

# HPC Final Project

Valentinis Alessio [SM3800008]

27/02/2024

## Contents

<b>I</b>	<b>Exercise 1</b>	<b>1</b>
<b>1</b>	<b>Broadcast operation</b>	<b>2</b>
1.1	Data collecting process . . . . .	2
1.2	Latency analysis . . . . .	2
1.2.1	Fix algorithm and vary allocation . . . . .	3
1.2.2	Compare algorithms fixing allocation . . . . .	4
1.2.3	Fixing allocation and vary message size . . . . .	5
1.3	Broadcast performance model . . . . .	5
<b>2</b>	<b>Barrier operation</b>	<b>5</b>
2.1	Data collecting process . . . . .	6
2.2	Latency analysis . . . . .	6
2.2.1	Vary allocation for fixed algorithm . . . . .	6
2.2.2	Compare algorithms fixing allocation . . . . .	8
2.3	Barrier performance model . . . . .	8

## Part I

### Exercise 1

The goal of the exercise is to estimate the latency of default openMPI implementation of two collective blocking algorithms, one of which being *broadcast*, varying the virtual topology with which the algorithms operate, along with the map by which the cores are allocated. My choice resided into:

- **Broadcast:** varying virtual topologies among *default*, *basic linear*, *flat chain* and *binary tree*.
- **Barrier:** varying virtual topologies among *default*, *basic linear*, *double ring* and *bruck*.

The benchmark library used is OSU microbenchmark, available [here](#). The tests were conducted on the EPYC nodes in the ORFEO clusters and all tests were performed following a *bash* script to automatize the process of data gathering.

# 1 Broadcast operation

MPIBroadcast is a fundamental collective communication operation provided by the Message Passing Interface (MPI) library. It allows one process, typically referred to as the root process, to efficiently broadcast data to all other processes in a communicator. This operation is crucial for distributing information across multiple processes in parallel computing environments. It's a widely used one-to-all operation that, by default is implemented in multiple ways, in order to optimize the process of sending data to multiple processes. We will go into some further details only of those operations which are covered in this study.

- Default: this kind of implementation should automatically choose the optimal way of sending the information, based on both size of the message and number of cores.
- Basic linear: it's the way we could think to implement the broadcast algorithm, i.e. the root process sends sequentially to all processes the message, one by one.
- Chain: In this implementation, as the name suggests, the message is sent from one process to the other, so process0 sends its message to process1, process1 to process2, and so on and so forth.
- Binary tree: as the name suggests, this virtual topology of the cores is such that they are places in a binary tree fashion, so process0 is in charge of sending the message to process1 and process2, process1 to process3 and process4, and so on.

For some of these operations message segmentation is inabled, so the message is divided in chunks, and these chunks are sequentially sent to the other processes. In particular we are referring to binary tree and chain.

## 1.1 Data collecting process

The process of collecting data has been authomatized through the use of a bash script, available in the GitHub [repository](#). To have a better view of the what happens with different mappings of the cores, the same gathering process has been brought on varying the `-map-by` parameter, and to collect as many data in one shot, the maximum message has been truncated to 2024 bytes, which is anyway well beyond the latency-dominated size region. To allow for a better data gathering process, the different mapping of the cores (by core, by socket and by node), the different scripts have been separated.

As previously announced, the tests have been brought up into the EPYC nodes, selecting and prioritizing the acces to two whole nodes. So, due to the characteristics of the computing architecture, I have conduced the test onto a maximum of 256 cores, distributed in four sockets and two nodes.

To have better quality of the data collected, I chose to set 100 warmup operation and 10000 effective operations, from which the average has been taken and recorded.

