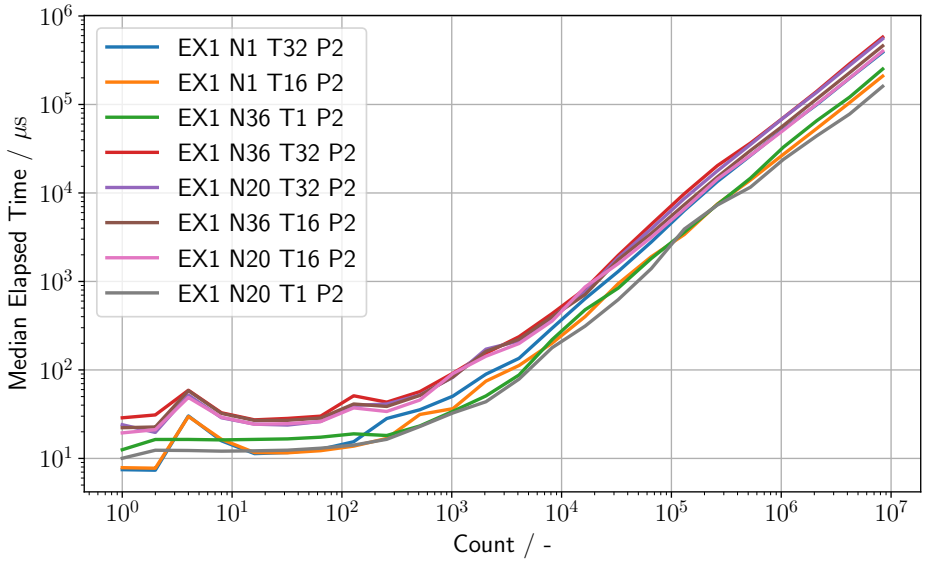

Figure 2: Median Timings for `MPI_Reduce + MPI_Bcast` with all Configurations and powers of 2

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

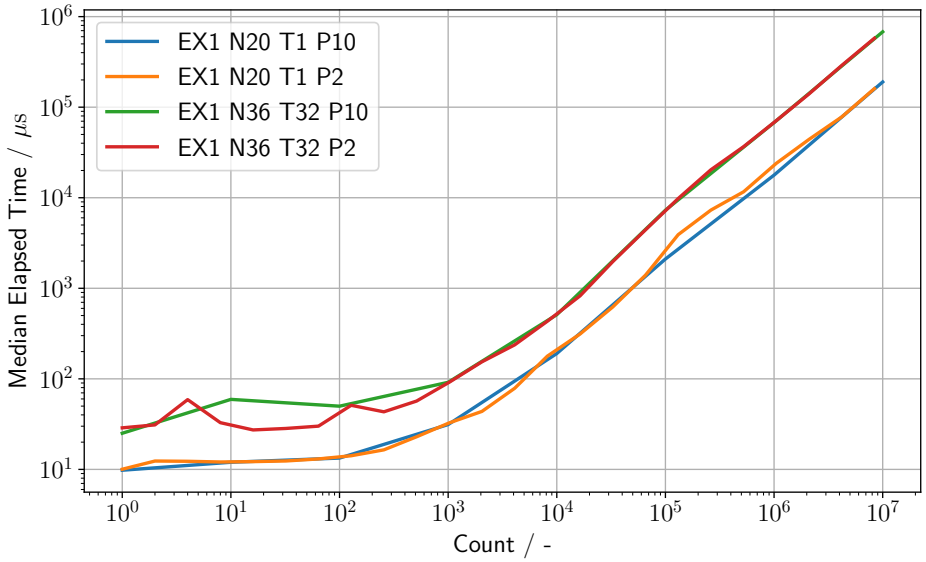

Figure 3: Median Timings for `MPI_Reduce + MPI_Bcast`, best and worst configurations

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 wuantities 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`

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 (Algorithms 16 and 17 in [2]).

