Annalisa Paladino
SM3800021

High Performance Computing 2023 – Exercise 1

Introduction

The OpenMPI library is an open-source implementation of the Message Passing Interface (MPI) standard, widely used for parallel programming in high-performance computing. It provides a set of software tools and libraries that facilitate efficient communication and coordination among multiple processes running on one or more machines in a network, typically in a cluster computing environment. Among its capabilities, openMPI includes several algorithms for executing collective operations, essential in parallel computing applications. These operations facilitate data exchange and synchronization between processes through algorithms dependent on point-to-point communications.

This project [1] aims to evaluate the performance of various openMPI algorithms for the two collective operations *broadcast* and *scatter*. The goal is to estimate the latency of default openMPI implementation, varying the number of processes and the size of the messages exchanged and then compare it with values obtained through the selection of different algorithms. To perform those analysis, the OSU Benchmark [2] has been used.

Set up

To assess the performance of broadcast and scatter operations on the ORFEO [3] cluster, two THIN nodes have been used, for a total of 24 cores (every THIN node has 2x12 cores), the number of processes considered varies from 2 to 48 and the size of the message from 2 byte to 1 MB. The processes were distributed evenly across the two nodes, to ensure that the latency measured always considers inter node communications.

All tests were performed following a bash script to automatize the process of data gathering:

- The data of the broadcast algorithm were collected using:
  ```
  mpirun --map-by core -np $processes --mca coll_tuned_use_dynamic_rules true --mca coll_tuned_bcast_algorithm 0 osu_bcast -m $size -x $repetitions -i $repetitions
  ```

- The data of the scatter algorithm were collected using:
  ```
  mpirun --map-by core -np $processes --mca coll_tuned_use_dynamic_rules true --mca coll_tuned_scatter_algorithm 0 osu_scatter -m $size -x $repetitions -i $repetitions
  ```

[1] https://github.com/Foundations-of-HPC/High-Performance-Computing-2023
[2] https://mvapich.cse.ohio-state.edu/benchmarks/
[3] https://orfeo-doc.areasciencepark.it

The `map-by` option is a feature that specify how processes should be mapped to the available hardware resources (`cores`, `sockets` or `nodes`) in a parallel computing environment; to decide the most appropriate map-by setup a few tests as been conducted by simply changing this option for both broadcast and scatter. For additional details regarding the scripts, consult the repository [4].

Broadcast

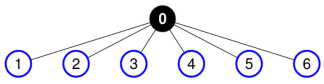
Broadcast is an essential collective communication operation within the Message Passing Interface (MPI) library. This operation enables a designated process, known as the root process, to efficiently distribute data to all other processes within a communicator. This operation is important for sharing information among numerous processes in parallel computing settings.

Broadcast algorithms

In this project, algorithms 0, 1, 2 and 4 will be examined:

- **Ignore** (bcast 0): It's the *baseline model*, by choosing this option the user doesn't specify a particular scatter algorithm, the default algorithm is chosen based on the size of the message, the number of processes and other internal factor of Open MPI

- **Basic Linear** (bcast 1): The algorithm employs a single level tree topology where the root node has $P - 1$ children. The message is transmitted to child nodes without segmentation.



- **Chain** (bcast 2): Each internal node in the topology has one child. 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** (bcast 4): This algorithm structures the message transmission in a binary tree format. The root process sends the message to two processes, and each internal process thereafter has two children. Before initiating communication, the message is divided in half. The left branch of the tree transmits the left half of the message, and similarly, the right branch transmits the right half. In the final step of

[4] https://github.com/annalisapaladino/High-Performance-Computing

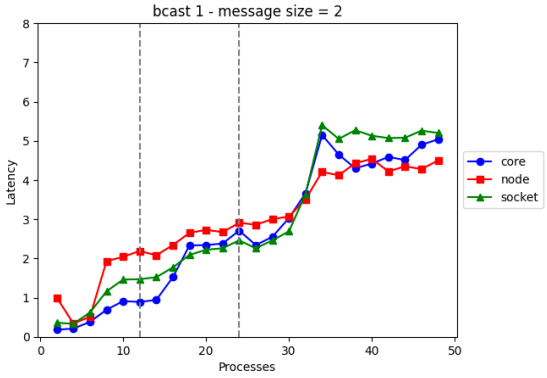
the operation, the right and left nodes exchange their respective halves of the message to complete the broadcast.

