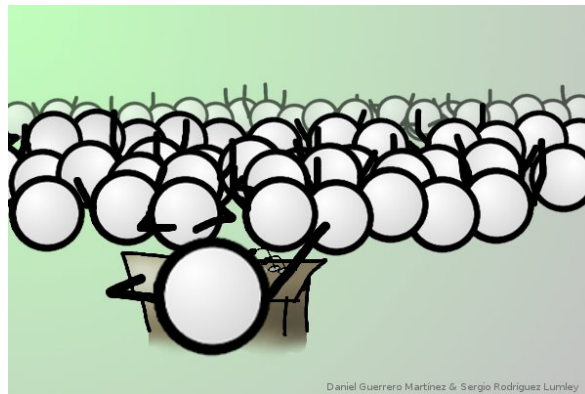


# High Performance Computing Project

---

## Implementation of MPI\_Bcast

---



### Elaborated by:

- Montassar Trimech
- Mohamed Aziz Tousli

### Supervised by:

- Saber Feki
- Malek Smaoui

**Academic Year :**

2019 – 2020

## I) Project overview

The main goal of this project is the comparison of different message-passing algorithms that depend on the **number of processors** and the **length of the message**. This document contains the implementation of MPI\_Bcast with blocking point to point communication operations using the following models: K-chain, Binary Tree and Binomial Tree. The report will explain also the method we used in order to make it possible to work for **any root** and **any message**.

## II) Code overview

In this part, we will explain the method we used in order to make it possible to work for any root and any message.

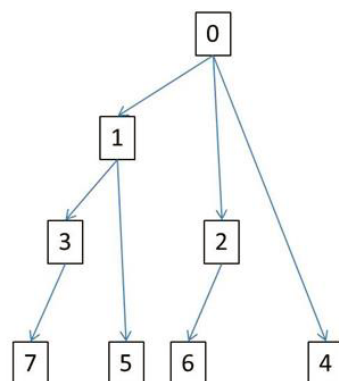
### 1. Any root

First of all, the code that we are going to implement will be in  $root = 0$ . And, in order to go from  $root = 0$  to  $any\ root$ , we developed a bijective function entitled *relative* and *relative\_inverse*. The first function will take us from *rank* to *relative\_rank*, and the second one will get us from *relative\_rank* to *rank*. The code source of the two functions is presented below.

```
1. int relative(int rank, int root, int size)
2. {
3.     int relative_rank;
4.     if (rank >= root)
5.         relative_rank = rank - root;
6.     else
7.         relative_rank = rank - root + size;
8.     return relative_rank;
9. }
10.
11. int relative_inverse(int relative_rank, int root, int size)
12. {
13.     int rank;
14.     if (relative_rank + root < size)
15.         rank = relative_rank + root;
16.     else
17.         rank = relative_rank + root - size;
18.     return rank;
19. }
```

The following table shows the bijection between *rank* and *relative\_rank* when  $root = 3$ . It illustrates the conversion of the different processors in order to turn  $root = 3$  to  $root = 0$ . Let's note that the table depends on the number of processors and the root. The following graph shows the process of Binomial Tree when  $root = 0$ .

<i>rank</i>	<i>relative_rank</i>
0	5
1	6
2	7
3	0
4	1
5	2
6	3
7	4



Let's suppose that the processor with *rank* = 5 will send.

- 1- Use the table (using *relative* function) to convert from *rank* = 5 to *relative\_rank* = 2.
- 2- Check from the graph (which is built on *root* = 0) that *rank* = 2 will send to *rank* = 6.
- 3- Use the table (using *relative\_inverse* function) to convert from *relative\_rank* = 6 to *rank* = 1.

Therefore, we can understand that the processor with *rank* = 5 will send to the processor with *rank* = 1.