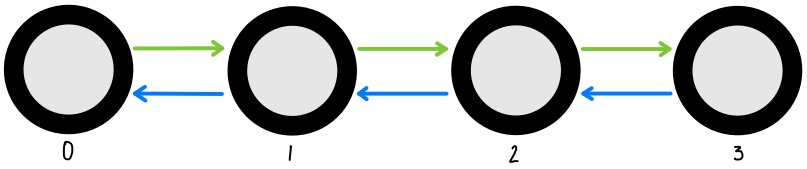

Figure 4: Linear structure for message passing scheme in exercises 2 and 3 where 0 is the root node.

Our implementation for the pipelined reduction – `MY_Reduce_P()` – aligns with the following idea: We distinct between a master process (rank = 0), interior processes (0 < rand < size −1) and an end process (rank = 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 (rank = size −1) sends blocks – one after the other – to the interior process with rank = 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 rank +1, perform a local reduction and send this data to their other neighbor with rank −1. This is repeated for all blocks. The master process receives block data from the interior node of 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 and n the total number of processes. The number of blocks results in a latency of $b-1$ and for the n processes are $n-1$ sucessive communications needed. In total, this leads to a maximum of $b+n-2$ 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 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 $b+n-2$ 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()`. In figure 4 the green arrows indicate the communication steps for `MY_Reduce_P()` and the blue arrows `MY_Bcast_P()`. Only once all reduction communications are finished (green arrows), the broadcasting (blue) starts. We end up with a total of $2 \cdot (b+n-1)$ communciation rounds for this

realisation of `MPI_Allreduce`.

In figures 5 and 6 below, it can be observed, that the linearly pipelined algorithm outperforms `MPI_Reduce`+ `MPI_Bcast` only for larger lenghts of count (approx. $> 5 \cdot 10^5$). The first one that outperforms it, is the one with blocksize $1 \cdot 10^4$ for both figures, meaning powers of 2 and 10.

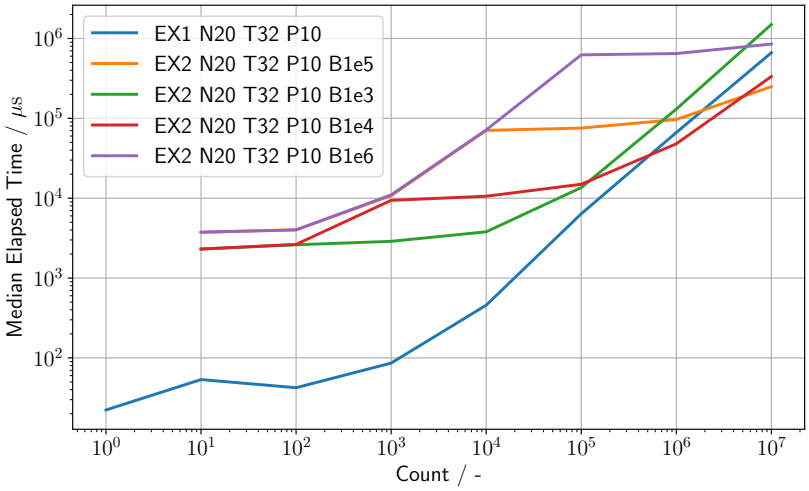

Figure 5: Median of elapsed time for linearly pipelined reduction and broadcasting. 20 nodes, 16 processes per node and powers of 2.

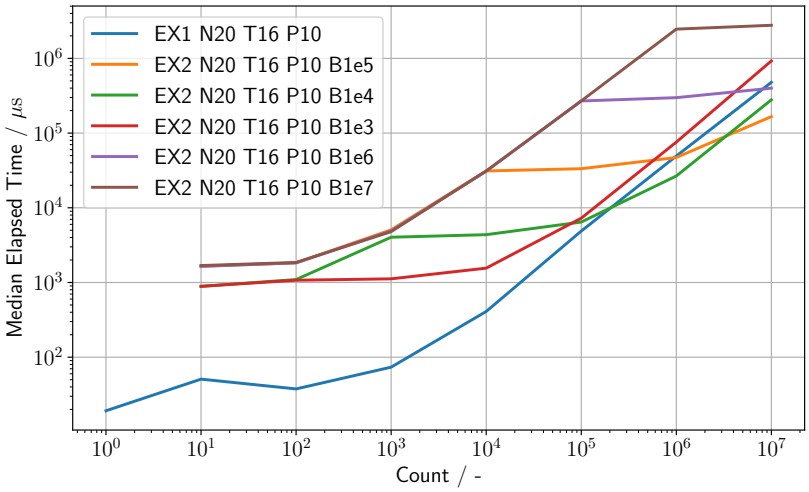

Figure 6: Median of elapsed time for linearly pipelined reduction and broadcasting. 20 nodes, 16 processes per node and powers of 10.