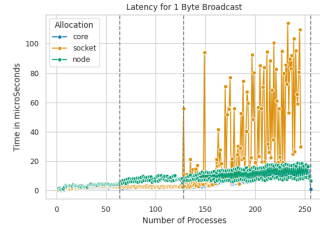
## 1.2 Latency analysis

The data collection process generated a pretty substantial dataset, providing different combinations for algorithm, cores allocation and message size to analyze the average latency of the communication. To tackle this complexity and various combinations, my effort was to sequentially fix some degrees of freedom, while varying the others, one or more at the time. At the end I tried to develop

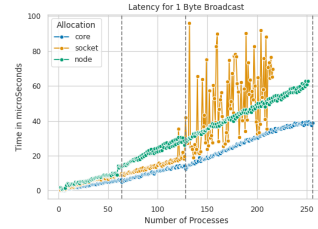
a model that could explain the interaction between the various covariates and the Average Latency measurements.

### 1.2.1 Fix algorithm and vary allocation

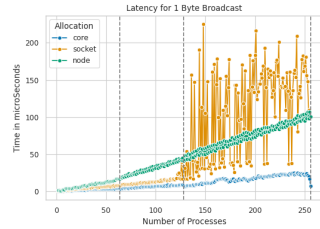
In this part we explore the behavior of the various algorithm, for data size of 1 MPI\_CHAR, so 1 byte, by varying the number of cores and cores allocation. This choose of the message size isolates and examines the "pure" latency of the communication.



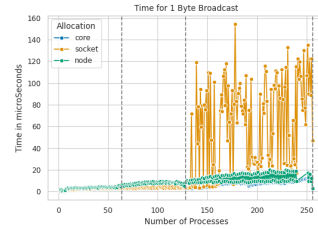
(a) Average Latency vs  $n$ . processes in default algorithm



(b) Average Latency vs  $n$ . processes in linear algorithm



(c) Average Latency vs  $n$ . processes in chain algorithm



(d) Average Latency vs  $n$ . processes in binary tree algorithm

Concerning the first plot, default algorithm behaves as expected, as we can clearly identify three regions, above all in core and socket mappings, with two noticable jumps and changing in behavior, while node allocation doesn't show remarkable and noticable jumps. The two jumps happen almost where expected, so at 64 cores, when we change socket, and at 128 cores, when we completely change node. It's worth to notice that the jump at the change of socket is by far more accentuated in the core mapping, as in that scenario cores are allocated as close as possible in the same socket.

Passing at the second plot, regarding the basic linear algorithm, we can observe almost the same behaviour at least for the core and node mapping. In socket mapping we can see that the change of behavior doesn't happen exactly at 128 cores as expected, suggesting some more complicated model beyond "naive" node/socket changes.

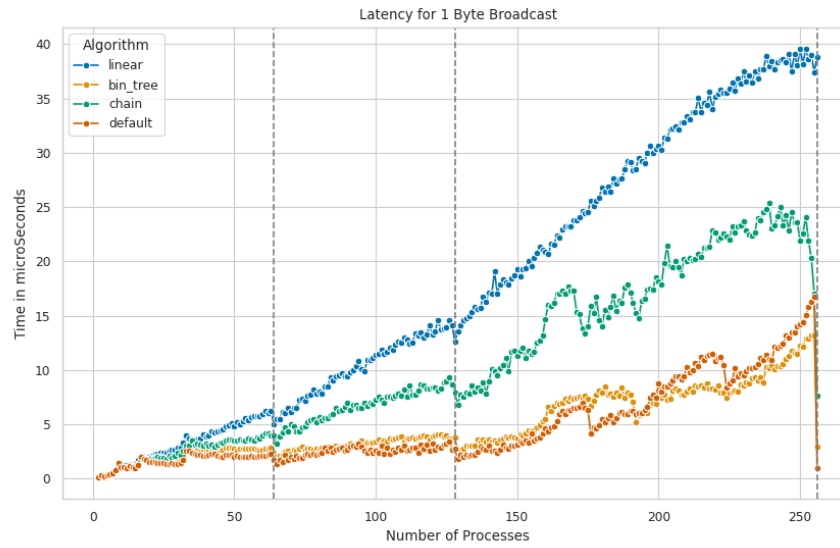
Going to the chain algorithm some similar observations to the previous ones hold, but with some peculiarities. In fact, differently from the previous ones, we can see the "serial" nature of the communication, as the different positioning of the processes deeply impact on the latency measurements, with the node allocation by far diverging from the other two, at least in the one-node situation.