Broadcast latency

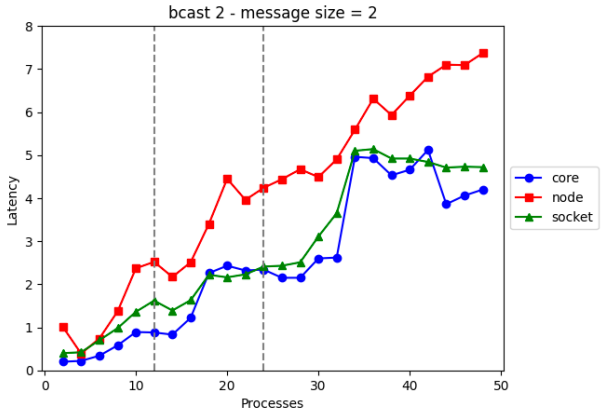
The data collection phase resulted in a substantial dataset, detailing latencies across various combinations of algorithms, process counts, process allocations, and message sizes. In the following analysis I tried to explore the interactions between variables in various way.

Fixing algorithm and compare processes allocation effects

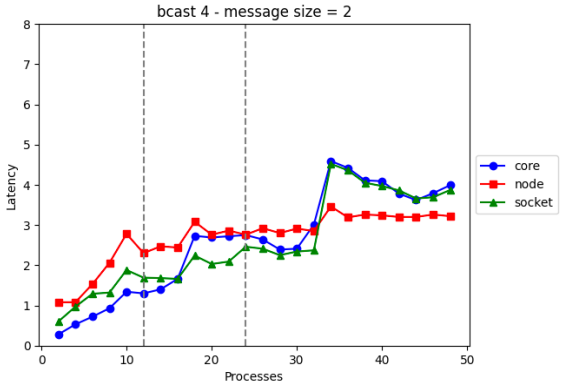
For each algorithm, I evaluated how process allocation affects operation latency. To ensure that my analysis remained independent of message size, I standardized it at 2 MPI_CHAR. This approach allowed me to isolate and scrutinize the pure latency intrinsic to the algorithms.



The behavior of the **Basic Linear** (bcast1) algorithm conforms to expectations. When switching nodes, both socket and core allocations exhibit noticeable increases in latency (in μs), whereas latency remains consistent during node allocations. This pattern is also evident in changes between sockets: latency increases are observed with core allocations but not with socket allocations. Notably, an interesting phenomenon is observed concerning latency spikes. Given the thin architecture configuration with 12 cores per socket and 24 cores per node, one would anticipate latency jumps at 12 and 24 processes. However, unexpectedly, these jumps occur at 16 and 32 processes, suggesting that factors other than simple node or socket transitions may influence communication dynamics.



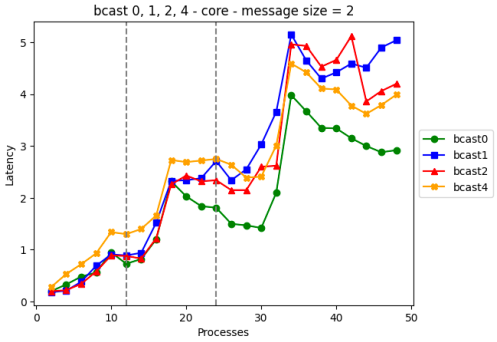
The **Chain** (bcast 2) algorithm is similar to the previous case, but it's possible to note a significant difference during node changes. In the linear algorithm scenario, when the process count exceeds 24, the three types of allocations - core, socket, and node - tend to converge. This convergence reflects the linear algorithm's structure, wherein messages originate from process 0 and are distributed to all others, maintaining uniform execution steps across allocations especially when processes extend across both nodes. Conversely, the chain algorithm exhibits a different pattern due to its sequential communication order among processes. Here, when processes are distributed by node, the increased message travel distance becomes evident compared to distributions by core or socket.



The **Split Binary Tree** (bcast 4) algorithm shows a better performance compared with the two algorithms seen above. Allocating processes by node reveals a stable latency profile after 24 cores (node change), indicative of optimized intra-node communication upon reaching a specific depth within the tree structure. When allocating by socket or by core, instead, an apparent jump followed by a decrease in latency is observed. This might be attributed to the hierarchical structure of the algorithm: the jump may correspond to the transition between different levels of the tree, introducing a slight delay in communication.