## 2. Any message

In this part, we will show how we dealt with the algorithm in order to accept a message of any type. First of all, in order to accept a message of any type, we will parametrize the buffer as a pointer on void.

```
void MPI_Binomial_Tree(void *buf, int cnt, MPI_Datatype dat, int root, MPI_Comm comm)
```

In addition, we should do the same thing when the message is received at the level of MPI\_Recv. Also, the received type is not mentioned specifically because *dat* represents any type of data.

```
void *message = (void *)malloc(N*sizeof(dat));
```

NB: In all the tests that will be discussed, we used type *MPI\_INT*. So, the message will be an array of integers.

```
1. int *buf = (int *)malloc(N*sizeof(int)); // N = size of the message
2. for (int i=0; i<N; i++)
3.     buf[i]=i;
```

## III) The Algorithms

The algorithms will be explained at the level of *root* = 0. We have already discussed the case for a different root.

Below is the *main()* function that will be used to test the different implemented algorithms.

```
1. int main(int argc, char** argv)
2. {
3.     // Basic MPI Initializations
4.     MPI_Init(&argc, &argv);
5.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6.     MPI_Comm_size(MPI_COMM_WORLD, &size);
7.     // Prepare the message : in this test the message is an array of integer
8.     int *buf=(int *)malloc(N*sizeof(int));    /// N number of elements in the array
9.     for (int i=0; i<N; i++)
10.         buf[i]=i;
11.     int root=0;
12.     // MPI_Binomial_Tree(buf,N,MPI_INT,root,MPI_COMM_WORLD);
13.     // MPI_Binary_Tree(buf,N,MPI_INT,root, MPI_COMM_WORLD);
14.     // MPI_Kchain(buf, N, MPI_INT, root,MPI_COMM_WORLD, k);
15.     // MPI_Bcast (buf, N,MPI_INT , root, MPI_COMM_WORLD );
16.     MPI_Finalize ();
17.     return (0);
18. }
```

## 1. Binomial Tree

### a. Explication

This algorithm follows this pattern.

#### 1<sup>st</sup> iteration:

*Root = 0* will send to processor with *rank = 1*.

#### 2<sup>nd</sup> iteration:

Now, processors with *rank = 0* and *rank = 1* have messages.

Processor with *rank = 0* will send to processor with *rank = 2*.

Processor with *rank = 1* will send to processor with *rank = 3*.

And so on, and so forth...

We can observe that the processor with *rank = 0* will send to a specific list of processors {1, 2, 4, 8, ...}, and processor with *rank = 1* will send to another specific list of processors {3, 5, 9, 17, ...}. Thus, when a processor receives a message, it will send it to a defined list of processors.

So, we can conclude that the algorithm can be represented in the following way.

#### 0<sup>th</sup> step:

*Root = 0* will send to processors with rank {1, 2, 4, 8, ...,  $k$ ;  $k \leq size$ }

#### i<sup>th</sup> step: ( $i > 0$ )

Processor with *rank = n* will receive the message **only and only once** from the processor with *rank =  $n - 2^{\log_2 n}$* , and send the message to the list of processors with *rank  $\in \{rank = 2^k + n, \text{ where } i \leq k \leq \#steps\}$* .

### b. Tests

Below is the function used to execute this algorithm.

```
MPI_Binomial_Tree(buf,N,MPI_INT,root,MPI_COMM_WORLD);
```

We will send a table of *integers* from 0 to 9 on one hand, and a table of *doubles* on the other hand.

$$buf_{int} = [0,1,2,3,4,5,6,7,8,9]$$
$$buf_{float} = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]$$

The following commands will be used in the terminal in order to test.

```
mpicc HPC_Project.c -o binomialTree
mpirun -np NB_PROCESSORS --oversubscribe ./binomialTree
```

NB: *oversubscribe* is used to force the computer to work on more than two processors.

Now, we will be testing on different roots, different number of processors and different types of messages.