Lastly, going to the binary tree algorithm, the lower latency measurements underscore and emphasize its (expected) superiority compared to the other algorithms. Moreover, we can see that allocating processes by node, we obtain some more stable latency when surpassing the 128 cores, showing some kind of optimized communication when dealing with processes inside the same node. Allocating by core or socket, instead, shows some jumps followed by immediate decreases in latency corresponding to the "region change" thresholds, which explanation may reside inside

the hierarchical nature of the algorithm. In fact these phenomena could be due to the transition between two different levels of the tree, that may introduce some delay in the communication. Once this transition is passed, the latency decreases, as the subsequent level of the tree gets filled.

### 1.2.2 Compare algorithms fixing allocation

In this second part of our analysis, we fixed the allocation type to *core*, as it shows all the expected changes in behavior of the algorithm, and is the less influenced by eventual inter-node traffic with other communication. Fixing this degree of freedom, we chose once more to analyze the 1 byte scenario, to isolate the pure latency of the communication, and we vary number of cores and type of algorithm.



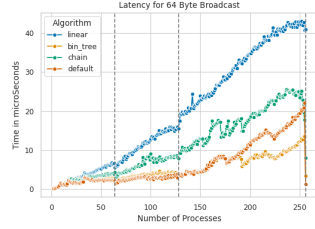
**Figure 2:** *Latency vs. n. processes by algorithm*

The figure above offer some insights into how the algorithm influences across three main regions:

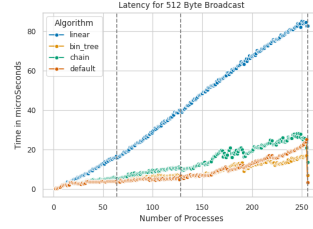
- **Within 64 cores:** in this intra-socket region we can observe almost identical performances, which can be indicative of some optimal speed of communication occurring between cores in the same socket. The slight advantage observed with respect to the default and binary tree algorithm may be due to the fact that the first selects the send policy automatically, while the latter organizes in a more optimal way the hierarchy of communication.
- **From 64 to 128 cores:** in this intra-node region we can begin to observe some more consistent advantage in terms of latency of the two algorithm mentioned above, which suggests that in this situation a choice among implementation may be done.
- **Above 128 core:** in this region we enter inter-node communication, and we can clearly state that the performance of the linear algorithm gets worse and worse, suggesting that with more processes to manage, the binary tree algorithm gains more than something in terms of performance, as the communication is done by a hierarchy processes, and not by one to all.

### 1.2.3 Fixing allocation and vary message size

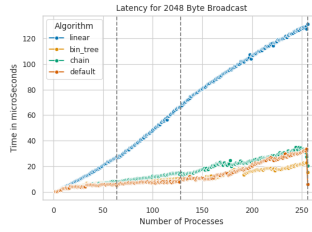
In this section we plan to analyze the various performance models among different message size.



(a) Average Latency vs  $n$ . processes for 64 byte message



(b) Average Latency vs  $n$ . processes in 512 byte message



(c) Average Latency vs  $n$ . processes in 2048 byte message

Varying message sizes we can observe more peculiar aspects of each implementations: while with a 1 byte message, the difference between implementations is pretty negligible as long we remain into the same node, varying message sizes, we can clearly see how the linear algorithm performs worse and worse increasing the message size. This behavior is due to two main factors: firstly the communication between processes is put as responsibility of the root process, which one by one sends the message to the other processes, while in all other algorithms the communication is organized in a more or less hierarchical way; secondly the linear approach is the only one among the analyzed ones that doesn't imply message segmentation, so in the linear algorithm the message is sent entirely from the root process to the remaining ones. The other methods, on the other hand, aside from the hierarchy of communications, allow for message segmentation, i.e. the message is divided in chunks and sent one chunk at the time via buffer-send (MPI\_Isend), as mentioned in the article ([Here put link to article in the github repo of HPC](#)).