In the following two figures 7 and 8 and 9 we can observe the benefits of the different blocksizes for each configuration. For increasing count, the execution of `MY_Reduce_P()` and `MY_BCast_P()` appears to benefit from larger blocksizes, since the slopes of the timing graphs are less steep after they reach a certain number of count.

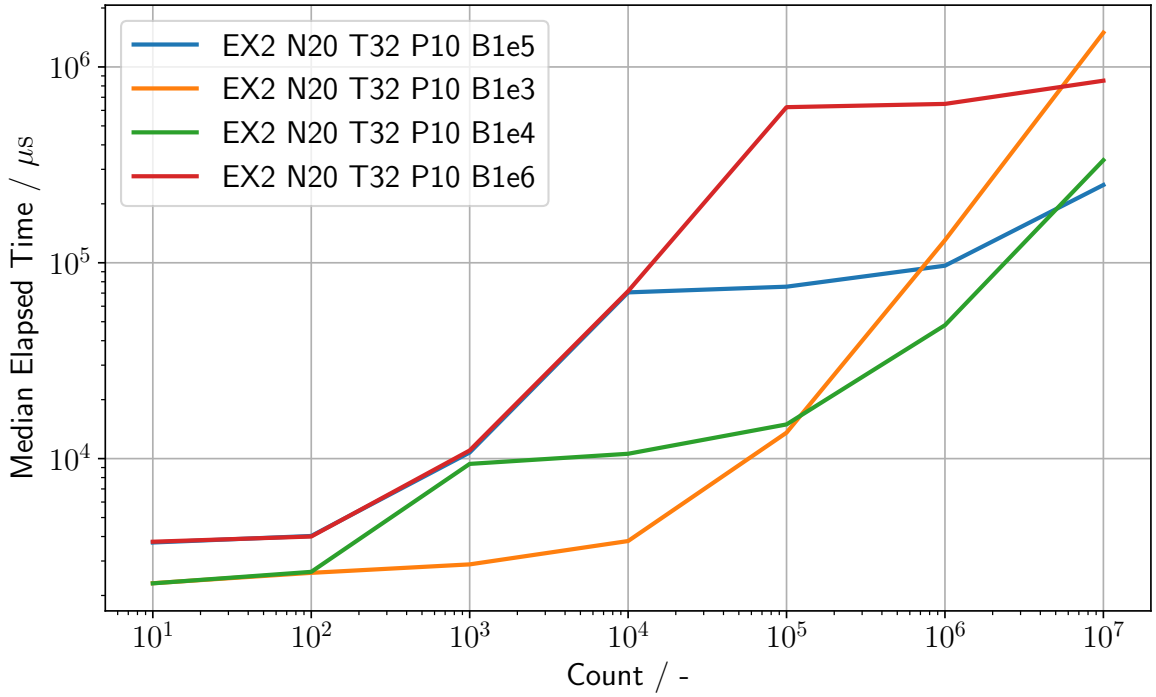

Figure 7: Caption for Ex2 plot 3

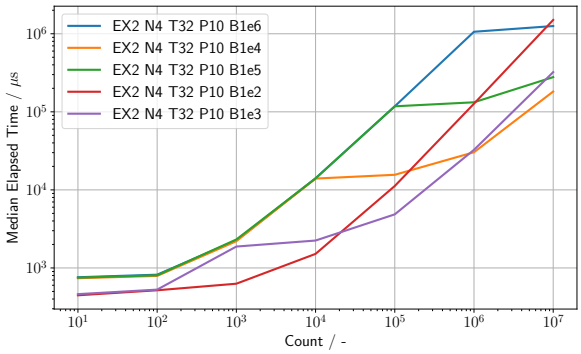

Figure 8: Caption for Ex2 plot 4

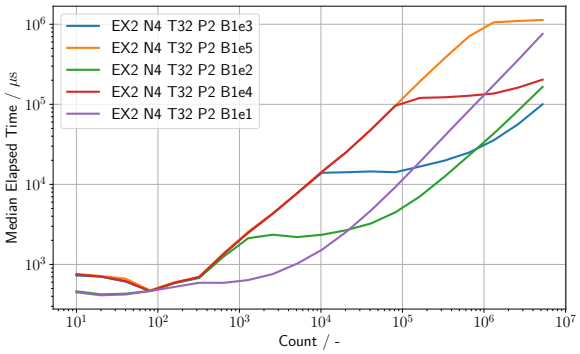

Figure 9: Caption for Ex2 plot 5

The same pattern can be observed for both configurations (Powers of 10 and 2) with 4 processes. In the next exercise, we will introduce a variable blocksize that is dependent on the number count.

3 Exercise 3 - Combining `MPI` Processes

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 (see chapter 5). 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. The same structure like in exercise 2 shown in figure 4 is used here in exercise 3 but now. In exercise 2, the blue arrows (broadcast) only started sending after ALL blocks where reduced, in exercise 3, the algorithm was designed such that a block could already be broadcasted once it was reduced, independent of the reduction-state of the other blocks.

For exercise 3 and the following exercises, we have implemented a simple variable blocksize based on the observations of the timings with different blocksizes in exercise 2.

C Code for variable blocksize

```c
int get_blockSize(int count){
    if (count < 1e4){
        return 1e2;
    }
    else if (count < 1e6){
        return 1e3;
    }
    else return 1e6;
}
```

In the figures below are illustrative results where we compare the performance of exercise 2 and 3. We can observe that algorithm coded in 3 is not growing performance wise with the time spent coding it. For count 10^3 the algorithm of Ex3 outperforms the linearly pipelined one of Ex2.

