MPI collective operations
High Performance Computing

Alessandro Della Siega
University of Trieste

May 2024

1 Introduction

The OpenMPI library implements several algorithms to perform collective blocking operations according to many different parameters. OSU Micro-Benchmark is a tool that allows us to evaluate the performance of these operations.

1.1 Computational architecture

In order to test the performance of the broadcast and reduce operations, we will use a high-performance computing cluster. The computational resources used to complete the assignment are the ones provided by the ORFEO cluster.

The ORFEO system architecture consists of different machines architecture, in this project the THIN partition is used. The THIN partition consist of 12 Intel nodes: two equipped with Xeon Gold 6154 and 10 equipped with Xeon Gold 6126 cpus.

1.2 OSU Micro-Benchmark

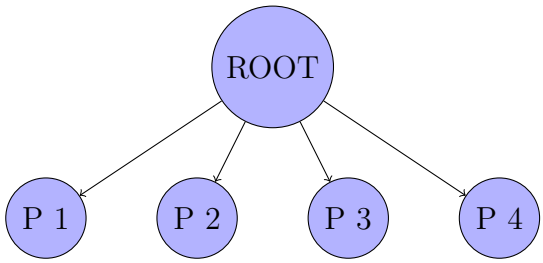
OMB includes benchmarks for various MPI blocking collective operations (Allgather, Alltoall,Allreduce, Barrier, Bcast, Gather, Reduce, Reduce-Scatter, Scatter and vector collectives). These benchmarks work in the following manner. Suppose users run the osu Bcast benchmark with N processes, the benchmark measures the min, max and the average latency of the MPI_Bcast collective operation across N processes, for various message lengths, over a large number of iterations. In our experiments, we will consider the average latency for each message length.

In this report, two operations are take into consideration: broadcast and reduce.

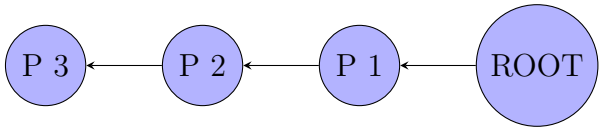
1.3 Broadcast

The broadcast operation `MPI_Bcast` is a one-to-all communication operation which allows a process (typically the root process) to distribute information across many processes. This operation is implemented in OpenMP using different algorithms. The algorithms that are considered in this analysis are:

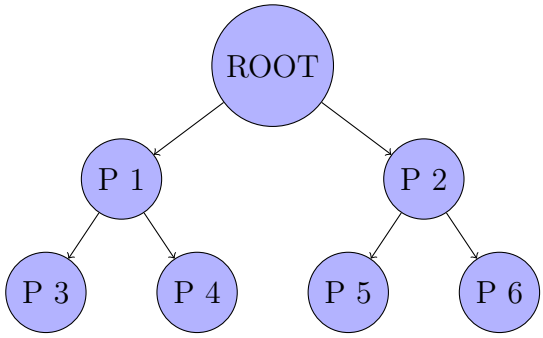
- **basic linear**, the root process sends to all the other process the message without segmentation;



- **chain**, the message is split into segments and transmission of segments continues in a pipeline until the last node gets the broadcast message. ith process receives the message from the $(i-1)$-th process, and sends it to $(i+1)$-th process;



- **split binary tree**, suppose that each process is a node of a binary tree and the root node is the root process, each parent node sends the message to its children. As the name of the algorithm suggests, the message is split in two halves before transmission and when each half of the message reaches its destination, it is necessary to merge the two halves.



1.4 Reduce

The reduce operation `MPI_Reduce` combines the elements provided in the input buffer of each process in the grouping the specified operation, and returns the combined value in the output buffer of the root process. The specific operation used to reduce the input in the OSU benchmark are sum, maximum, and minimum.

The algorithms that are considered in this analysis are:

- **basic linear** , the root process receives the message from all the other process and reduces the messages;
- **chain**, if we order the processes with respect to their rank, each process sends the message to the next process, each process receives the message from the previous process and reduces the messages;
- **binary tree** , suppose that each process is a node of a binary tree and the root node is the root process. Each parent node receives the message from its children and reduces the received messages passing the result to its parent node.

2 Experimental setup

In this section we discuss how the OSU benchmark tool is used to evaluate the performance of the broadcast and reduce operations.

