HPC-Exercise2

Sara Carpenè

February 2024

1 Introduction

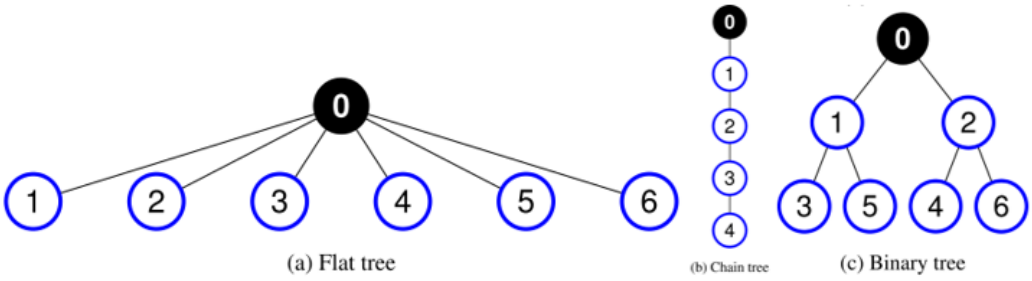
This report is about the solution of exercise 2a.

It has been chosen to implement a broadcast algorithm on a distributed memory with MPI. The main idea behind a broadcast algorithm is to ensure that a message originating from the root process is efficiently and reliably delivered to all other nodes in the system. The propagation mechanism depends on the specific broadcast algorithm being used, for this report it has been implemented and tested the following algorithm:

a) *Chain:* Processes are organized in a linear chain or sequence, where each node forwards the message to its immediate successor until it reaches the end of the chain.

b) *Flat:* The root directly sends the message to all other processes in the system.

c) *Binary tree:* Processes are organized in a binary tree structure, and the message is forwarded from the sender to the leaves of the tree.



As message to be exchanged in the broadcast it has been chosen an array. The size of the array is fixed to 2^{10} in the weak scaling and varies from 2 bytes to 1MB for strong scaling. The test have been performed on the ORFEO HPC cluster. It has been chosen different nodes between THIN and EPYC partition, this will be clarified in the following sections.

2 Implementation

To implement the different broadcast algorithms, it was chosen to use blocking send and receive to ensure that the messages were exchanged with success.

2.1 Chain Broadcast

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void chain_broadcast(int *data, int my_rank, int num_procs, int root_rank, int
    num_elements) {
    MPI_Status status;
    int parent_rank = my_rank - 1;
    int child_rank = my_rank + 1;


    if (my_rank == root_rank) {
        MPI_Send(data, num_elements, MPI_INT, child_rank, 0, MPI_COMM_WORLD);
    } else {
        MPI_Recv(data, num_elements, MPI_INT, parent_rank, 0, MPI_COMM_WORLD, &
    status);
        if (child_rank < num_procs) {
            MPI_Send(data, num_elements, MPI_INT, child_rank, 0, MPI_COMM_WORLD)
    ;
        }
    }
}
```

In this implementation, the ranks of the processes to which the message has to be sent and from which the message has to be received are computed first. Then, the receiving and sending operation are performed by all the nodes, with except of the root node that only sends, without receiving, whereas last node that only receive.

2.2 Flat broadcast

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void flat_tree_broadcast(int *data, int my_rank, int num_procs, int root_rank,
    int  num_elements) {
    MPI_Status status;

    if (my_rank == root_rank) {
            for (int i=1; i<num_procs;i++){
      MPI_Send(data,  num_elements , MPI_INT, i,  0, MPI_COMM_WORLD);
    }}
    else {
        MPI_Recv(data,   num_elements , MPI_INT, root_rank, 0, MPI_COMM_WORLD, &
    status);
    }
}
```

In this algorithm only the root process sends data to all the others processes. The remaining part of the processes only receives from the root process.