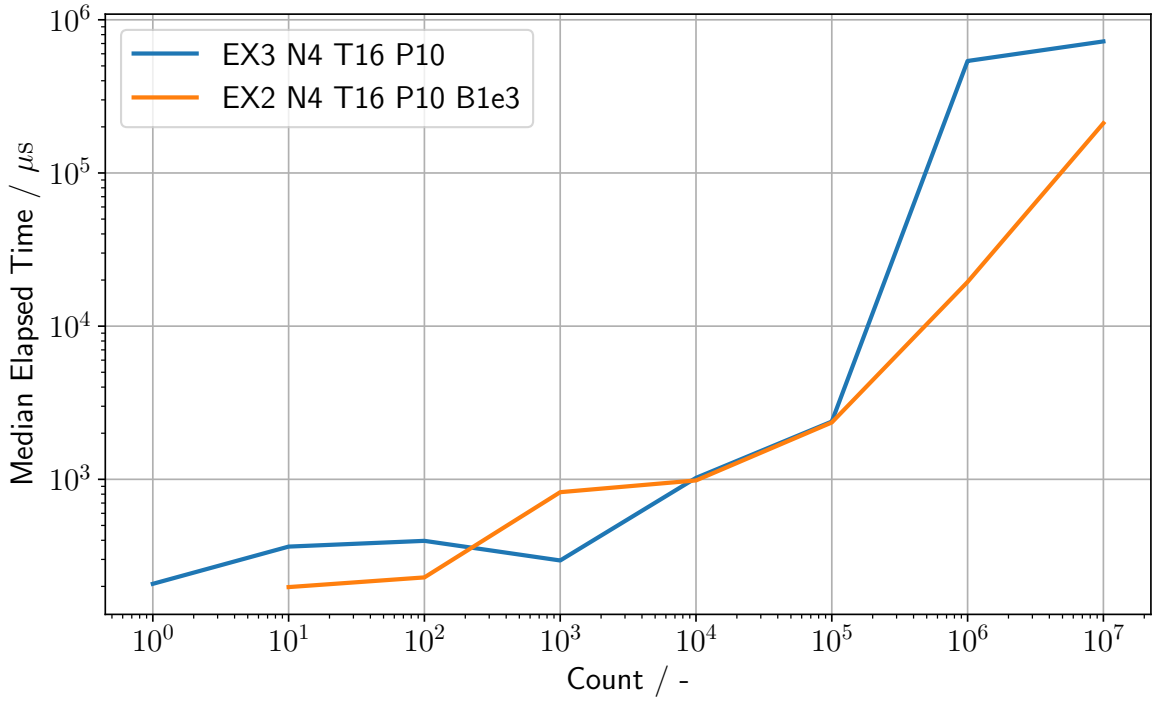

Figure 10: Median of elapsed time for `MY_Allreduce_P()`. 4 nodes, 16 processes per node, powers of 10, static blocksize for the EX2 timing and variable blocksize for EX3 timing. 20 nodes, 16 processes per node and powers of 10. Performance improvement for count 10^3.

In figure 11 we show the configuration of 4 nodes, both powers and all 3 tasks per processes numbers. It can be seen that the slope decreases