Final Project HPC - Exercise 1

Performance Evaluation of OpenMPI Collective Operations

Emanuele Ruoppolo
Università degli Studi Trieste
emanuele.ruoppolo@studenti.units.it

Abstract

The aim of this project is to assess the performances of the OpenMPI library's collective operations on the EPYC partition of the ORFEO cluster. Many optimizations options have been tested for two specific collective algorithms: broadcast and scatter. Particularly the results presented evaluate the performance of a four of their possible implementations.

1 Introduction

The main goal of this project is to evaluate the performances of the default OpenMPI implementations of the broadcast and scatter operations, particularly focusing on different algorithms, messages size and number of processes. To achieve this, we used the OSU benchmark, which is a set of MPI performance measurement tools. The tests were run on up to two nodes of the EPYC partition of the ORFEO cluster.

Specifically a single node of the partition provides 2 x AMD EPYC 7H12 processors, based on the "Rome" architecture, housing a total of 128 cores (2x64). Each processor contains eight Core Complex Dies (CCDs) and one I/O die (IOD). Sockets are connected with an infinity fabric that consists of three links at 16GT/s for a theoretical bandwidth of 96 GB/s. Each CCD contains two Core Complexes (CCXs), and Each CCX has four cores and 16 MB of L3 cache.

2 Experimental Setup

The tests were conducted using two EPYC nodes, meaning a total of 256 cores. The processes were allocated uniformly across the two nodes, ensuring that the measured latency consistently reflects inter-node communication. Performance metrics are analyzed and visualized using results obtained for different data sizes.

The data collection has been conducted by Bash scripts managing both the resource allocation, the mpirun call and the data aggregation. Due to time constraints multiple bash scripts were used, each one dedicated to a specific test, each test was evaluated on multiple iterations (10^4) in order to get stable results.

2.1 Algorithms

The collective operations selected for the tests were:

- **Broadcast**: a one-to-all communication operation that sends the same message from the root process to all other processes in the communicator.
- **Scatter**: a one-to-all communication operation that distributes a message from the root process to all other processes in the communicator.

We compared the default implementation of both the algortihms, which should be the most optimal one among all the possible considering aspects like the message size and the number of processes, with three of the possible implementations provided by the OSU benchmark, particularly, **Basic Linear**, **Binary Tree** and **Pipeline** for the broadcast operation and **Basic Linear**, **Binomial** and **Non-blocking Linear** for the scatter operation. These different approaches, which optimize in different ways the data transimission, are briefly described above and can be visualized in Figure 1 for the broadcast operation and in figures 2 and 3 for the scatter operation.

- **Basic Linear**: Root node sends individual messages linearly to all participating nodes.
- **Binary Tree**: Each internal process has two children with sequential data transmission from each node to both children.
- **Pipeline**: Linear structure, unidirectional propagation from root to leaves.
- **Binomial**: Employs a binomial tree structure for scatter operations.
- **Non-blocking Linear**: Overlapping receive and send operations, ensuring efficient communication.

2.2 Mapping

Before conducting all the tests, we evaluated the impact of some different mapping strategies on the performance of the collective operations. We compared the performances of the `--map-by core` and `--map-by socket` strategies, which are common strategies used in HPC applications. The `--map-by` option is a crucial runtime parameter that allows to specify how processes are mapped to the available resources, affecting performance, especially in terms of communication overhead and memory access efficiency. The test was obviously conducted in order to evaluate the impact of the mapping strategy on the latency of the collective operations and select the most efficient strategy.

- Using the `--map-by core` strategy processes are placed on individual cores, typically within the same socket initially, until all cores on that socket are occupied. Intra-socket communication benefits from lower latency because the cores share the same L1/L2/L3 caches and have direct access to the same memory controller;
- Using the `--map-by socket` strategy processes are distributed across sockets earlier in the mapping sequence. Inter-socket communication introduces additional latency due to the need for cross-socket data transfers via the interconnect between sockets.

We tested the two mapping strategies for the broadcast operation with a small message size of 4 byte, using the **Pipeline** and **Binary Tree** algorithm. The results in Table 1 shows that the **map-by-core** strategy is more efficient in terms of latency, especially for a linear structure algorithm as the **Pipeline** one, while the effect on the **Binary Tree** is less pronounced. This was attributed to the higher latency associated with distant communication paths between subsequent processors in the map-by socket scenario.