2.3 Binary tree broadcast

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void binary_tree_broadcast(int *data, int my_rank, int num_procs, int root_rank,
    int num_elements) {
    MPI_Status status;
    int parent_rank = (my_rank - 1) / 2;
    int left_child_rank = 2 * my_rank + 1;
    int right_child_rank = 2 * my_rank + 2;

    if (my_rank == root_rank) {
        if (left_child_rank < num_procs)
            MPI_Send(data, num_elements, MPI_INT, left_child_rank, 0,
    MPI_COMM_WORLD);
        if (right_child_rank < num_procs)
            MPI_Send(data, num_elements, MPI_INT, right_child_rank, 0,
    MPI_COMM_WORLD);
    } else {
        MPI_Recv(data, num_elements, MPI_INT, parent_rank, 0, MPI_COMM_WORLD, &
    status);
        if (left_child_rank < num_procs)
            MPI_Send(data, num_elements, MPI_INT, left_child_rank, 0,
    MPI_COMM_WORLD);
        if (right_child_rank < num_procs)
            MPI_Send(data, num_elements, MPI_INT, right_child_rank, 0,
    MPI_COMM_WORLD);
    }
}
```

Same thing can be said about this algorithm: at first it computes the rank of the parent and of the children. Then the communication proceeds in a binary tree structure. This implementation works thanks to the usage of blocking send and receive.

3 Strong scalability

To test the strong scalability it has been implemented a main in which a for loop allows to increase the size of the message at each iteration (with a bitwise left shift operator).

Before testing the different broadcast implementation it has been chosen to perform some warm-up communication. This was done because in the data collected without this strategy the time elapsed for the 2 dimensional array was surprisingly high with respect to the other collected data for big sized arrays, maybe this was due to the fact that there is some time needed to setup the communication, thus introducing the warm-up solved the problem.

To collect data the following steps were performed:

1. Synchronize the processes using `MPI_Barrier(MPI_COMM_WORLD)`.
2. Call the implemented broadcast functions 1000 times and then compute the average of the times obtained by each process.