again at certain numbers of counts for both powers of 2 and 10, this is most likely not due to the improved algorithm, but due to our variable blocksize.

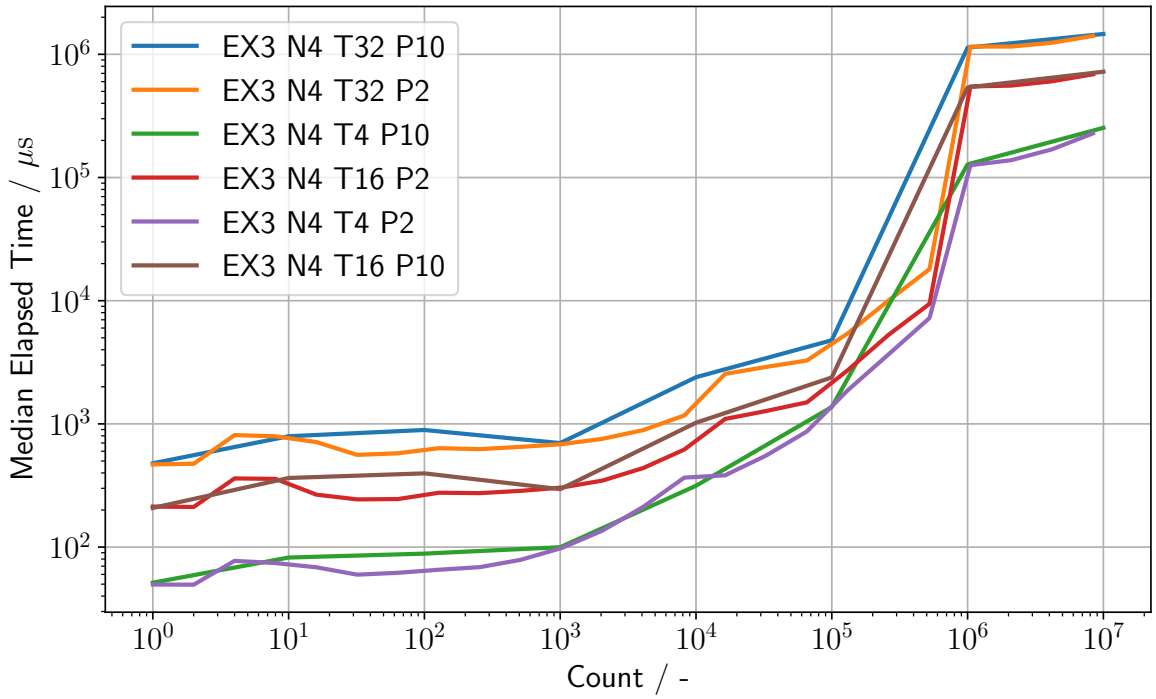

Figure 11: Median of elapsed time for `MY_Allreduce_P()`. 4 nodes, 4 processes per node, powers of 2 and 10, and variable blocksize. Performance improvements visible most likely due to variable blocksize.

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.

