Modeling of MPI Collective Operations: Broadcast and Reduce

Marco Tallone

March 2024

Abstract

Collective operations are the building blocks of parallel applications. For this reason, there is an interest for efficient implementations of these operations. This work focuses on the broadcast and reduce MPI implementations and presents two model for latency prediction of these operations: one based on point-to-point communications and the other on a linear regression model. The models are compared among each other and with the actual performance of the MPI implementations both in terms of explainability and overall accuracy to understand the trade-offs between the two approaches.

1 Introduction

Collective operation in *Message Passing Interface (MPI)* [2] are foundamental building blocks for parallel applications. They are used to exchange data among processes and are often used to implement higher level parallel algorithms. Efficient implementations of these operations that aim to minimize latency and maximize throughput are therefore a must for high performance computing.

This report studies the case of the broadcast and reduce collective implementations of the MPI standard by presenting two models for latency prediction. In particular, since the *OpenMPI* library offers the possibility to choose among different algorithms for these operations, this study, together with the default MPI implementation, also considers the *linear*, *chain* and *binary tree* algorithms for the selected operations.

The main problem covered in this report is the task of predicting the overall latency of these operations for various message sizes and number of processes in multi-core systems. The measurement of latency times in different conditions has therefore been carried out by running the well-known *OSU Micro-Benchmarks* suite [6] on the *ORFEO* cluster of the AREA Science Park in Trieste [5] by testing different processes allocations and message sizes.

With the collected data, two different models for latency estimation have been built. The first one is based on point-to-point communications latencies and builds on top of the model presented by *Nuriyev et al.* [4]. The second model is based on a linear regression approach and uses the collected data to train a model that can predict the latency of the collective operations.

The underlying goal of this project has been to compare the two approaches and attempt to find the best compromise between accuracy and affidability in predictions and the overall explainability of the model, taking into account the architecture on which the benchmark has been conducted.

The following sections are organized as follows. In Section 2, the studied algorithms for the collective operations will be presented and analyzed from a theoretical point of view. Section 3 describes the architecture and the precise methodology used for the measurements. Sections 4 and 5 will then present the implemented models for the latency predictions, followed by the obtained results and some final considerations in the last two sections of this report.

With the aim of maintaining a clear and concise report of the work, the following sections will only present the most noticeable results of the study, while further analysis and the detailed model implementations will be available on the author's GitHub repository [**?**].

2 Collective and Point-to-Point Communications

In order to better understant the models presented in this report, it's first necessary to briefly explain how the studied algorithms for the modelled collective operations work.
The two collective operations studied in this work are the **broadcast** operation, where a single process (the root) sends the same message to all processes, and the **reduce** operation, where all processes send their data to a single process (the root) that aggregates them.
OpenMPI architecture is based on a modular approach where different software components, plugged in the library kernel, provide specific implementation features. The `tuned` component, in particular, is responsible for the selection of different algorthms for a particular collective operation. Since an MPI collective operation can be seen as a set of point-to-point tranmission between processes, the algorithms differ one from the other in the way they manage these point-to-point communications. In the case of this study, a total of 4 different algorithms have been considered for both operations.
First of all, the **default** algorithm has been used for both operation as a baseline. In MPI, the default algorithm has no precise implementation since the library itself decides at runtime one of the many possible algorithms available for the specific collective operation requested, based on a decision routine.
The **linear** algorithm, also known as **flat tree** algorithm, has then been taken into account for this study. As depicted in figure 1a, this algorithm translates into a single level tree topology where the root has $P-1$ children and the message is sent (received)[1] to (from) children processes without segmentation. Since *OpenMPI* implements this algorithm using non-blocking send and blocking receive point-to-point communications, the tree topology of this algorithm is known in literature as a *Non-Blocking Fat Tree (NBFT)* [4]. The execution time $T^c_{NBFT}(P,m)$ of such algorithm, while transmitting a message of size m to $P-1$ processes throught a channel of communication c, can be upper bounded by the execution time $T^c_{BFT}(P,m)$ of its *Blocking Fat Tree (BFT)* counterpart, which only uses blocking send-receive communications. At the same time, $T^c_{NBFT}(P,m)$ must logically be greater or equal to the time of a single point-to-point communication, hence the following inequality holds:

$$T^c_{p2p}(m) \leq T^c_{NBFT}(P,m) \leq T^c_{BFT}(P,m) = (P-1)\cdot T^c_{p2p}(m) \tag{1}$$

where $T^c_{p2p}(m)$ is the time of a single point-to-point communication transmitting a message of size m through channel c.
The third algorithm considered has been the **chain** algorithm. In this case, each node of the topology has one child, therefore the message is split in n_s segments of size m_s and transmission of segments continues in a pipeline where process i sends (receives) segments to (from) process $i+1$. Therefore, this algorithm describes a chain tree of height $P-1$, that can be completed in $P+n_s-2$ sequential stages. Since each stage consists of multiple concurrent NBFTs of $P=2$ nodes, as shown in figure 1b, the overall execution time of the algorithm will be equal to the sum of the maximum times $T^{c_i}_{NBFT}(2,m_s)$ of each i^{th} stage.

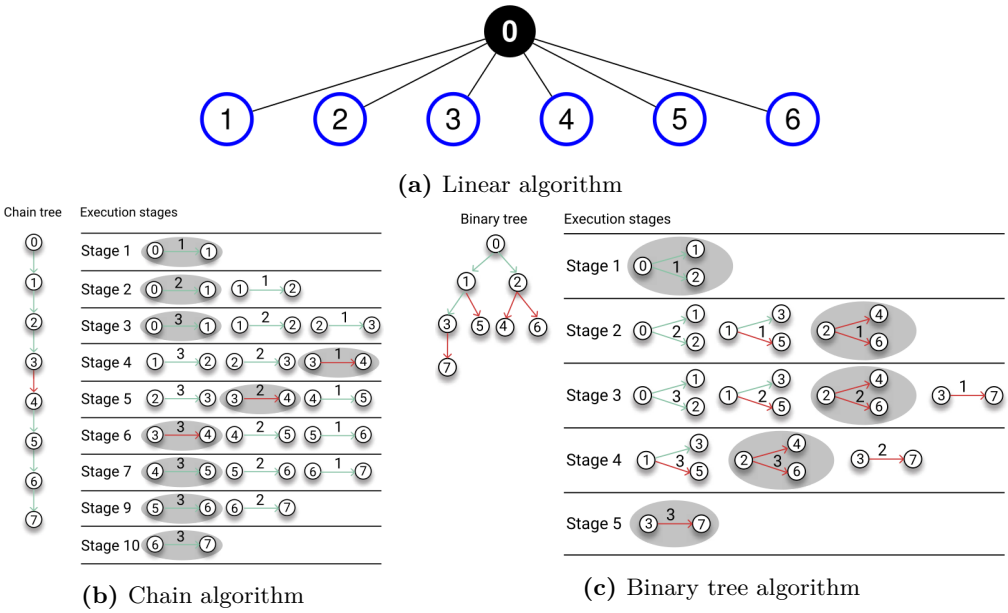
$$T_{chain}(P,m,n_s) = \sum_{i=1}^{P+n_s-2} \max_{j_i} T^{c_{j_i}}_{NBFT}(2,m_s) \tag{2}$$

Where c_{j_i} is the communication channel of the j^{th} NBFT running at the i^{th} stage.
The last algorithm studied is the **binary tree** algorithm (figure 2c). As the name suggests, this algorithm is represented by a binary tree where each node i exchanges segmented messages with two children $2i$ and $2i+1$. Assuming for simplicity that the tree is complete, then its height will be $\lfloor \log_2(P) \rfloor$ and the algorithm will be completed in $\lfloor \log_2(P) \rfloor + n_s - 1$ stages, each cosisting of multiple NBFT of either 2 or 3 nodes. Therefore, in this case:

$$T_{binary}(P,m,n_s) = \sum_{i=1}^{\lfloor \log_2(P) \rfloor + n_s - 1} \max_{j_i} T^{c_{j_i}}_{NBFT}(P_i,m_s) \tag{3}$$

Where the number of NBFTs changes at each stage, starting from 1 from stage 1 and reaching a maximum of $2^{\lfloor \log_2(P) \rfloor}$ in the middle stages.