Example 1:  $root = 0$ ,  $NB\_PROCESSORS = 10$ ,  $type = MPI\_INT$ .

```
montassar@montassar-X75VC:~/HPC$ mpirun -np 10 --oversubscribe ./binomialTree
I'm process rank : 0
I'm process rank : 1
Rank 1: Received the message from rank 0 with values : 0 1 2 3 4 5 6 7 8 9
Rank 0 : send the message to rank 1 with values : 0 1 2 3 4 5 6 7 8 9
Rank 0 : send the message to rank 2 with values : 0 1 2 3 4 5 6 7 8 9
Rank 0 : send the message to rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 1 : send the message to rank 3 with values : 0 1 2 3 4 5 6 7 8 9
Rank 0 : send the message to rank 8 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 8
Rank 8: Received the message from rank 0 with values : 0 1 2 3 4 5 6 7 8 9
Rank 1 : send the message to rank 5 with values : 0 1 2 3 4 5 6 7 8 9
Rank 1 : send the message to rank 9 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 2
Rank 2: Received the message from rank 0 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 5
Rank 2 : send the message to rank 6 with values : 0 1 2 3 4 5 6 7 8 9
Rank 5: Received the message from rank 1 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 3
Rank 3: Received the message from rank 1 with values : 0 1 2 3 4 5 6 7 8 9
Rank 3 : send the message to rank 7 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 6
Rank 6: Received the message from rank 2 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 7
Rank 7: Received the message from rank 3 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 4
Rank 4: Received the message from rank 0 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 9
Rank 9: Received the message from rank 1 with values : 0 1 2 3 4 5 6 7 8 9
```

Example 2:  $root = 0$ ,  $NB\_PROCESSORS = 15$ ,  $type = MPI\_INT$ .

```
montassar@montassar-X75VC:~/HPC$ mpirun -np 15 --oversubscribe ./binomialTree
I'm process rank : 0
SublimeText led the message to rank 1 with values : 0 1 2 3 4 5 6 7 8 9
Rank 0 : send the message to rank 2 with values : 0 1 2 3 4 5 6 7 8 9
Rank 0 : send the message to rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 0 : send the message to rank 8 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 2
Rank 2: Received the message from rank 0 with values : 0 1 2 3 4 5 6 7 8 9
Rank 2 : send the message to rank 6 with values : 0 1 2 3 4 5 6 7 8 9
Rank 2 : send the message to rank 10 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 8
Rank 8: Received the message from rank 0 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 1
Rank 1: Received the message from rank 0 with values : 0 1 2 3 4 5 6 7 8 9
Rank 1 : send the message to rank 3 with values : 0 1 2 3 4 5 6 7 8 9
Rank 1 : send the message to rank 5 with values : 0 1 2 3 4 5 6 7 8 9
Rank 1 : send the message to rank 9 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 3
Rank 3: Received the message from rank 1 with values : 0 1 2 3 4 5 6 7 8 9
Rank 3 : send the message to rank 7 with values : 0 1 2 3 4 5 6 7 8 9
Rank 3 : send the message to rank 11 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 6
Rank 6: Received the message from rank 2 with values : 0 1 2 3 4 5 6 7 8 9
Rank 6 : send the message to rank 14 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 7
Rank 7: Received the message from rank 3 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 5
Rank 5: Received the message from rank 1 with values : 0 1 2 3 4 5 6 7 8 9
Rank 5 : send the message to rank 13 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 13
Rank 13: Received the message from rank 5 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 14
Rank 14: Received the message from rank 6 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 9
Rank 9: Received the message from rank 1 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 10
Rank 10: Received the message from rank 2 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 4
Rank 4: Received the message from rank 0 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 12 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 12
Rank 12: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 11
Rank 11: Received the message from rank 3 with values : 0 1 2 3 4 5 6 7 8 9
montassar@montassar-X75VC:~/HPC$
```

Example 3:  $root = 3$ ,  $NB\_PROCESSORS = 8$ ,  $type = MPI\_INT$ .

```
montassar@montassar-X75VC:~/HPC$ mpicc HPC_Project.c -o binomialTree
montassar@montassar-X75VC:~/HPC$ mpirun -np 8 --oversubscribe ./binomialTree
I'm process rank : 3
Rank 3 : send the message to rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 3 : send the message to rank 5 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 5
Rank 5: Received the message from rank 3 with values : 0 1 2 3 4 5 6 7 8 9
Rank 5 : send the message to rank 1 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 1
Rank 1: Received the message from rank 5 with values : 0 1 2 3 4 5 6 7 8 9
Rank 3 : send the message to rank 7 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 4
Rank 4: Received the message from rank 3 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 6 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 6
Rank 6: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 0 with values : 0 1 2 3 4 5 6 7 8 9
Rank 6 : send the message to rank 2 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 0
Rank 0: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 2
Rank 2: Received the message from rank 6 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 7
Rank 7: Received the message from rank 3 with values : 0 1 2 3 4 5 6 7 8 9
montassar@montassar-X75VC:~/HPC$
```