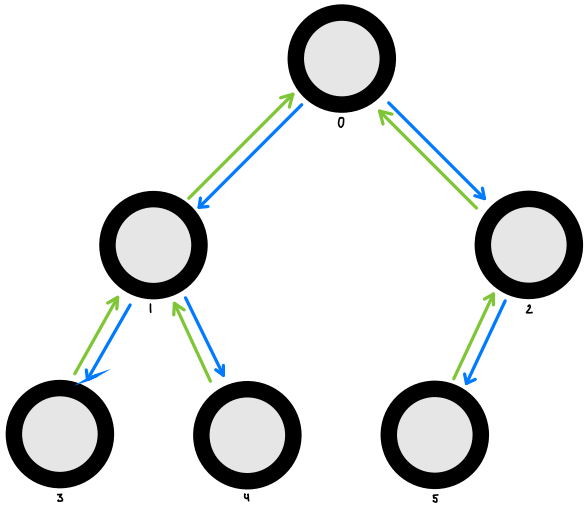

Figure 12: Binary tree topology for message passing scheme in exercises 4 and 5

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 implementatations of `MY_Reduce_T()` and `MY_Bcast_T()`. In figure 12 the considered binary tree structure can be seen ver well. The green arrows indicate the communication steps for `MY_Reduce_T()` and the blue arrows `MY_Bcast_T()`. Only once all reduction communications are finished (green arrows), the broadcasting (blue) starts – same as discussed in exercise 2.

The path of the exercise 4 graph after count = 10^3 would normally continue to intercept and perfrom worse than the others plotted, due to the blocksize adjustment that occurs at count = 10^6 this does not happen.

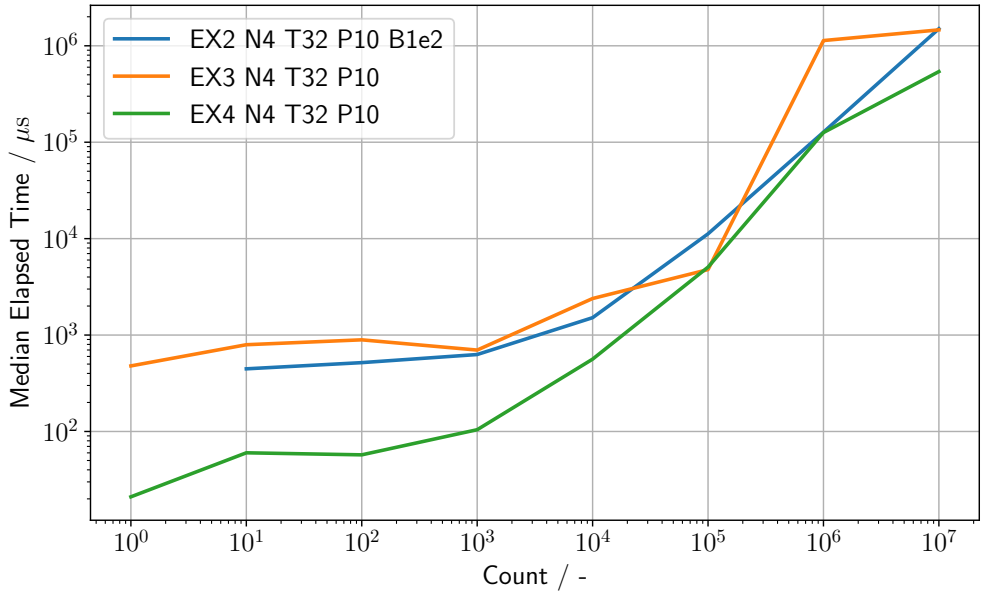

Figure 13: Median elapsed time for 4 nodes, 32 tasks per node and powers of 10. Clearly visible improvement over all count sizes. At count = 10^5 the `MY_Allreduce_P()` outperforms the tree algorithm `MY_Reduce_T()` + `MY_Bcast_T()` slightly. At count = 10^6 the tree algorihtm and the blocksize = 100 algorithm from exercise 2 coincide.