## 1.3 Broadcast performance model

As a conclusive step of the analysis I attempted to formulate a performance model to try to better understand the not-so-straight-forward dynamics playing into these algorithms. At first I gave a try at estimating both latency and bandwidth within point-to-point communications employing again the OSU-microbenchmark library. The intention was to try and develop the Hockney model I found explained in some articles. However, the results obtained didn't fit well with the collected data, suggesting me to try something different. Maybe biased by my mathematical background, or by some freshly sustained exams, I opted for the implementation of a linear model.

## 2 Barrier operation

In this second part of the exercise, I chose to analyze and study the performance model of the MPI\_Barrier algorithm. It's an operation that allows for the synchronization of different process

within the same communicator. It consists in a construct that guarantees that by the end of the operation, all the processes have at least entered the operation. This collective blocking operation becomes crucial when we want to synchronize the processes involved in some parts of our distributed program to ensure coherent and predictable execution. Among the plethora of synchronization mechanisms, the barrier algorithm stands as a fundamental construct, enabling synchronization points within parallel programs. By default, the library OpenMPI offers different implementation of this algorithm, always characterized by peculiar virtual topology of the cores, or by different hierarchy of the execution. In this study I have examined only four of the several algorithms present.

- Default: this kind of implementation chooses automatically the optimal way to synchronize the processes, based on their number and allocation.
- Linear: in this algorithm all nodes refer to a preselected root; once everyone has reported to the root, it sends a release message to everyone.
- Double Ring: in this algorithm, a zero-byte message is sent from a preselected root to its right circularly; a node can leave Barrier once it receives the message for the second time.
- Bruck algorithm: this implementation, alternatively from the two just described, which require  $P$  communication steps, only requires  $\lceil \log_2 P \rceil$ . At a generic step  $k$ , process  $r$  receives zero-byte message from and sends the same message to process  $r - 2^k$  and  $r + 2^k$  respectively. This process goes on until each process have received the message from all other processes.

## 2.1 Data collecting process

The process of collecting data, similarly to the first part, has been automatized through the use of a bash script. To ensure a better all around view of the process, the same gathering process has been brought up on varying the *-map-by* parameter. To allow the generation of a more complete dataset, the different scripts are divided other than by algorithm, also by core mapping.

As before, the tests have been conducted into the EPYC nodes, selecting two whole computing nodes fully reserved for this task.

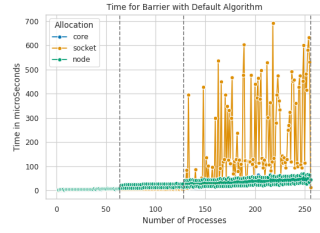
Similarly of what happened in the first part, to have more consistent and less biased data, I chose to select 100 warm up operations and 10000 effective measurements, from which the average has been taken and recorded.

## 2.2 Latency analysis

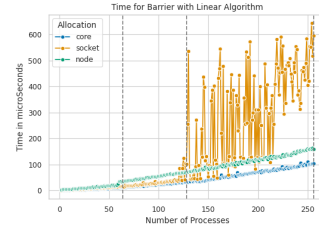
The data collection process generated a complete dataset, that can in principle allow for a quite extensive analysis. Similarly to the first part, my effort was to select and fix some degrees of freedom, while varying the others, and permitting for a simpler visualization of the data. At the end I tried to formulate a model that could explain the complex interaction between the various parameters.

### 2.2.1 Vary allocation for fixed algorithm

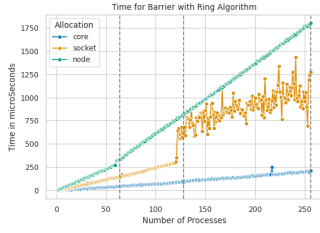
In this part we are going to explore the behaviour of the various algorithms, by varying number of processes and core allocation.