Example 4:  $root = 6$ ,  $NB\_PROCESSORS = 8$ ,  $type = MPI\_INT$ .

```
montassar@montassar-X75VC:~/HPC$ mpicc HPC_Project.c -o binomialTree
montassar@montassar-X75VC:~/HPC$ mpirun -np 8 --oversubscribe ./binomialTree
I'm process rank : 6
Rank 6 : send the message to rank 7 with values : 0 1 2 3 4 5 6 7 8 9
Rank 6 : send the message to rank 0 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 0
Rank 0: Received the message from rank 6 with values : 0 1 2 3 4 5 6 7 8 9
Rank 6 : send the message to rank 2 with values : 0 1 2 3 4 5 6 7 8 9
Rank 0 : send the message to rank 4 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 2
Rank 2: Received the message from rank 6 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 7
Rank 7: Received the message from rank 6 with values : 0 1 2 3 4 5 6 7 8 9
Rank 7 : send the message to rank 1 with values : 0 1 2 3 4 5 6 7 8 9
Rank 7 : send the message to rank 3 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 4
Rank 4: Received the message from rank 0 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 1
Rank 1: Received the message from rank 7 with values : 0 1 2 3 4 5 6 7 8 9
Rank 1 : send the message to rank 5 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 3
Rank 3: Received the message from rank 7 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 5
Rank 5: Received the message from rank 1 with values : 0 1 2 3 4 5 6 7 8 9
montassar@montassar-X75VC:~/HPC$
```

Example 5: *root = 0, NB\_PROCESSORS = 8, type = MPI\_DOUBLE.*

```
montassar@montassar-X75VC:~/HPC$ mpicc HPC_Project.c -o binomialTree
montassar@montassar-X75VC:~/HPC$ mpirun -np 8 --oversubscribe ./binomialTree
I'm process rank : 0
Rank 0 : send the message to rank 1 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
Rank 0 : send the message to rank 2 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
Rank 0 : send the message to rank 4 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
I'm process rank : 4
Rank 4: Received the message from rank 0 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
I'm process rank : 1
Rank 1: Received the message from rank 0 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
Rank 1 : send the message to rank 3 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
Rank 1 : send the message to rank 5 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
I'm process rank : 3
Rank 3: Received the message from rank 1 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
Rank 3 : send the message to rank 7 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
I'm process rank : 2
Rank 2: Received the message from rank 0 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
Rank 2 : send the message to rank 6 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
I'm process rank : 5
Rank 5: Received the message from rank 1 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
I'm process rank : 7
Rank 7: Received the message from rank 3 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
I'm process rank : 6
Rank 6: Received the message from rank 2 with values : 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
montassar@montassar-X75VC:~/HPC$
```

NB: We observe that the processor that sends the most is the root, which is obvious since it is characterized in the algorithm.

## 2. Binary Tree

### a. Explication

This algorithm can be represented following this pattern.

0<sup>th</sup> step:

*Root = 0* will send to processors with *rank = 1* and *rank = 2*.

i<sup>th</sup> step: ( $i > 0$ )

Processor with *rank = n* will receive the message **only and only once** from the processor with  $rank = \frac{n-1}{2}$ , and send the message to the processors with  $rank = 2.n + 1$  and  $rank = 2.n + 2$ .

### b. Tests

Below is the function used to execute this algorithm.

```
MPI_Binary_Tree(buf,N,MPI_INT,root,MPI_COMM_WORLD);
```