[1] This refers to the broadcast case, between brackets the analogous for the reduce one.



(a) Linear algorithm

(b) Chain algorithm

(c) Binary tree algorithm

Figure 1: Schematic representation of (a) linear, (b) chain and (c) binary tree algorithms.

Equation 1 allows to approximate the execution time of a NBFT as follows:

$$T^c_{NBFT}(P,m) \approx \gamma^c(P,m) \cdot T^c_{p2p}(m) \tag{4}$$

where $\gamma^c(P,m)$ is a *parallelization factor*, representing the increase in latency of $P-1$ concurrent point-to-point communications in channel c, originating from the same root and transmitting a message of size m, with respect to a single point-to-point communication. Starting from this approximation, is therefore possible to build a model for the prediction of collective operations latencies based on single point-to-point communications latencies as it'll be explained in the next sections. In fact, the ability of predicting the latency of an arbitrarily large NBFT makes also possible to extend the model for predicting latencies of the previously introduced algorithms thanks to equations 2 and 3.

3 Epyc Architecture and Latency Measurements

The details of the implemented models, as well as the collection of the latencies data, are strictly connected to the architecture on which the benchmarks have been conducted. In particular, the *EPYC* partition of the *ORFEO* cluster has been used for the measurements. Nodes of this partition are equipped with two sockets with AMD EPYC 7H12 (Rome) cpus installed, for a total of 128 cores per node [1]. Inside each of these cpus, cores are organized in 4 NUMA regions. Each region contains then 2 Core Complex Die (CCD), which in turn contain 2 Core Complexes (CCX) with 4 cores each that share the L3 cache [7].
Hence, the `osu_bcast` and `osu_reduce` benchmarks have been used on 2 EPYC nodes to collect the latencies of all the algorithms presented in sections 2 for various numbers of processes in the range $[2, 256]$ and for multiple message sizes from 1 B up to 2^{20} B. Additionally, since the `map-by` flag of the *OpenMPI* library offers the interesting possibility of allocating MPI processes in this architecture according to different policies, these measurements have been conducted for the all three of the following mapping policies:

- `map-by core` : processes are allocated to cores in a round-robin fashion;
- `map-by socket` : processes are allocated to sockets in a round-robin fashion;
- `map-by node` : processes are allocated to nodes in a round-robin fashion.

Concerning the point-to-point communications, a premise is needed. As introduced in section 2, the execution time of the different algorithms is also related to the specific communication channel c used during the single point-to-point transmissions. This study tries to model the complex architecture of the EPYC node considering 4 possible channels:

- the **cache** channel (0), used by processes in the same CCX sharing the L3 cache;
- the **core** channel (1), used by processes allocated in the same socket;
- the **socket** channel (2), used by processes in the same node, but in different sockets;
- the **node** channel (3), used by processes allocated in different nodes.

The numbers in brackets hint to the fact that channels are ordered according to the increasing latency of the point-to-point communications as later explained in equation 8, under the assumption that such order remains constant for all algorithms and message sizes.
This specific channel modelling therefore guided the measurements of the point-to-point latencies in the following way. For each of the 4 channels, the `osu_latency` test has been run multiple times by allocating one process in a constant fixed position of one node while increasingly changing the core allocation of the other process until all the possible combinations had been tested. This allowed to map the point-to-point latencies of the different channels as represented in figure 2. Moreover, since these measurements have been repeated for different message sizes, the data collected for each channel have been linearly fitted as a function of the message size in order to model the single point-to-point communications of each channel using the Hockney model [3]:

$$\widehat{T}^{c}_{p2p}(m) = \alpha^{c} + \beta^{c} \cdot m \tag{5}$$

where α^c and β^c are respectively the latency and the inverse bandwidth of the channel c.
Finally, one last measurements has been conducted in order to obtain an estimate for the *parallelization factor* $\gamma^c(P, m)$ previously introduced. Since for each channel the number of processes exclusively using that specific channel ranges from 2 up to $P^{\max(c)}_{NBFT}$, the execution time $T^{c,m}_{NBFT}(P)$ of a NBFT that only uses one channel c have been measured for all possible values of $P \in [2, P^{\max(c)}_{NBFT}]$ and all channels. This has been repeated for multiple message sizes m and the measured values have then been fitted accoridng to the relation