3. Take the maximum of the computed averages across all processes using `MPI_Reduce(&averages, &maximum, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD)`.
4. The process 0 writes the results on the designed csv.

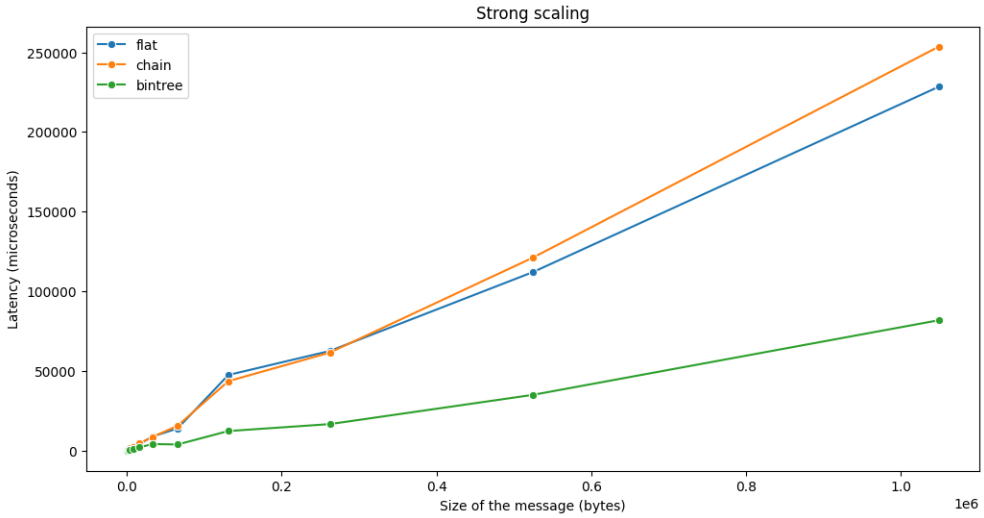


Figure 1: Strong scaling on EPYC node

In figure 1 it has been plotted the data collected on 2 EPYC nodes using all 256 processes. The size of the array passed as message varies from 1 to 2^{20}. It is clear that the binary tree performs better, this might be due to a better core utilization as trees can distribute the broadcasting load more effectively across the network.

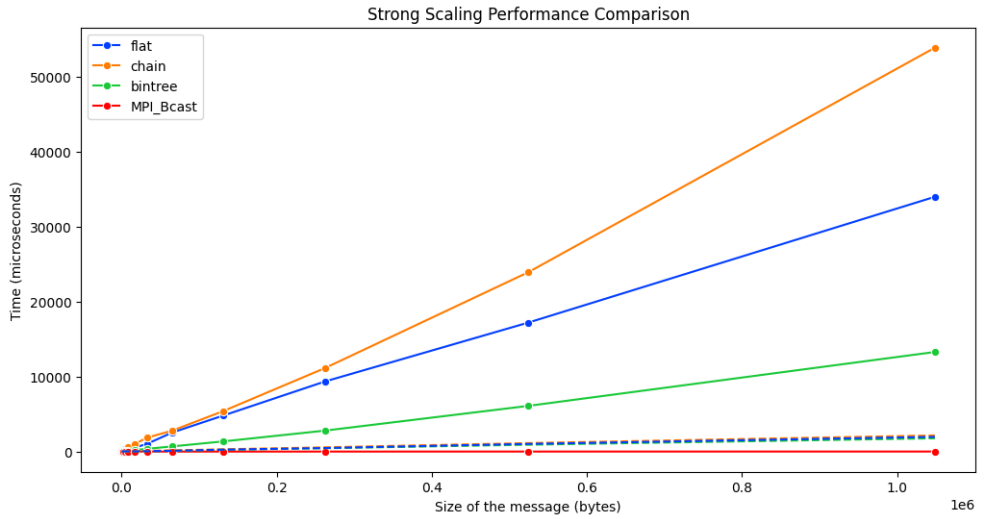


Figure 2: Comparison with MPI

In figure 2 it can be seen the results of running the same test on 2 THIN node. In this case the trend for every algorithm seems to be more linear, and it is clear that the chain implementation is the one that performs worst.

The red one is the data regarding the MPI_Bcast function of the MPI library, as it was expected the MPI implementation performs significantly better also for big size of messages.

The data collected with the corresponding algorithms in the OSU benchmark have been plotted in the same figure in dashed lines. As for the OSU implementations the binary tree is the best one and chain is the worst one.

4 Weak scalability

To test the weak scalability the data were collected with the same strategy of the one in strong scalability: warm-up of the communication, synchronization of the processes, reduction of the averages of the data and writing of the results.

Figure 3: Weak scaling epyc

To collect the data for figure 3 the size of the array has been fixed to 2^{10}. The test have been run on two EPYC nodes, and the number of processes varies from 2 to 256 "by two to two", since on every EPYC node on ORFEO there are 2x64 cores. The binary tree algorithm shows a relatively flat performance curve, which means it maintains a consistent performance level as the number of processes increases. This is expected from