In figures 14 and 15 below, we can observe the slope improvements again due to the blocksize adjustment. What also pops out is that the graphs are tightly packed and appear to perform similar even though they have different configurations regarding the tasks per node.

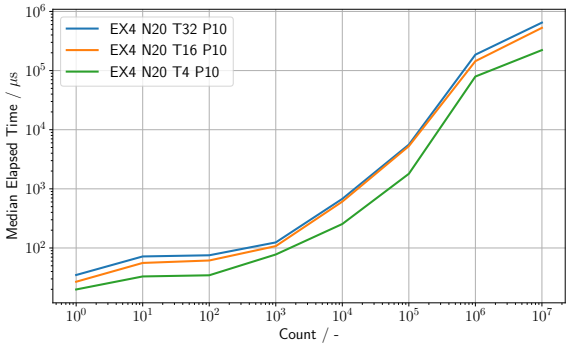

Figure 14: `MY_Reduce_T()` + `MY_Bcast_T()` tree algorithm with 20 nodes and powers of 10. Visible slope reduction due to blocksize adjustment at count = 10^6 again.

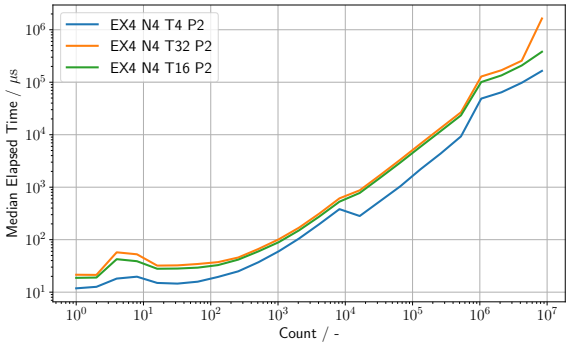

Figure 15: `MY_Reduce_T()` + `MY_Bcast_T()` tree algorithm with 20 nodes and powers of 2. Visible slope reduction due to blocksize adjustment at count = 10^6 as well.

5 Exercise 5 - Integrated, Improved Binary Tree Algorithm

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 [1] 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. We refer to the same structure as it was used for exercise 4 – figure 12. In exercise 4, the blue arrows (broadcast) only started sending after ALL blocks where reduced, in exercise 5, the algorithm was designed such that the green arows (reduction) and blue arrows (broadcasting) are allowed in the same communication steps. Meaning, a block could already be broadcasted once it was reduced, independent of the reduction-state of the other blocks.

For estimating the time complexity of `MY_Allreduce_T()` under a round based linear time transmission cost model, we make use of the pipelining lemma. With latency of $d_{max} = \lfloor \log_2(n) \rfloor$, which equals the maximum level of the binary tree, and a new block of blocksize b in every first round, we end up with a best possible runtime of

$$\alpha(d_{max} - 1) + 2\sqrt{(d_{max})\alpha\beta b} + \beta b, \quad \alpha, \beta \in \mathbb{R}$$

where α and β are latency parameters.