Test Type	**Message Size**	**Avg Latency (μs)**
Map by socket - Pipeline Algorithm	4	28.01
Map by core - Pipeline Algorithm	4	11.40
Map by socket - Binary Tree Algorithm	4	3.22
Map by core - Binary Tree Algorithm	4	2.76

Table 1: Latency vs map-by comparison for Broadcast operation. 128 cores per Node

Even if this effect may reduce when dealing with bigger messages we decide to fix in all our tests the same mapping and use the `--map-by core` strategy.

3 Experiments

Two different experiments have been conducted: the first aiming to test the overall performances of the different implementations of the two collective operations varying both the message size and the number of processes, the latter focused on modelling the behaviour, when varying the number of processes, of the different implementations of the operations while keeping the message size fixed.

3.1 Overall performances evaluation

Description of the experiment: Latency measurement for each of the operation algorithm implementations, varying cores from 1 to 128 per node, totaling 256 cores, and ranging message sizes from 1 to 2^{20} bytes.

The first data visualization of these tests is shown in figures 4 and 5. We see that all but the default implementations follow a similar increasing trend, both in message size and number of core directions. We can assess a similar behavior for the Pipeline and the Basic Linear implementation, in the case of the broadcast operation, and for the Non-Blocking linear and the Basic Linear implementation, in the case of the scatter operation, while the binary tree and the binomial implementation show a better management of resources, particularly when increasing the number of processes. The default algorithms show their dynamic behavior, i.e. they heuristically choose the best implementation depending on the message size and the number of processes.

To better understand the differences between the different implementations, we plotted the logarithmic relative difference in latency of the different implementations, by pairing them. The results are shown in figures 6 and 7. These plots confirm the previous observations and show that in most cases, Binary Tree and Binomial are the most efficient implementations for the broadcast and scatter operations, respectively, especially when dealing with a high number of processes and a large message size.

An overall comparison of the best performing algorithms for the broadcast and scatter operations is shown in figures 8 and 9. Here we can see that for the broadcast operation, the pipeline algorithm is almost never the most efficient, while all implementations of the scatter operation have some cases where they are the best performing.

3.2 Performance models

Description of the experiment: Latency measurement for each of the operation algorithm implementations, varying cores from 1 to 128 per node, totaling 256 cores, while keeping a fixed small message size of 4 bytes.

The main goal of this experiment was modelling a theoretical performance model for each of the implementations of the broadcast and scatter operations. Since this is a challenging task, the first assumption was to consider the data transfer instantaneous, so that the only source of latency was due to the time needed to set up the communication between the processes. For this reason the message size was set to 4 bytes. The first step was then to measure the latency point to point across the different regions of an EPYC node, and between two different nodes, as shown in Table 2.

Region	**Latency (s)**
Same CCX	0.15e-6
Same CCD, Diff. CCX	0.31e-6
Same NUMA	0.34e-6
Same SOCKET	0.36e-6
Diff. SOCKET	0.65e-6
Diff. NODE	1.82e-6

Table 2: Latency values for different core binding regions

The second step consisted in checking how the resources were allocated by using the MPI flag `--report-bindings`, and we found that they were allocated in a subsequential fashion across the first node and then distributed across the second. This was useful in order to understand the "distance" between the processes.

The **Pipeline** model considers a linear unidirectional propagation from the first to the last process, the latency was then modeled as the sum of the point-to-point communications between subsequent cores based on the binding region which is known, since we adopted a bind to core policy. The model is then:

$$T_{\text{pipeline}}(n) = \sum_{i=0}^{n-1} T_{\text{pt2pt}}(i, i+1) \tag{1}$$

Where $T_{\text{pipeline}}(n)$ is the latency for n-th process, and $T_{\text{pt2pt}}(i, i+1)$ is the latency between the i-th and the $i+1$-th process.

The **Basic Linear** model considers the root node sending indivual messages linearly from the first to the last process, the latency was then modeled as the sum of the point-to-point communications between root core and the subsequent cores. However this model gradually overestimates the latencies of the "distant" ($n > 128$) cores. This could be due to some inner optimization of the MPI implementation, i.e. the messages are sent in parallel without waiting the previous one to be received, as a Non-Linear Blocking. To consider this factor we modeled a sort of accounting method analysis, by considering a

discount factor on each point-to-point communication, increasing as the point-to-point communication is distant from the root.