Example: *root = 4, NB\_PROCESSORS = 8, type = MPI\_INT.*

```
montassar@montassar-X75VC:~/Lot$ mpicc HPC_Project.c -o binaryTree
montassar@montassar-X75VC:~/Lot$ mpirun -np 8 --oversubscribe ./binaryTree
I'm process rank : 4
I'm process rank : 2
Rank 2: Received the message from rank 6 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 5
Rank 5: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 5 : send the message to rank 7 with values : 0 1 2 3 4 5 6 7 8 9
Rank 5 : send the message to rank 0 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 6
Rank 6: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 6 : send the message to rank 1 with values : 0 1 2 3 4 5 6 7 8 9
Rank 6 : send the message to rank 2 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 5 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 6 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 1
Rank 1: Received the message from rank 6 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 3
Rank 3: Received the message from rank 7 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 7
Rank 7: Received the message from rank 5 with values : 0 1 2 3 4 5 6 7 8 9
Rank 7 : send the message to rank 3 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 0
Rank 0: Received the message from rank 5 with values : 0 1 2 3 4 5 6 7 8 9
```



### 3. K-Chain

#### a. Explication

This algorithm can be represented following this pattern.

0<sup>th</sup> step:

*Root* = 0 will send to processors with rank {1, 2, 3, 4, ..., *K*}. *K* is a pre-defined constant.

i<sup>th</sup> step: (*i* > 0)

Processor with *rank* = *n* will receive the message **only and only once** from:

If (*n* ≤ *K*):

Processor with *rank* = 0.

Else:

Processor with *rank* = *n* − *k*.

And send the message to the processors with *rank* = *n* + *k*.

#### b. Tests

Below is the function used to execute this algorithm.

```
MPI_Kchain(buf,N,MPI_INT,root,MPI_COMM_WORLD);
```

Example: *root* = 4, *NB\_PROCESSORS* = 8, *type* = *MPI\_INT*.

```
montassar@montassar-X75VC:~/Lot$ mpicc HPC_Project.c -o K_chain
montassar@montassar-X75VC:~/Lot$ mpirun -np 8 --oversubscribe ./K_chain
I'm process rank : 4
Rank 4 : send the message to rank 5 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 5
Rank 5: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 6 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 7 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 7
Rank 7: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 0 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 0
Rank 0: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 1 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 1
Rank 1: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 2 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 2
Rank 2: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
Rank 4 : send the message to rank 3 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 3
Rank 3: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
I'm process rank : 6
Rank 6: Received the message from rank 4 with values : 0 1 2 3 4 5 6 7 8 9
```

## IV) Comparison with MPI\_Bcast

Let's note that the main features of the comparison are **the number of processors** and **the size of the message**. The comparison will be based on **time execution** mainly.

NB: In order to optimize the code execution and to have a concrete comparison, we removed everything that is related to printing to the screen, because *printf()* takes a lot of time to execute.

There are two ways to make the comparison between Binomial Tree and MPI\_Bcast.

- The first method is by using the bash function *time* which gives three different execution duration values:
  - *Real* is wall clock time, time from start to finish of the call.
  - *User* is the amount of CPU time spent in user-mode code within the process.
  - *Sys* is the amount of CPU time spent in the kernel within the process.
  - *User+Sys* is how much actual CPU time your process used.

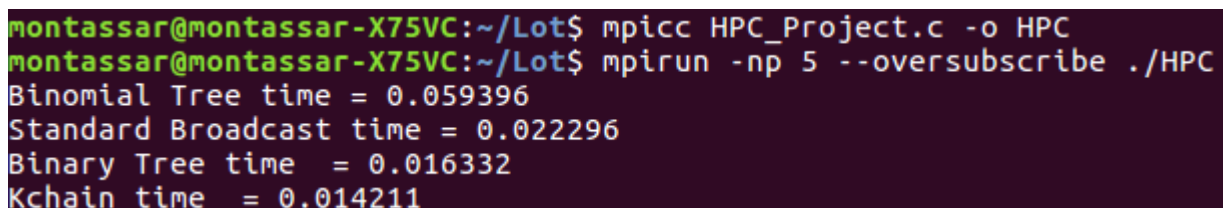
→ So, in this method, we can either use *Real* or *User+Sys* to make the comparison. The most important thing is that we compare using the same feature.
- The second method is by using the already implemented function *MPI\_Wtime()*. The latter returns time in seconds since an arbitrary time in the past. This “time in the past” is fixed from the first use of the function, so it is guaranteed that it won’t change during the life of the process.
  - We will be going to use this method, because we can get the time of the different algorithms on one execution – using the synchronization with *MPI\_Barrier()* – instead of executing it multiple times, like the first method.

