Vienna University of Technology

184.725 High Performance Computing

TU Wien Informatics

Exercise 2

Authors:
Camilo Tello Fachin
12127084

Supervisor:
Prof. Dr. Jesper Larsson Träff

January 15, 2023

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Abstract

Here documented the results of exercise 2, the Programming part of High Performance Computing. A presentation for Marco Schmid MSc.

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.

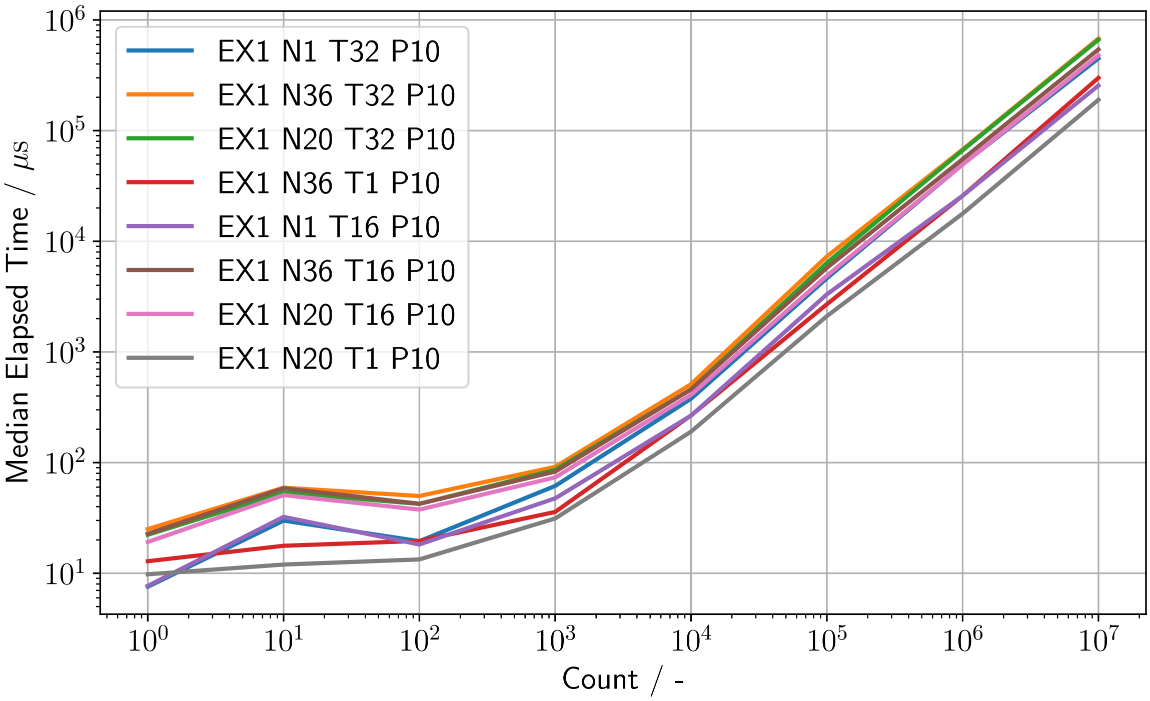

Figure 1: Median Timings for `MPI_Reduce + MPI_Bcast` for all Configurations and powers of 10. Since every `MPI` process performs its own timing, we use `MPI_Reduce(..,MPI_MAX)` to retrieve the timing of the slowest process. After 10 Warmup rounds in Ex1 we ran a 100 iterations to compute the statistic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

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 measurement are shown in the legend, where N stands for the number of Nodes, T for the number of tasks per thread 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.