Fixing processes allocation and compare algorithms

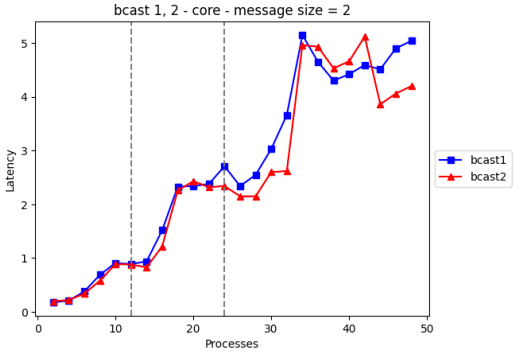
Now I fixed the process allocation to core and I assessed the algorithmic impact on the operation latency. The message size is again fixed to 2 MPI_CHAR.



As seen above, the binary tree algorithm demonstrates lower latency values between the 3 algorithms analyzed. This outcome aligns with theoretical expectations, as tree structures are known for efficiently distributing communication across their branches, thus improving overall performance. The default configuration surpasses all the algorithms, achieving the lowest latency. This result was anticipated:

default configurations are typically optimized to balance multiple use cases, ensuring robustness and efficiency across different environments. Furthermore, these settings may utilize platform-specific optimizations, leveraging system characteristics to boost communication efficiency.

Moreover, it's possible to notice that Basic Linear and Chain algorithms show similar behaviors, let's see them more in details:



- *Within Socket*: Both the linear and chain algorithms demonstrate nearly identical performance within this region. This indicates that intra-socket communication, which occurs between cores on the same socket, is sufficiently rapid that the specific algorithm used becomes nearly irrelevant to overall performance.

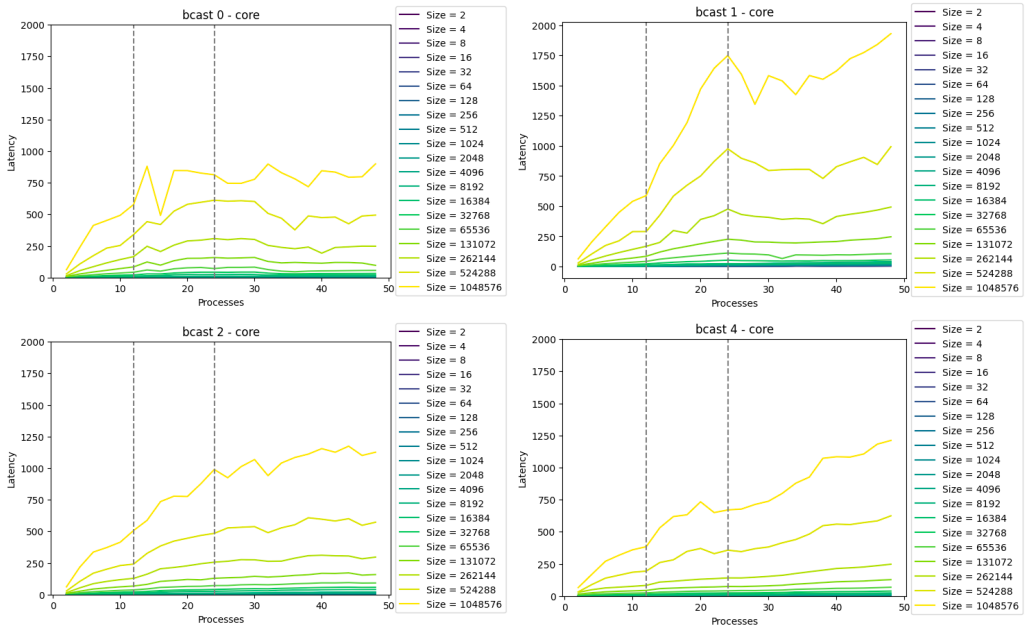
- *Within Node*: Transitioning to communications encompassing the entire node, a slight distinction between the algorithms emerges, suggesting a more subtle influence of the choice between chain and linear algorithms. Although the difference is minimal, the chain

algorithm exhibits a marginally better performance, highlighting a nuanced effect on intra-node communication.

- *Outside Node*: For communication outside the node, where latency is naturally higher, the differences between the chain and linear algorithms become more distinct. This observation is expected as the chain algorithm, which utilizes contiguous cores, contrasts sharply with the linear approach of broadcasting from rank 0 to all other ranks. This divergence illustrates how algorithm choice impacts performance more significantly in scenarios involving higher latency.