For example, we will use this piece of code in order to calculate the time for a Binomial Tree.

```
1. double time_BinomialTree, average_time;
2. MPI_Barrier(MPI_COMM_WORLD);
3. time_BinomialTree -= MPI_Wtime();
4. MPI_Binomial_Tree(buf, N, MPI_INT, root, MPI_COMM_WORLD);
5. MPI_Barrier(MPI_COMM_WORLD);
6. time_BinomialTree += MPI_Wtime();
7. MPI_Reduce(&time_BinomialTree, &average_time, 1, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);
8. if (time == root)
9.     printf("Binomial Tree time en ms = %f\n", 1000*(average_time/size));
```

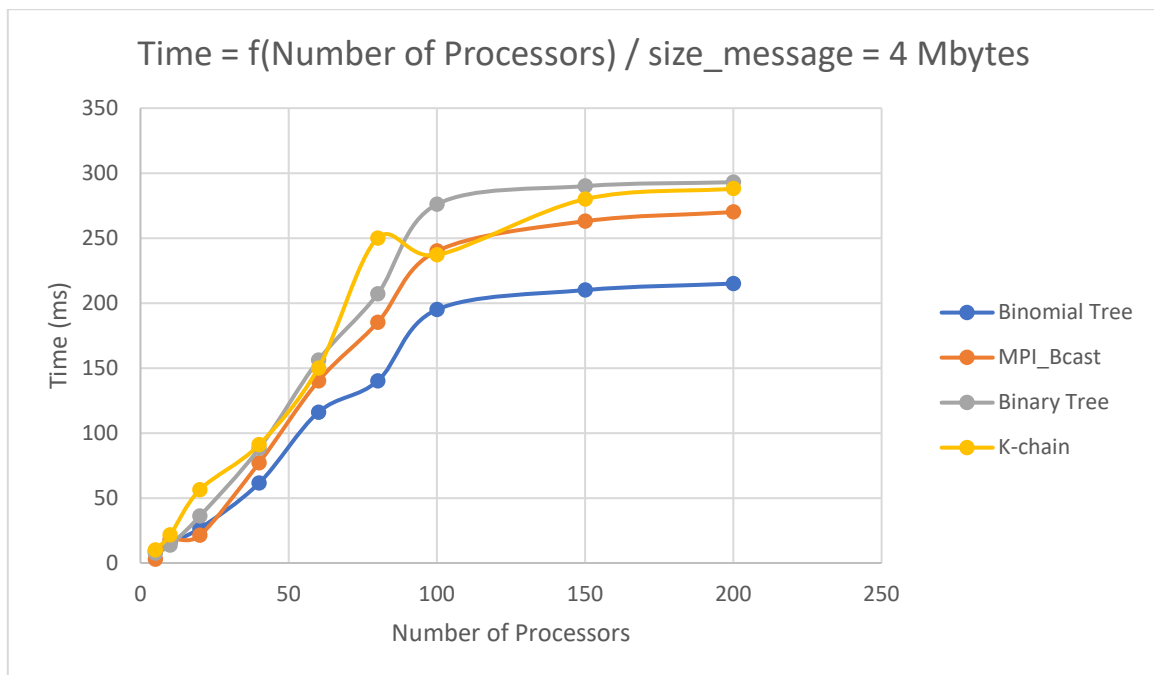
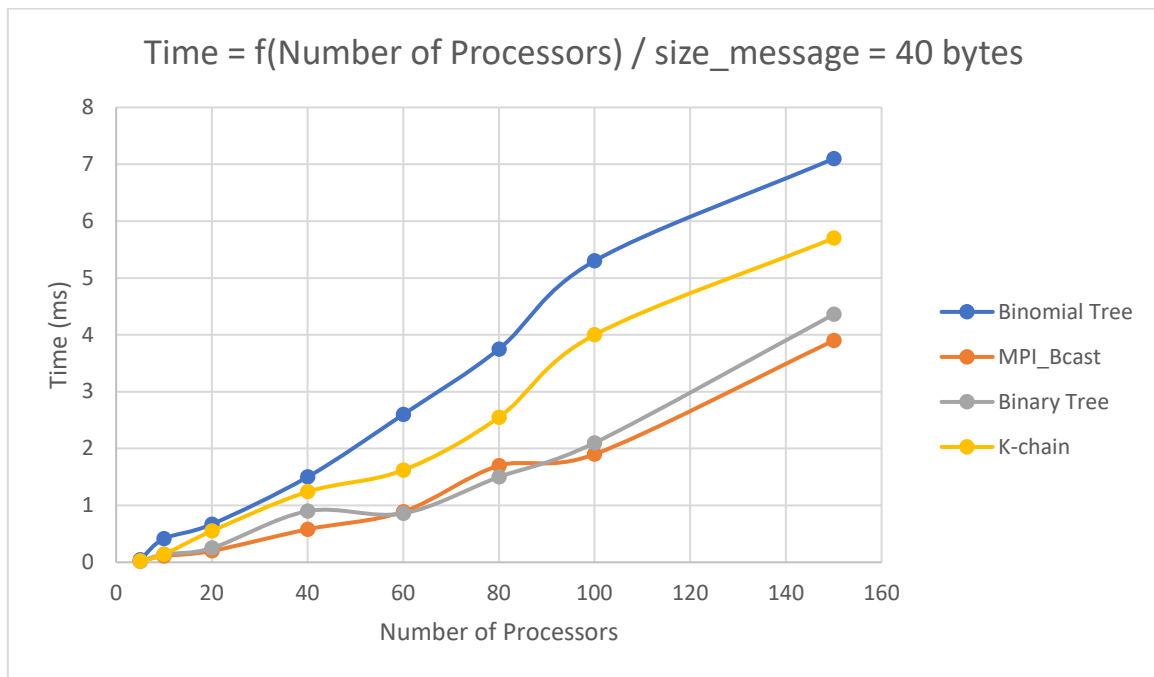
Let’s note that *MPI\_Reduce()* sums the time execution of each processor, and then we divide by size to get the average.