The model is then:

$$T_{\text{linear}}(n) = \sum_{i=0}^{n-1} T_{\text{pt2pt}}(0, i) \cdot \text{discount}(i) \tag{2}$$

Where $T_{\text{linear}}(n)$ is the latency for n-th process, $T_{\text{pt2pt}}(0, i)$ is the latency between the root and the i-th process, and discount(i) is the discount factor for the i-th process, which is a fixed value empirically estimated.

The **Binary Tree** model incorporates the binary structure inherent in the problem. A basic model could be thought as follows: after determining the tree's height, the total communication latencies across all layers are calculated. At each layer, parallel point-to-point communication occurs among different processes, and the model accounts for this by considering the maximum point-to-point latency within that layer, determined by the core bindings. Despite its simplicity, this model consistently underestimates observed results. Latency increases noticeably only when the number of cores doubles to a power of two, highlighting a clear underestimation. Additionally, the model assumes perfect parallelism, even as point-to-point communications double at each layer, further contributing to the discrepancy.

To mitigate this issues, two penalty factors were introduced. The first addresses the latency of point-to-point communications in each layer, while the second considers the total number of processes involved. These factors, derived empirically, significantly improve the model's alignment with measured data compared to the original version. The model is then:

$$T_{\text{binary}}(n) = \text{n-proc-pen}(n) + \sum_{i=1}^{H(n)} T_{\text{pt2pt}}(i) \cdot \big(1 + \text{comm-pen}(i)\big) \tag{3}$$

Where $T_{\text{binary}}(n)$ is the latency for n-th process, $T_{\text{pt2pt}}(i)$ is the latency of the longest communication occurring on the i-th level, n-proc-pen(n) is the penalty factor for the total number of processes, comm-pen(i) is the penalty factor depending on the communications on the i-th layer, and $H(n)$ is the height of the tree for the n-th process.

For the scatter operations we are able to use the linear model and the binary tree model already described. The first for both the **Basic Linear** and the **Non-Blocking Linear** implementations, the latter for the **Binomial** implementation, concluding that these operations are similar in latency growth to the broadcast operations. The results of the models compared with the measured data are shown in figures 10 and 11. Since these simplicistic models are based on the assumption of instantaneous data transfer, they are not able to capture the real behavior of the operations, they overlook data fluctuations and heuristic optimizations of the real algorithms, but still they can be useful to understand and describe the general trend of the latency growth.