Fixing processes allocation and compare algorithms for different message size

When systematically varying the message size, Split Binary Tree (bcast 4) seems to outperform the other algorithms. While in the previous analysis, with a fixed message size of 2, Basic Linear (bcast1) and Chain (bcast 2) used to show very similar behavior, now with the increase of the message size we can see a real difference between the two algorithms, in fact now bcast 2 outperform bcast 1. A crucial factor contributing to its efficacy is the algorithm's ability to segment messages into chunks during transmission, a capability that the linear broadcast approach lacks. Consequently, as message size increases, the latency of the linear algorithm exhibits significant escalation, thereby highlighting the expanding performance gap in comparison to the chain algorithm.



Broadcast performance models

To end my analysis of broadcast I created performance models to better understand the behavior of the data. My initial approach involved estimating both latency and bandwidth within point-to-point communication routines by employing the OSU benchmark. The intention was to leverage these estimated values to construct a model analogous to the Hockney model. Unfortunately, the results I obtained did not align well with the collected data, which prompted me to explore alternative methodologies. Therefore, I decided to develop a model and test it for the various algorithms. In selecting predictors for the model,

I considered as a variable the number of processes and I maintained the message size fixed to 2 MPI_CHAR, while I chose to maintain a fixed process allocation using --map-by core.

The model produced for Basic Linear (bcast1) is:

$$\text{latency} = -0.233587 + 0.117043 \times \text{processes}$$

with an R^2 adjusted of 0.9381.

The model produced for Chain (bcast2) is:

$$\text{latency} = -0.214384 + 0.109159 \times \text{processes}$$

with an R^2 adjusted of 0.8548.

The model produced for Split Binary Tree (bcast4) is:

$$\text{latency} = -0.523 + 5.518 \times \text{processes} - 2.529 \times \text{processes}^2$$

with an R^2 adjusted of 0.8793.

Scatter

Scatter algorithms are designed to distribute data to all processes in a communicator, originating from a specified root. While similar to broadcasting, which sends the same data to all processes, scatter algorithms distribute distinct chunks of data to different processes.

Scatter algorithms

In this project, algorithms 0, 1, 2 and 3 will be examined:

- **Ignore** (scatter 0): It's the *baseline model*, by choosing this option the user doesn't specify a particular scatter algorithm, the default algorithm is chosen based on the size of the message, the number of processes and other internal factor of Open MPI

- **Basic linear** (scatter 1): Each process receives a contiguous block of data from the root process. The root process sends data to each process in sequence.

- **Binomial** (scatter 2): It uses a binary tree structure to distribute data. Each process receives data from a process that is logarithmically distant from the root process.

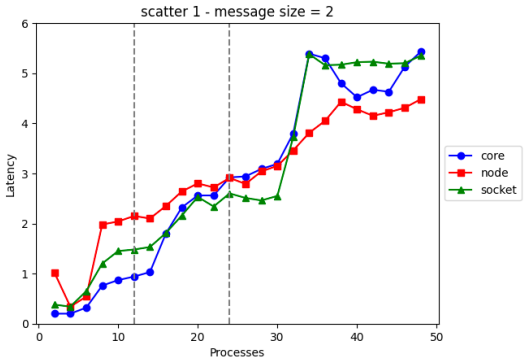
- **Linear nb** (scatter 3): In a non-blocking scatter, the scatter operations are executed asynchronously. This means that the MPI implementation does not pause for the completion of one scatter operation before initiating the next. Such an approach can enhance performance by enabling the system to concurrently manage communication and computation tasks, which is especially beneficial in systems experiencing high network latency.

Scatter latency

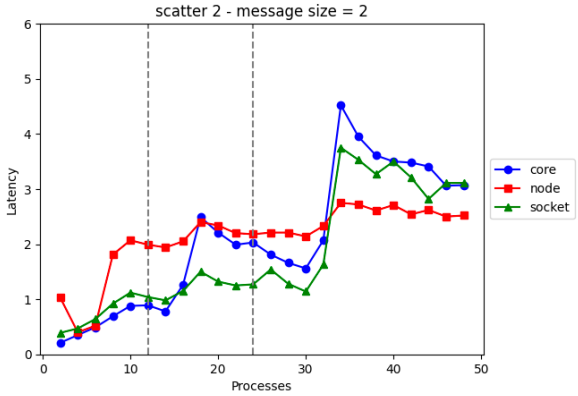
Fixing algorithm and compare processes allocation effects

For each algorithm, I evaluated how process allocation affects operation latency. To ensure that my analysis remained independent of message size, I standardized it at 2 MPI_CHAR.