And this is a screenshot of one of the executions.



```
montassar@montassar-X75VC:~/Lot$ mpicc HPC_Project.c -o HPC
montassar@montassar-X75VC:~/Lot$ mpirun -np 5 --oversubscribe ./HPC
Binomial Tree time = 0.059396
Standard Broadcast time = 0.022296
Binary Tree time = 0.016332
Kchain time = 0.014211
```

Now, we will draw two charts. In the first one, we will fix the size of the message at 40 bytes, and vary the number of the processors. In the second one, we will fix the size of the message at 4 Mbytes.



The charts above represent the variation of the time in function of the number of processors for the three implemented algorithms and the already defined MPI\_Bcast. Overall, it can be seen that MPI\_Bcast has the best results for a short message, whilst, Binomial Tree presents faster execution for longer messages.

It can also be seen that the charts have a logarithmic curve for the different algorithms, which is logical since they contain  $\log(P)$  in their expressions; where  $P$  is the number of processors. For instance, the time of execution of Binomial Tree is approximated by the following expression:

$$time = \log(P) \cdot (\alpha + \beta n); \text{ where}$$

$$\begin{cases} P = \text{Number of processors} \\ n = \text{Size of message} \\ \alpha = \text{Startup time (adding header, establishing connections ...)} \\ \beta = \text{Transfer time (seconds per words)} \end{cases}$$

In addition, it is obvious that the curve will converge since *time* will take the value of  $n \times \text{constant}$ , because *n* is too big.

Besides, we can confirm from this expression that Binomial Tree gives better results when the message is shorter, because *time* is linear with *n*. In fact, it can be seen from the chart above that it reaches 9ms in execution for short messages, however, it exceeds 200ms for longer messages.

## V) Conclusion

Let's note that Binomial Tree, like any broadcasting algorithm which sends *m* as a whole, will need time  $O(\log P (\alpha + \beta n))$ . One of the solutions is that, we should not immediately transfer the entire message. Rather, we should chop the message into smaller packets, which are sent independently. This way, broadcasting can spread its activity much faster by reinterpreting message *m* as an array of *k* packets of size  $\frac{n}{k}$ , using pipelines.