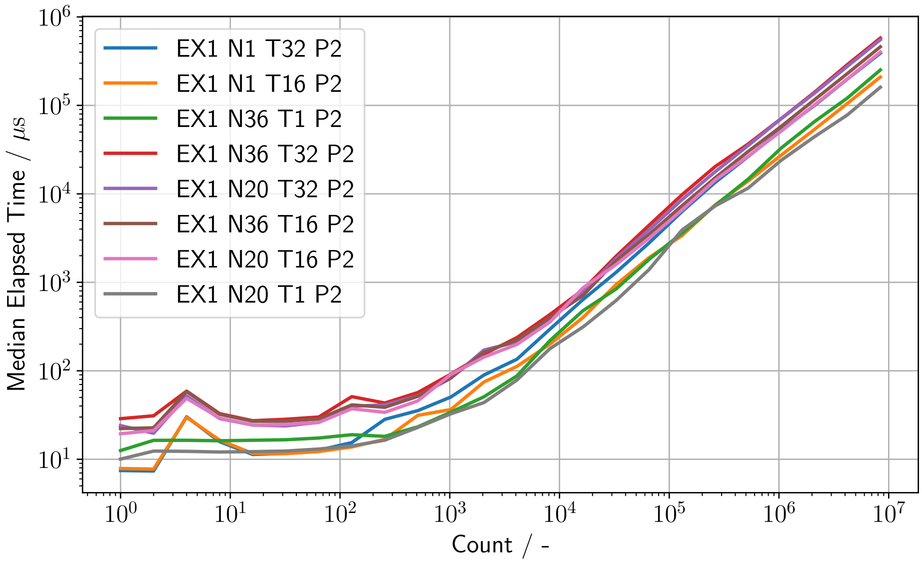

Figure 2: Median Timings for `MPI_Reduce + MPI_Bcast` with all Configurations and powers of 2

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 (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 (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.

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.

The trivial combination of `MY_Reduce_P()` and `MY_Bcast_P()` can be seen as a pipelined variant of

`MPI_Allreduce()`.

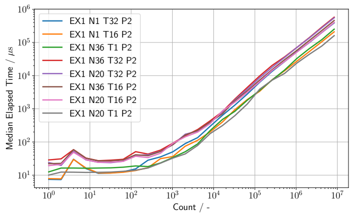

Figure 3: Complete (but unbalanced) binary tree of height $d = 4$ with $p = 9$ nodes.

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 (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 it's 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.

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().

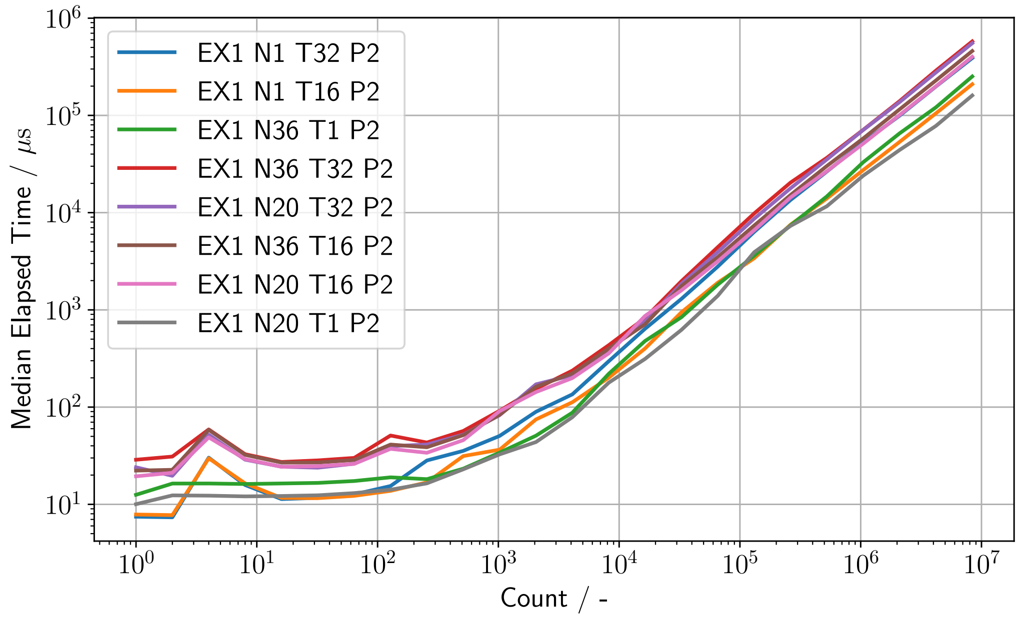

Figure 4: Benchmark both algorithms - yarg32() is faster than yarg() for all feasible values of d

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 [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.

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

Not implemented.

7 Exercise 7 - Implemenetation and Benchmarking of Improved Version (BONUS)

Not implemented.