For each operation, we vary the following parameters:

- **algorithm**, the algorithm used to perform the colective operation;
- **message size**, the size of the message that is sent. It is measured in `MPI_CHAR` unit (1 byte). We will range from 2^1 byte to 2^{20}. We choose this range in order to saturate the osu benchmark capabilities. In fact the maximum message size that can be sent is $1MB$ which is equivalent to 2^{20} bytes.
- **number of processes**, the number of processes that are involved in the operation. This parameter will range from 2 to 48 by step 2. This choice is due to the fact that, according to the THIN architecture, the maximum number of processes that can be used is 48;
- **map by**, the mapping of the processes to the hardware. We will consider the following options: `socket`, `core`;

The result is the latency and it is measured in microseconds.

For each run we set 10^4 repetitions and we take the average of the time taken to perform the operation. We also set a warm-up phase of 10^4 repetitions, in order to avoid the cache overhead of the first runs.

3 Results

We have many parameters to take into account when we evaluate the performance of the broadcast and reduce operations. In order to make the analysis easier, we will fix some of them and vary the others.

3.1 Broadcast

By fixing the message size, we notice that the latency scales in a complex way with the number of processes. In fact, it is interesting to notice, as show in Figure 1, that there is a ramp up in the latency when we reach 16 processes and 32 processes. This behaviour has no clear explanation.

When we set a large message size, Figure 2, a substantial positive performance is observed for the binary tree algorithm. This algorithm takes advantage of the fact that the message is divided into smaller parts and sent to the processes in a binary tree fashion.

The linear algorithm, on the other hand, is the algorithm that scales worse with respect to the message size. This is not surprising as it sends the message to all processes one by one.

The mapping does not seem to have a significant impact on the performance of the broadcast operation. The performance is almost the same for all the mappings. An exception is the chain algorithm. Generally, it performs better using the map-by-core policy for the processes.

In the end, if we set the number of processes and vary the message size, we can see that the latency depends linearly on the message size. The binary tree algorithm has a better performance than the linear algorithm, as we can see in Figure 4. We can see a spike in the latency in Figure **??** when we reach a message size of approximately 0.2 megabyte for the Binary Tree algorithm. This behaviour is not clear and needs further investigation.

3.2 Reduce

Let us start by setting the message size and varying the number of processes. For the reduce operation, it is interesting that the chain algorithm is the one that works worse for 2 bytes and 1024 bytes message size, as shown in Figure 5 and Figure 6. This could be explained by the fact that the chain algorithm needs to perform more operations than the other algorithms (one reduction per node) and this is a bottleneck if the size of the message is small.

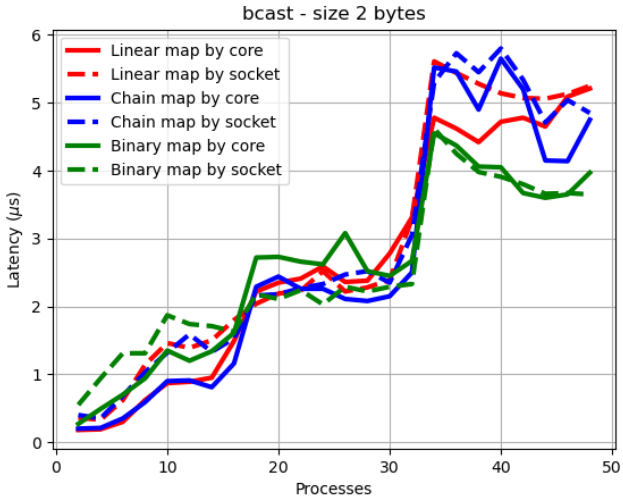


Figure 1: Broadcast 2 bytes fixed message size

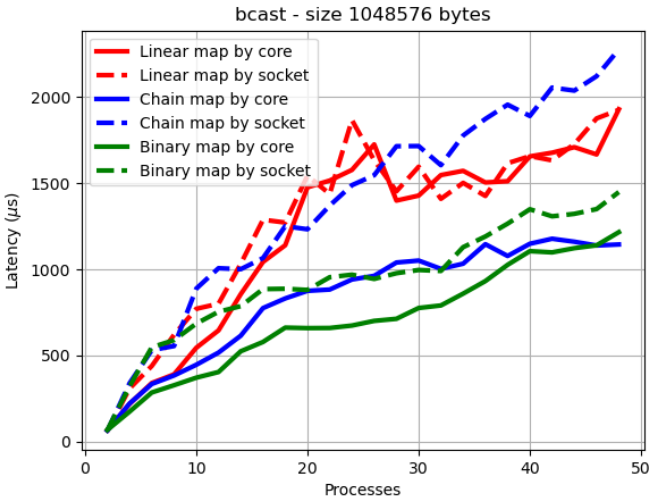


Figure 2: Broadcast 1 megabyte fixed message size

For what concernes small messages, the linear algorithm performs better than the chain algorithm.