A Appendix: Images

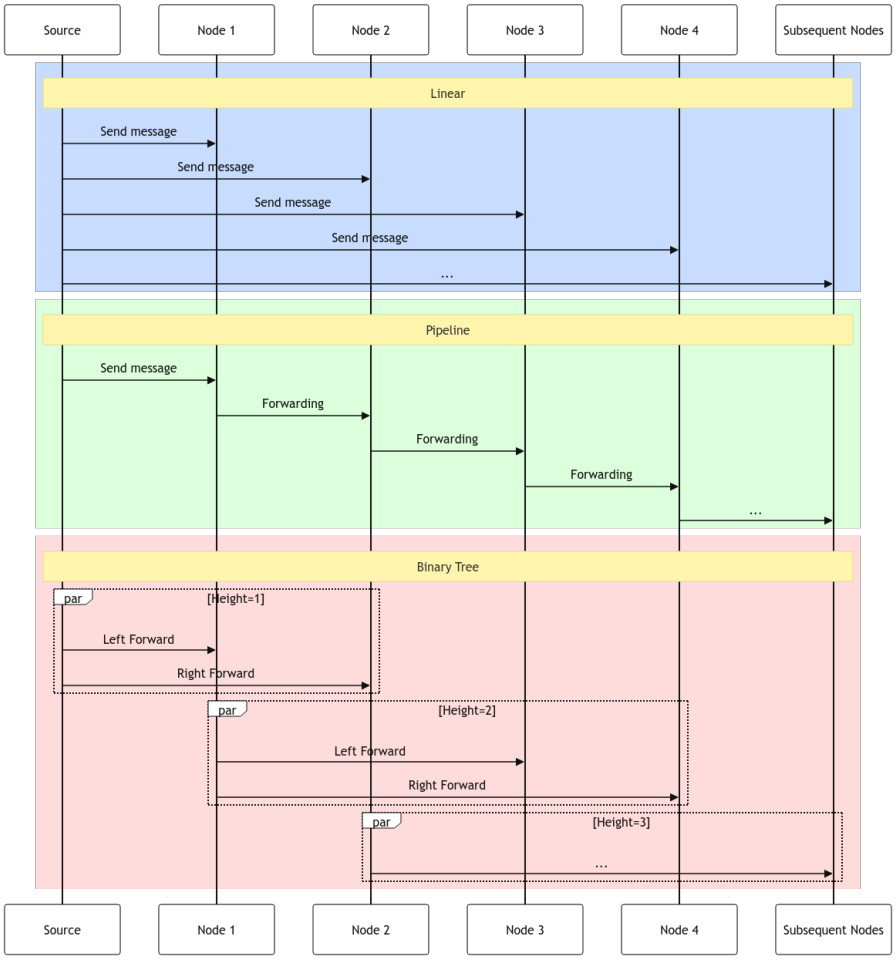

Figure 1: Algorithms behaviour for the broadcast operation.

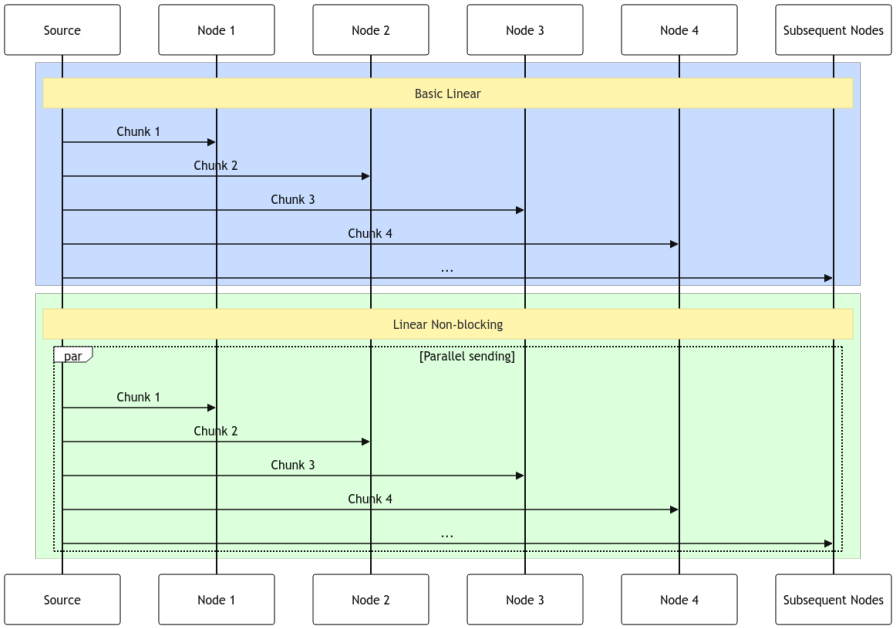

Figure 2: Linear algorithms behaviour for the scatter operation.

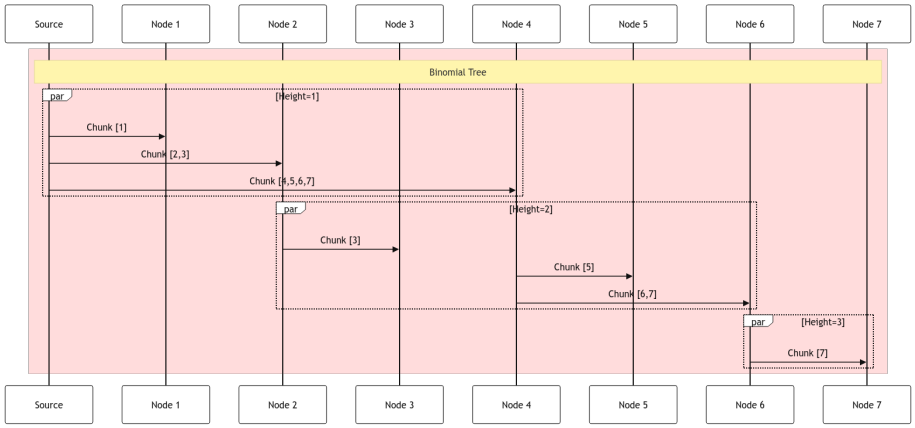

Figure 3: Binomial algorithm behaviour for the scatter operation.