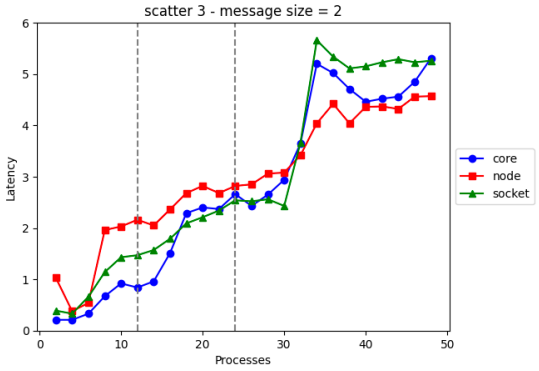
This approach allowed me to isolate and scrutinize the pure latency intrinsic to the algorithms.



In **scatter 1**, all three allocations show a steadily increasing latency (in μs) trend, core and socket allocations demonstrate similar patterns with gradual increases in latency, suggesting that the overhead of inter-core or inter-socket communication might be impactful. The latency increase in node allocation is more pronounced, peaking significantly around 30 processes before a drop. This indicates a critical threshold where node-spanning communication becomes highly inefficient.



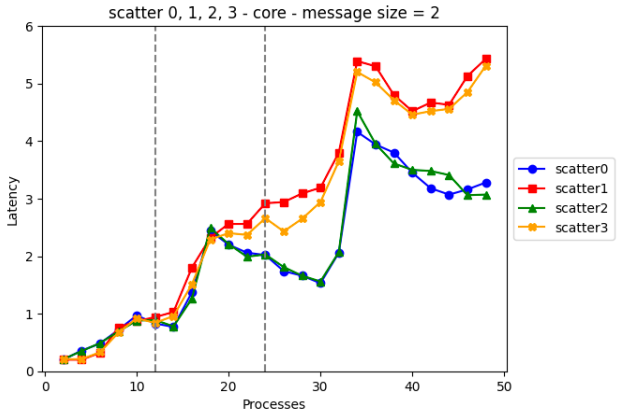
Scatter 2's latency profile reveals an interesting pattern where node and socket allocations appear closely tied. Core allocation latency shows significant volatility, peaking sharply around 20 processes and again at 30. This pattern could highlight specific inefficiencies or limits in core-to-core communication within the same or across different CPUs. Node and socket allocations show correlation in latency changes, suggesting that node-crossing impacts socket communication directly, possibly due to physical layout or communication protocol efficiencies. Moreover, the behavior of scatter 2 underscores slower latency values than the other two algorithms analyzed, showcasing their superior performance.



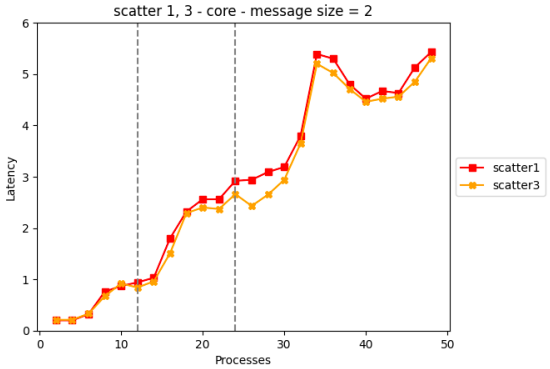
In **scatter 3,** core allocation shows high latency spikes, indicating potential challenges in managing core-level communication effectively in this scatter method. Both node and socket show a steady increase until 30 processes and then a subsequent drop. This suggests a scalability limit being reached, after which adding more processes does not significantly impact latency due to some form of optimization or saturation effect.

Fixing processes allocation and compare algorithms

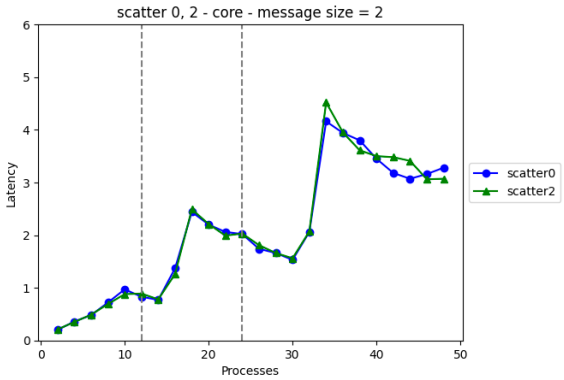
Now I fixed the process allocation to core and I assessed the algorithmic impact on the operation latency. The message size is again fixed to 2 MPI_CHAR.