When we set a large message size, Figure 7, the latency scales linearly

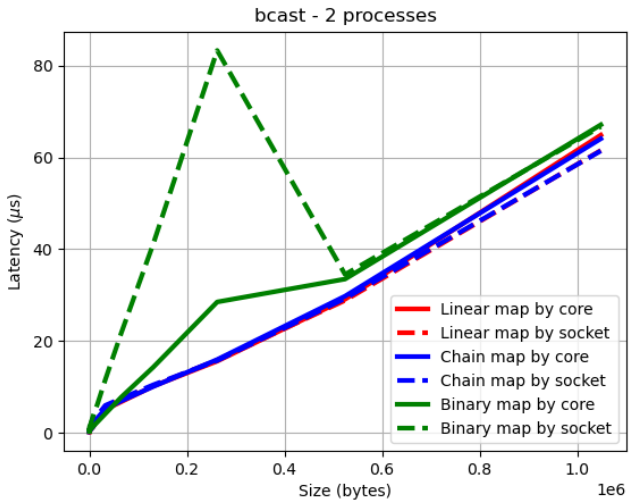


Figure 3: Broadcast with 2 processes

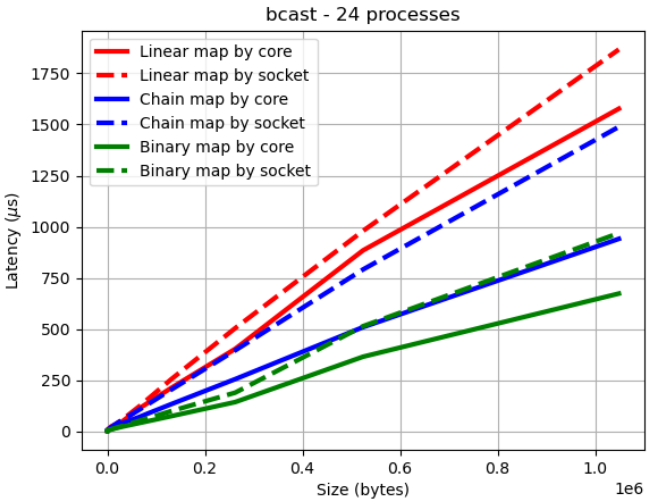


Figure 4: Broadcast with 24 processes

with the number of processes. Also in this case, by fixing the size of the message, the latency is proportional with the number of processes.

As the message size increases, the mapping has no significant impact

on the performance of the reduce operation performed by linear and binary algorithm. However, the chain algorithm performs significantly better using map by node policy for the processes, as depicted in Figure 7. For large messages, the binary tree algorithm is to be preferred.

In the end, if we set the number of processes and vary the message size, we can see that the latency depends linearly on the message size. The binary tree algorithm has a better performance than the linear algorithm, as we can see in Figure 9. Also in this case, as expected, the tree algorithm and the chain algorithm perform better than the linear algorithm.