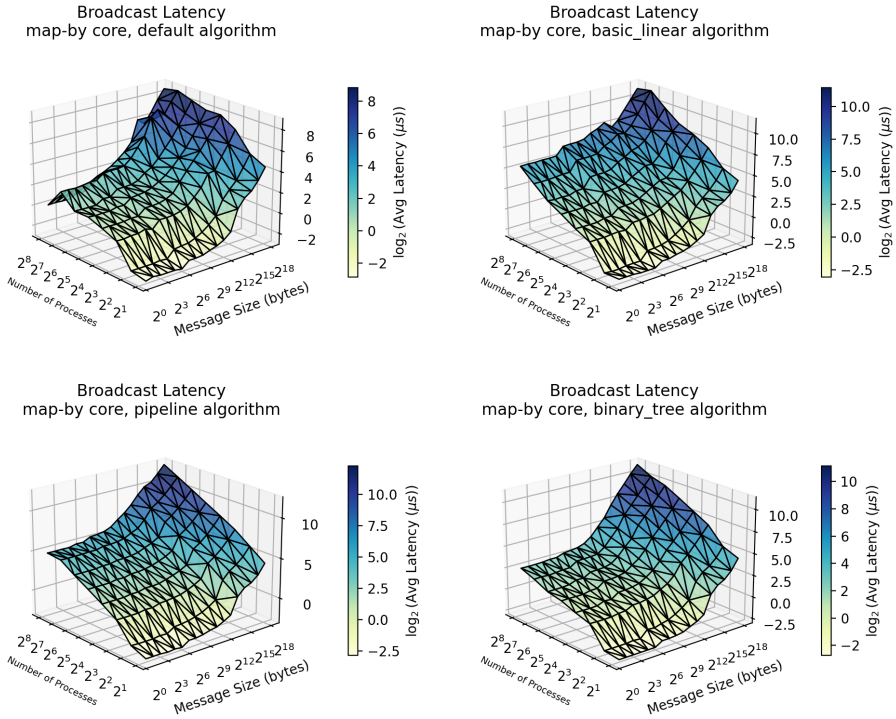

Figure 4: Broadcast operation latency 3D heatmaps for different algorithmic strategies.

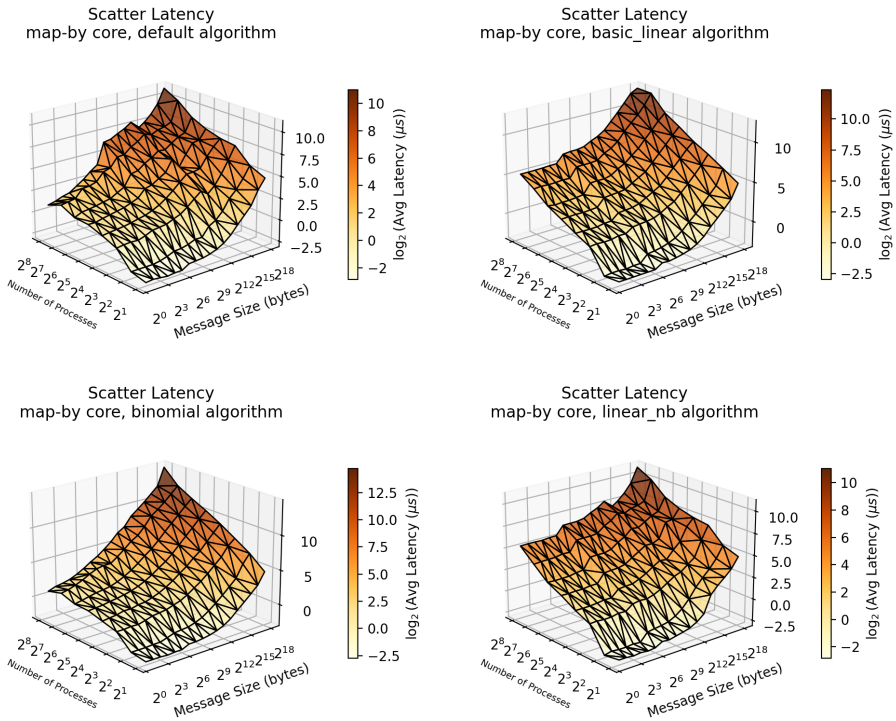

Figure 5: Scatter operation latency 3D heatmaps for different algorithmic strategies.