As seen in the previous analysis scatter 2 show the lowest values of latency, it performs almost identically to the baseline model scatter 0, these two algorithms exhibit superior performance. Moreover, this plot shows that there are some similarities also between scatter 1 and 3, let's see them more in details.



- *Within Socket*: Both scatter 1 and scatter 3 demonstrate nearly identical performance within this region. This indicates that intra-socket communication, which occurs between cores on the same socket, is sufficiently rapid that the specific algorithm used becomes nearly irrelevant to overall performance.
- *Within Node*: Transitioning to communications encompassing the entire node, a slight distinction between the algorithms emerges, suggesting a more subtle influence of the choice between the two algorithms. Although the difference is minimal, scatter 4 exhibits a marginally better performance, highlighting a nuanced effect on intra-node communication.
- *Outside Node*: For communication outside the node, where latency is naturally higher, the differences between the two algorithms become more distinct.

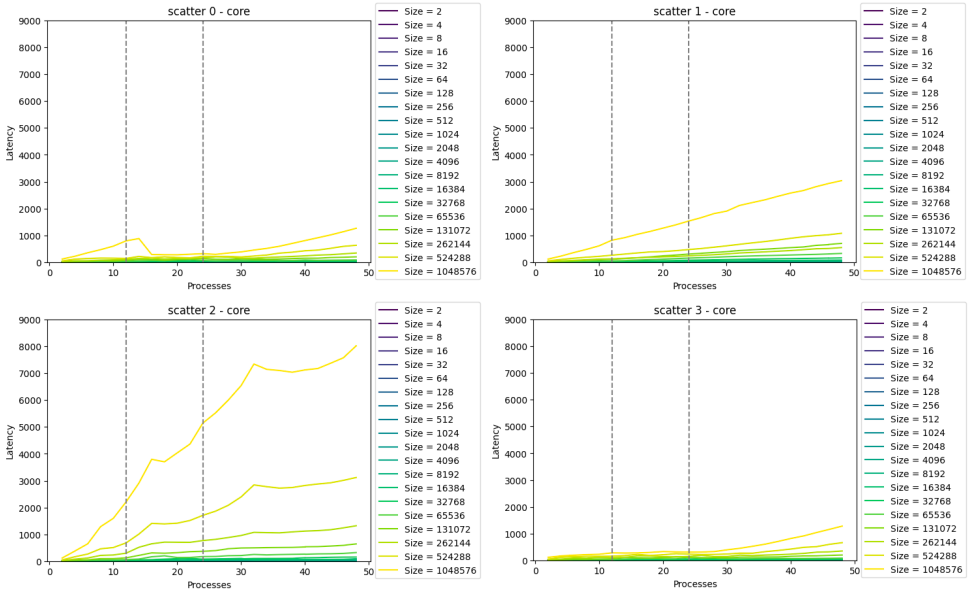


Both scatter 0 and scatter 2 demonstrate nearly identical performance in the *within Socket* and *within Node* regions. Also, in a part of the *outside Node* region the two algorithms show identical behaviour, only after 30-40 processes the differences between the two algorithms become more distinct. These two algorithms show a better performance than scatter 1 and 3.

Fixing processes allocation and compare algorithms for different message size

When systematically varying the message size, scatter 3 seems to outperform the other algorithms, the latencies recorded for this algorithm are very similar to those of the default algorithm, suggesting that the non-blocking linear algorithm could be advantageous. In comparison to the linear scatter algorithm, the most significant improvement with a non-blocking approach is observed with small message sizes and large numbers of processes.

While in the previous analysis, with a fixed message size of 2, scatter 2 was the most efficient algorithm by showing very similar behavior to the baseline model scatter 0, now considering bigger message sizes scatter 0 remains one of the best models found, while scatter 2 is without any doubt the worst one in terms of latency.



Scatter performance models

To end my analysis of scatter I created performance models to better understand the behavior of the data. In selecting predictors for the models, I considered as a variable the number of processes and I maintained the message size fixed to 2 MPI_CHAR, while I chose to maintain a fixed process allocation using `--map-by core`.

The model produced for Scatter 1 is:

$$\text{latency} = -0.215616 + 0.124241 \times \text{processes}$$

with an R^2 adjusted of 0.9355.

The model produced for Scatter 2 is:

$$\text{latency} = 0.141486 + 0.077624 \times \text{processes}$$

with an R^2 adjusted of 0.7515.

The model produced for Scatter 3 is:

$$\text{latency} = -0.278261 + 0.120580 \times \text{processes}$$

with an R^2 adjusted of 0.9317.