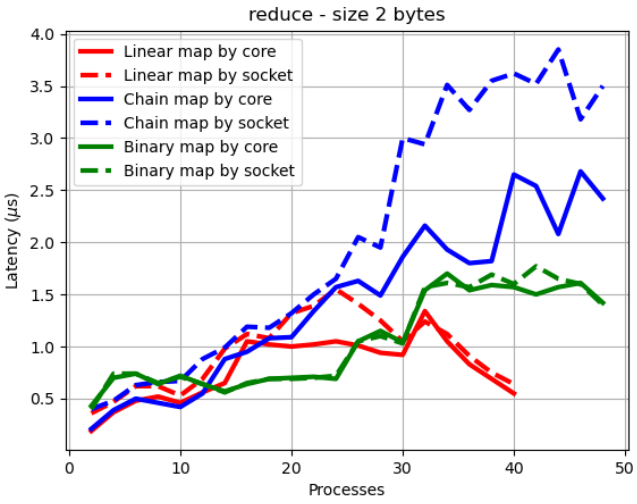


Figure 5: Reduce 2 bytes fixed message size

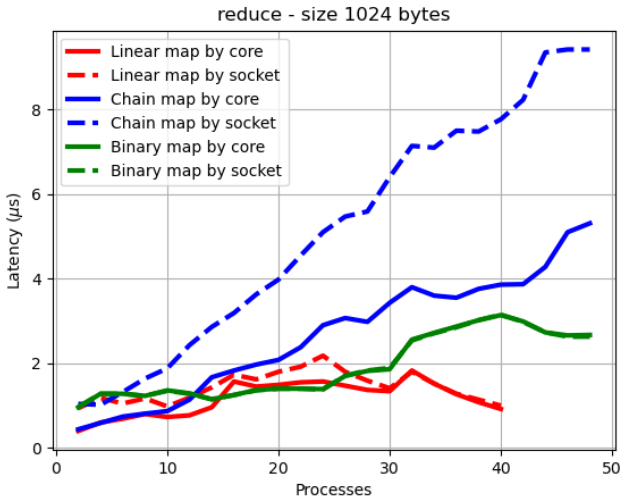


Figure 6: Reduce 1024 bytes fixed message size

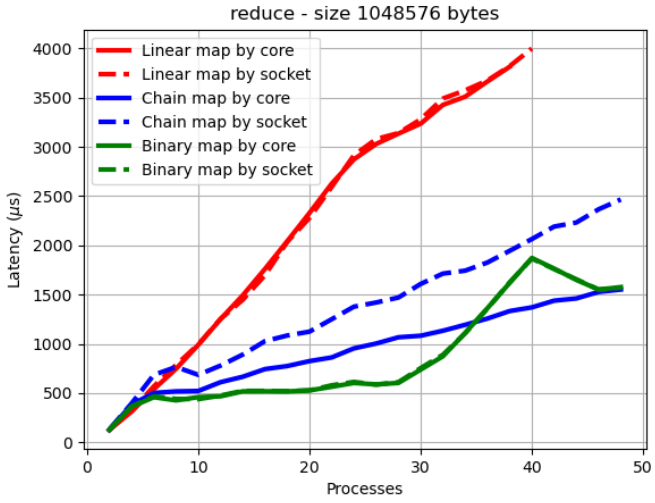


Figure 7: Reduce 1 megabyte fixed message size

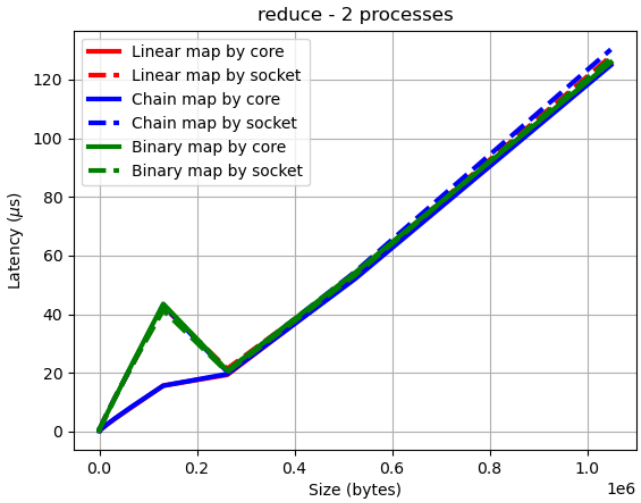


Figure 8: Reduce with 2 processes

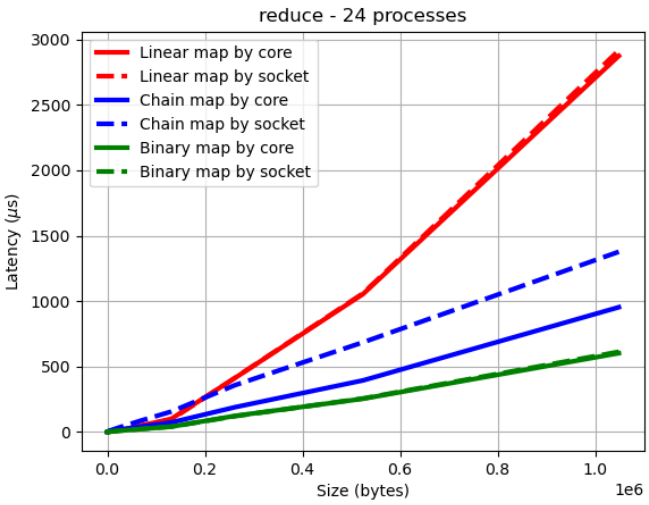


Figure 9: Reduce with 24 processes

4 Linear model

We have seen that the broadcast and reduce operations have different behaviours depending on the number of processes and the message size. In order to explain better the results, we will introduce a model that can help us in getting a better understanding of the performance of these operations.

Let's consider the broadcast operation with linear algorithm and mapping of the processes by core.

The baseline model determined by the collected data is the following.

$$Latency = \beta_0 Processes + \beta_1 Size \tag{1}$$

Table 1: OLS Regression Results

	Dep. Variable: Latency					
	Coef.	Std. Err.	t	P > abs(t)	[0.025	0.975]
Processes	1.1264	0.235	4.798	0.000	0.665	1.588
Size	0.0011	0.000025	45.400	0.000	0.001	0.001

We do not include a intercept in the model since a message of 0 byte does not make sense. The R^2 achieved by the baseline model is 0.84 and the AIC is 6096.