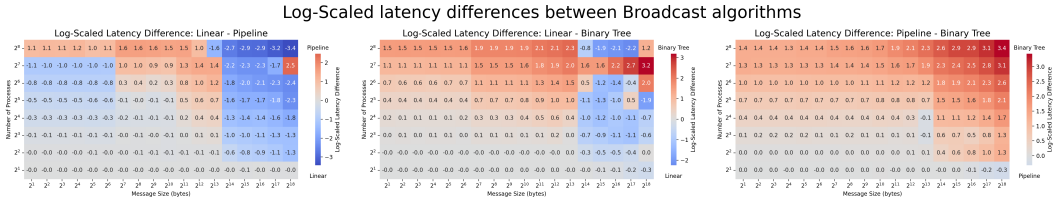

Figure 6: Broadcast relative latency logarithmic difference heatmaps for different algorithmic strategies.

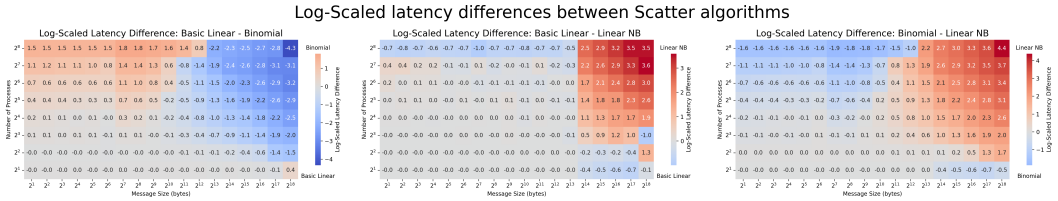

Figure 7: Scatter relative latency logarithmic difference heatmaps for different algorithmic strategies.

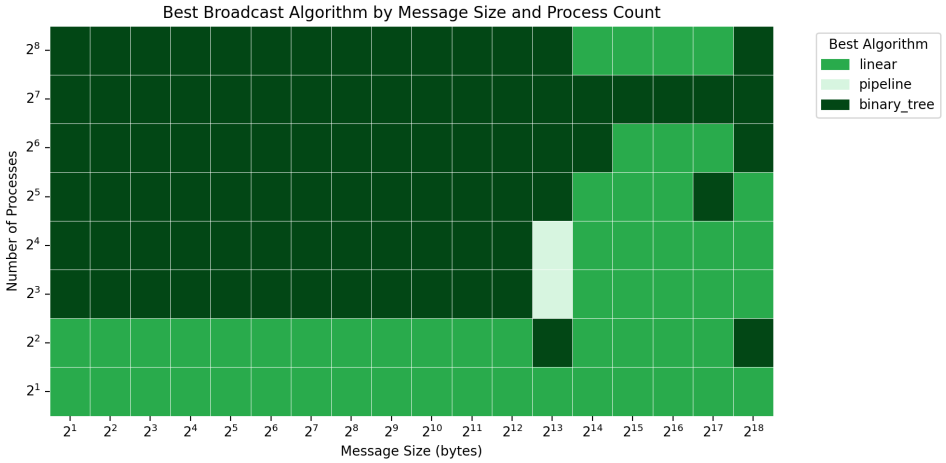

Figure 8: Best performing broadcast algorithm for different message sizes and number of processes.