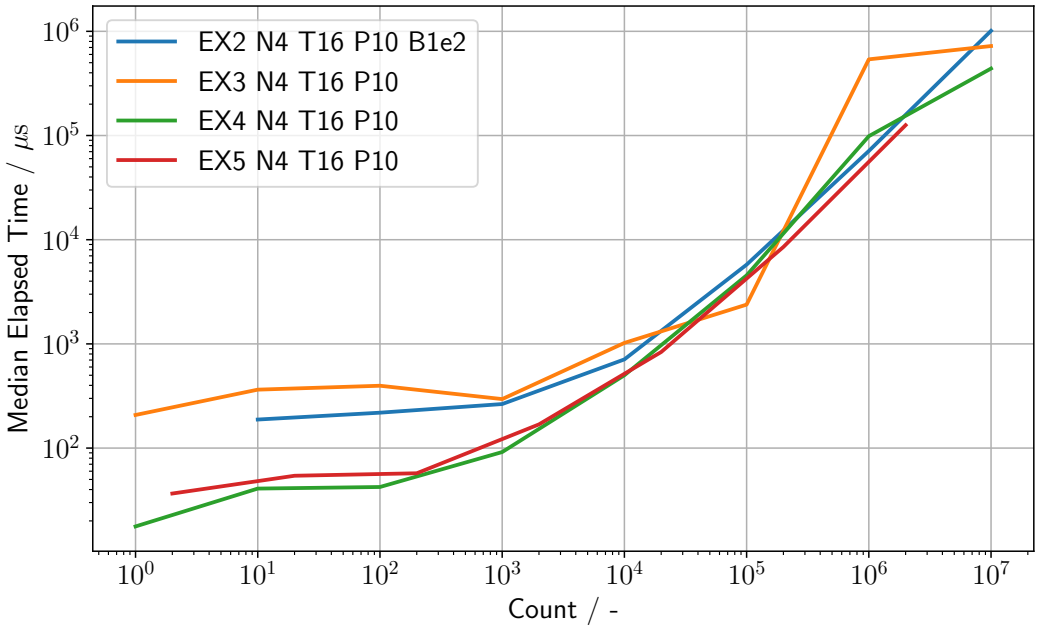

Figure 16: Median elapsed time for 4 nodes, 16 tasks per node and powers of 10, for algorithms from the previous exercises. There is no major difference in the performance of `MY_Reduce_T()` + `MY_Bcast()` compared to `MY_Allreduce_T`. For count $< 10^4$, `MY_Reduce_T()` + `MY_Bcast()` is of sligthly better performance, but for larger count `MY_Allreduce_T` is more efficient. Additionally, we can see that for even higher count, the implementatation of exercise 3 – `MY_Allreduce_T()` – seems to outperform `MY_Reduce_T()` + `MY_Bcast()` and `MY_Allreduce_T`.

In figures 17 and 18 below we can observe that the `MY_Allreduce_T()` tree algorithm benefits from a low number of tasks per processes. Interestingly, the timings for powers of 2 in figure 18 seem to all reduce their slope again when the blocksize adjustment occurs, while the timings for powers 10 in figure 17 on the right do not.

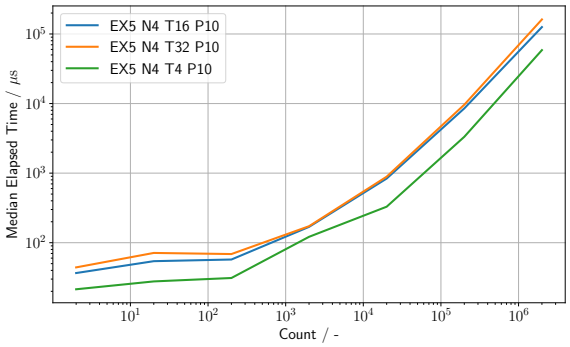

Figure 17: `MY_Allreduce_T()` tree algorithm with 4 nodes and powers of 2.

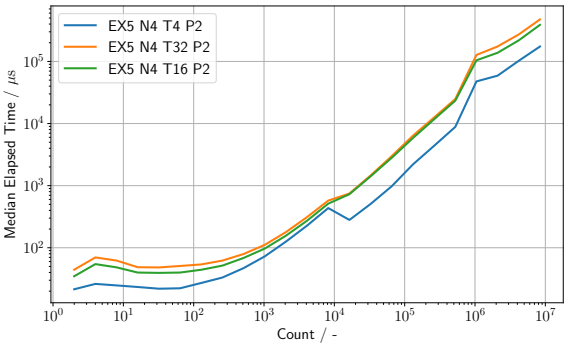

Figure 18: `MY_Allreduce_T()` tree algorithm with 4 nodes and powers of 10. Visible slope reduction due to blocksize adjustment at count = 10^6 again.

6 Exercise 6 - Improvement with Sibling Leave Communication

Not implemented.

7 Exercise 7 - Implemenetation and Benchmarking of Improved Version

Not implemented.

References

[1] J. L. Träff, "A doubly-pipelined, dual-root reduction-to-all algorithm and implementation," *CoRR*, vol. abs/2109.12626, 2021. arXiv: 2109.12626. [Online]. Available: https://arxiv.org/abs/2109.12626.

[2] J. L. Träff, "Algorithms for Collective Communication," Feb. 2022.