By applying some transformations to the data, we can improve the model. Let's consider the following improved model.

$$\log_2 Latency = \beta_0 Processes + \beta_1 \log_2 Size \tag{2}$$

Table 2: OLS Regression Results

	Dep. Variable: log2_Latency					
	Coef.	Std. Err.	t	P > abs(t)	[0.025	0.975]
Processes	0.0274	0.0040	6.693	0.000	0.019	0.035
log2_Size	0.3256	0.0098	33.346	0.000	0.306	0.345

The improved model, is characterized by a R^2 of 0.88 and the AIC is 1842. The second model is able to explain the data better than the baseline model. The R^2 is higher and the AIC is lower.

This model allow us to understand the behaviour of the broadcast operation with linear algorithm and mapping of the processes by core. The interesting results are the following:

- an increase of one unit in the number of the processes, leads to and increase of $2^{0.0274} \approx 1.01$ in the latency;
- an increase of one unit in the message size, leads to an increase of 0.3256 in the latency.

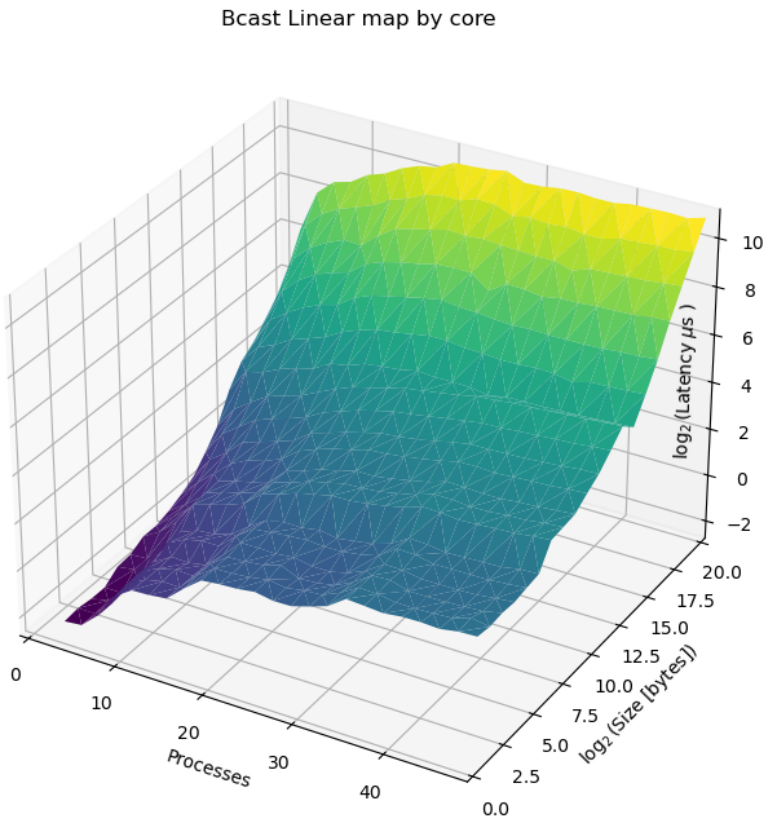


Figure 10: 3D plot of the broadcast operation with linear algorithm and mapping of the processes by core. The plot shows the relationship between the number of processes, the logartihm of message size and the logarithm of the latency.

5 Conclusions

This analysis of MPI collective operations, focusing on broadcast and reduce, highlights key performance insights influenced by various algorithms and configurations.

For broadcast operations, latency shows a complex relationship with the number of processes when the message size is fixed, notably increasing at 16 and 32 processes without a clear explanation. The binary tree algorithm performs best with large messages due to its efficient distribution method, while the linear algorithm scales poorly as it sequentially sends messages to each process. Mapping policy generally has minimal impact, except for the chain algorithm, which benefits from the map by core policy.

In reduce operations, the chain algorithm struggles with small messages due to its high number of reduction operations per node, whereas the linear algorithm performs better in these cases. With large messages, latency scales linearly with the number of processes, and the binary tree algorithm again shows superior performance. Mapping has little impact on large messages for linear and binary algorithms, but the chain algorithm performs better with the map by node policy.

Overall, latency scales linearly with message size when the number of processes is fixed. The binary tree algorithm consistently offers the best performance for both broadcast and reduce operations. The linear model used in the analysis helped clarify the dependencies between latency, message size, and the number of processes, supporting these findings. These insights can guide optimization and tuning of parallel applications using MPI.