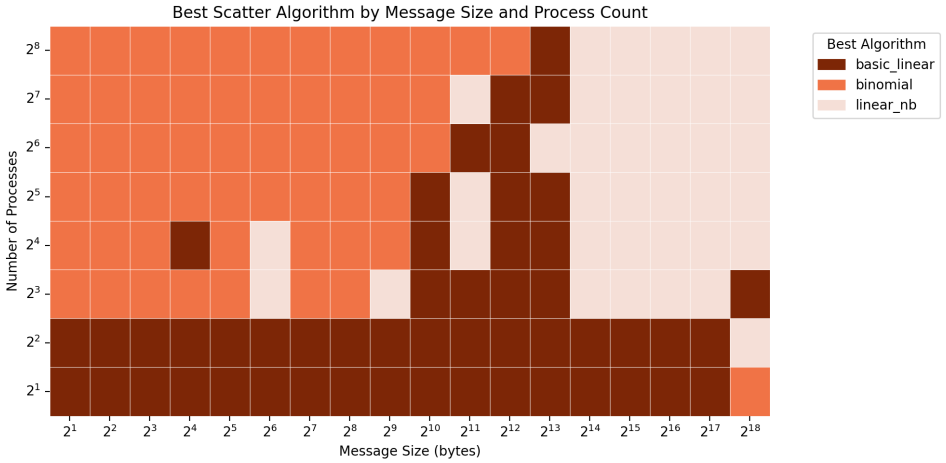

Figure 9: Best performing scatter algorithm for different message sizes and number of processes.

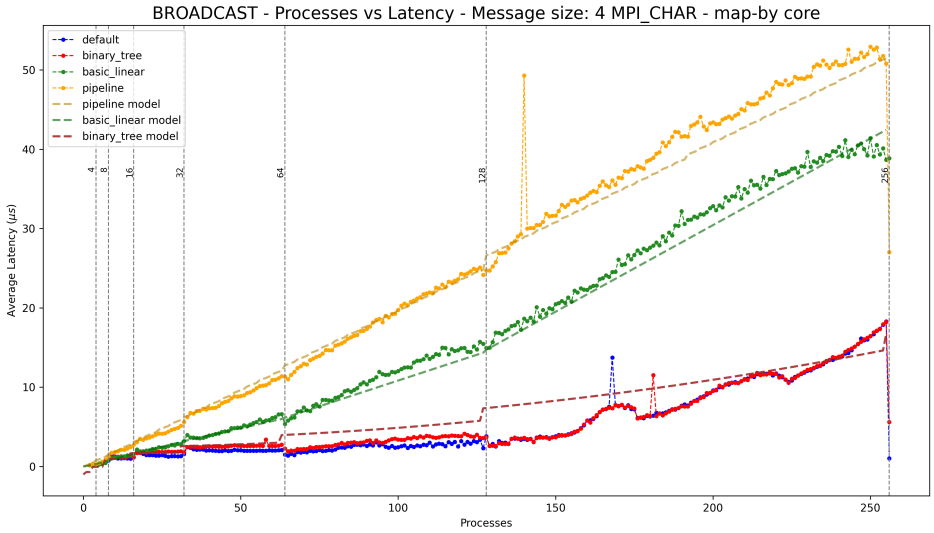

Figure 10: Broadcast operation models compared with measured data.

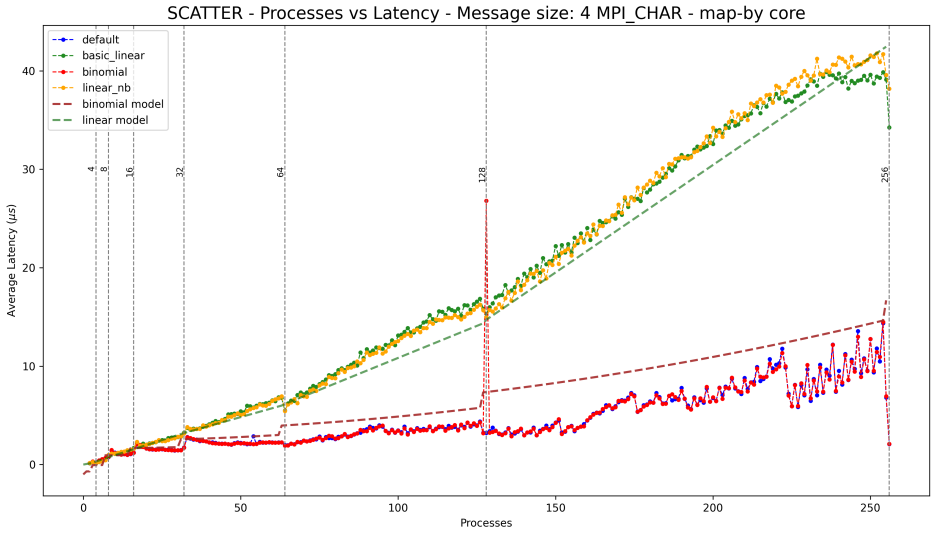

Figure 11: Scatter operation models compared with measured data.