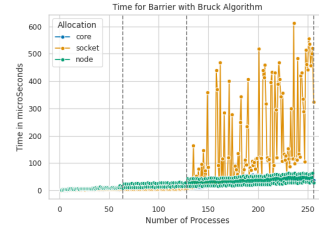
(a) Average Latency vs  $n$ . processes in default algorithm



(b) Average Latency vs  $n$ . processes in linear algorithm



(c) Average Latency vs  $n$ . processes in double ring algorithm



(d) Average Latency vs  $n$ . processes in bruck algorithm

Concerning the first plot we can clearly identify three different regions, one with less than 64 cores, where the three allocations seem to encounter the same latency, a second up to 128 cores, where node allocations shows some worse behavior, and a third, when we exceed the number of processes into one node, where the socket allocation goes crazy while node distribution gains something with respect to core one. It's worth notice that, contrary to the broadcast algorithm, we cannot notice remarkable jumps in latency when we pass through some thresholds, as we are dealing with a zero-sized message communication among all processes.

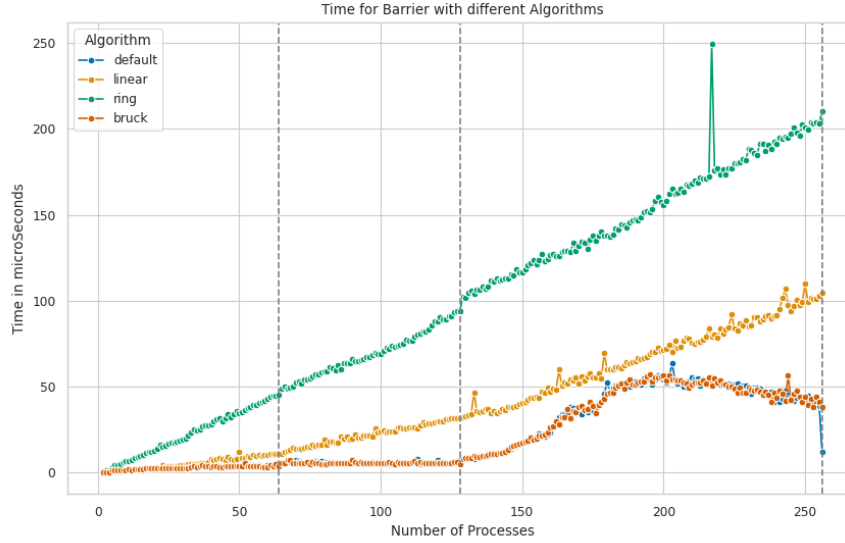
Passing to the second plot, regarding the linear algorithm, we can see almost the same characteristics as before, just with some more regular pattern into the node allocation. We can see that the changes in behavior don't happen where expected, suggesting some more complex interaction than the simple node/socket changes.

Going to the Double Ring algorithm, we can see some pretty worse behavior in general, with higher values of the latency, and with clear separation between the three types of allocations. This may be due to the "serial" nature of the communication, where process0 waits for process1, which waits for process2, and so on and so forth until each process receives the message for the second time. This reflects in an almost linear behavior in all the types of allocation, with the node distribution being from the beginning the worst one.

Lastly, passing to the bruck algorithm, we can see some similar behavior to the default algorithm, with the same region division and almost the same behavior, suggesting that this algorithm may have an optimal hierarchy of communication, making it suitable for the default implementation.

### 2.2.2 Compare algorithms fixing allocation

In this second part of our analysis, we fixed the core allocation to *core*, which showed to be more stable, other than being the least influenced by inter-node traffic.



**Figure 5:** *Latency vs n. processes by algorithm*

The figure above illustrates the four algorithms analyzed and offers some points of analysis straight away:

- Linear behaviors: we can clearly see as the double ring and linear algorithms have a quite consistent linear trend, justifying the linear number of communication required to complete the operation.
- Logarithmic trends: we can also see as the default and bruck implementation behave almost in the same way, suggesting that the latter is preferred. To justify the trend of the latency measurements, we have to take into consideration the bruck algorithm, where each process has to manage  $\log_2 P$  Send operations. So within the same node we can see almost clearly the logarithmic behavior of the latency, while whe trepassing the node limit, due also to the inter-node latency in the communication, we have a "jump", and the same logarithmic trend reestablish when we have enough cores into the second node.

## 2.3 Barrier performance model

Similar to the first part of the exercise, in this section my aim is to develop a statistical model that can explain the latency measurements as a function of the others degrees of freedom.