a binary tree algorithm, which should efficiently distribute the broadcast load across the processes in a hierarchical manner. The substantial grow of the time required by the flat implementation may be caused by the grow of the number of operation that the root has to do. Like the chain algorithm it is required more time to pass the message from the root to the last process. The data plotted in figure 4 have been obtained by running the same test on 4 THIN nodes, the number of processes varies from 2 to 96. The results are comparable to those obtained for the EPYC nodes, with the most notable difference observed in the flat implementation. In this case, the time required to complete the broadcast operation increases at a slower rate.

In figure 5 it has been plotted the results of running on 2 THIN nodes the algorithms presented above. In red there are the times of the MPI_Bcast function. In dashed lines there are the corresponding algorithms available in the OSU benchmark. It can be seen that the one that scales worst is the chain algorithm.

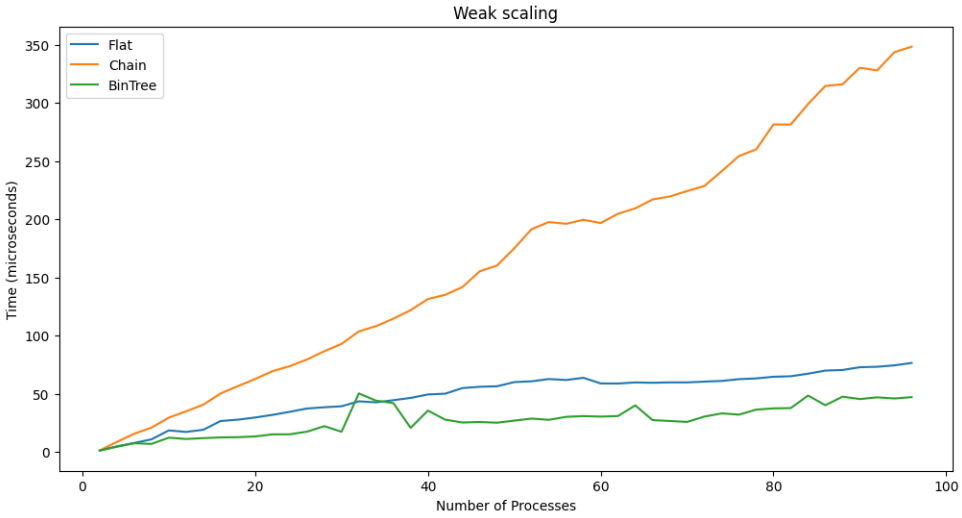


Figure 4: Weak scaling 4 thin

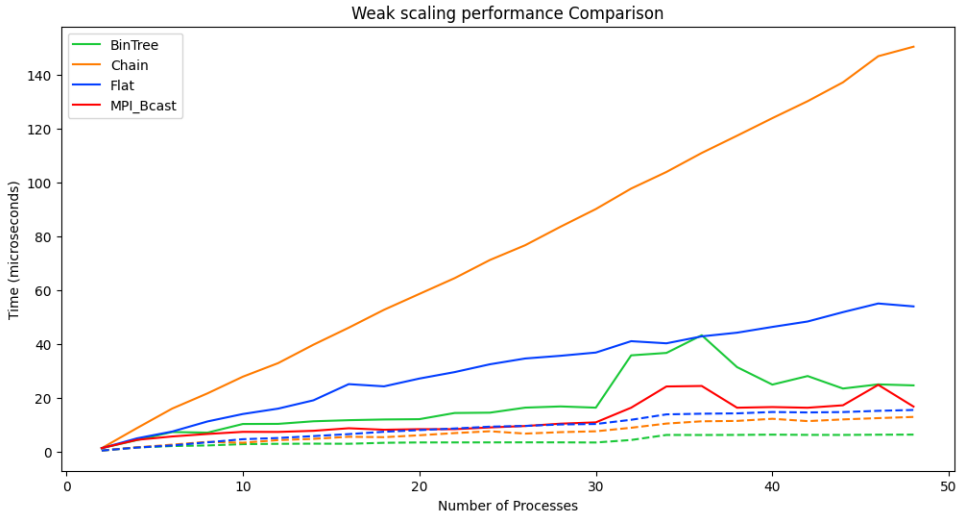


Figure 5: Comparison with MPI

5 Conclusions

The performances of the implemented algorithms are compatible with the expectations: the binary tree performs better than the other two. Maybe the implementations could be improved by using non blocking send and receive in order to decrease the message overhead, this would need a more sophisticated work in designing the algorithms to avoid that some processes start to send the message before receiving it or before the reception has been completed.

6 Referencies:

1. ORFEO documentation: https://orfeo-doc.areasciencepark.it/

2. Emin Nuriyev, Juan-Antonio Rico-Gallego, and Alexey Lastovetsky. "Model-based selection of optimal MPI broadcast algorithms for multi-core clusters.". Journal of Parallel and Distributed Computing, vol. 165, 2022, pp. 1-16. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2022.03.012.

3. MPI tutorial: https://github.com/mpitutorial/mpitutorial