$$\widehat{T}^{c,m}_{NBFT}(P) = \alpha^{c,m} + \beta^{c,m} \cdot (P-1) \tag{6}$$

where $\alpha^{c,m}$ is the latency of the NBFT and $\beta^{c,m}$ is the cost in the latency of a NBFT due to the addition of one process to the communication in the same channel c. According to equation 4, these fits have then been used to experimentally obtain the discrete function:

$$\widehat{\gamma}^{c}(P, m) = \frac{\widehat{T}^{c,m}_{NBFT}(P)}{\widehat{T}^{c}_{p2p}(m)} \quad \forall P \in [2, P^{\max(c)}_{NBFT}] \ , \ \forall m \tag{7}$$

that serves as a platform-specific but algorithm-independent estimation of $\gamma^c(P, m)$.

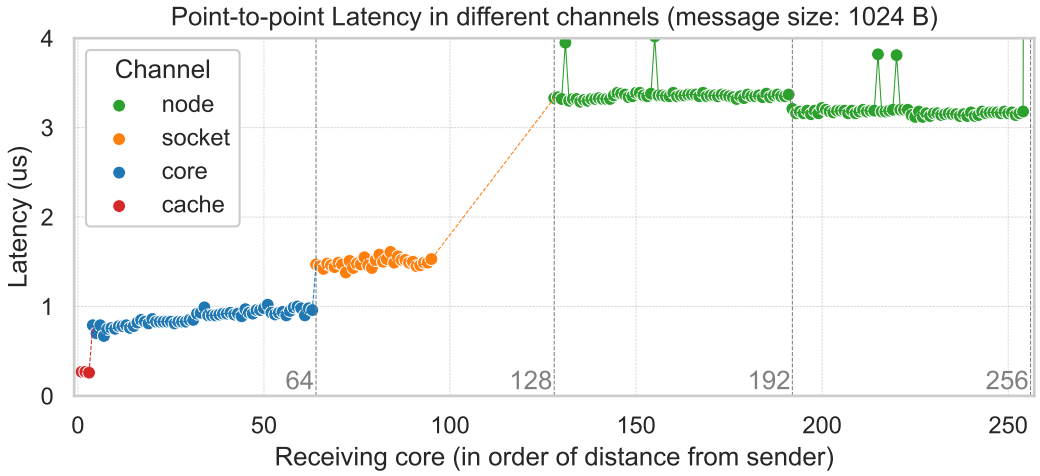


Figure 2: Point-to-point latencies for all possible receiving cores in different channels of the 2 EPYC nodes. Socket channel is missing values from 96 to 127 due to outliers.

4 Point-to-Point Communications Model

Throught the use of OOB programming in `Python` , a model has been built in order to mimic the execution of the **broadcast** and **reduce** algorithms presented in section 2. The core idea has been to build an "artificial" architecture of the EPYC nodes used in the experiment that was able to simulate the most important aspects in the latency measurements.
The model (organized in a `Python` module) is in fact a collection of classes and methods that accurately reproduce the internal architecture of the EPYC nodes used for the benchmark and some of the behaviours of the MPI library. For instance, this module offers the possibility of instantiating an object of class `Process` that can be binded to a specific `Core` , located in one of the 2 `Socket` objects, enclosed in a `Node` class. This seemingly complex structure is in truth notthing else than a simplified reproduction of the EPYC node's internal architecture. `Process` objects are then given a `send` method that aims to replicate the execution of a NBFT as explained in section 2. In fact, this method takes in input a list of receiving processesses and a message size, computes the relative positions of all the other processes with respect to the sender to understand which communication channels to use, and returns the latency of the NBFT estimated using equation 4, using the fitted point-to-point latencies and the discrete function $\widehat{\gamma}^c(P,m)$ obtained in equation 7.
In other words, this model can accurately reproduce the execution time of an arbitrarily large NBFT for any message size. According to equations 2 and 3, this can then be used to also estimate the execution time of the **chain** and **binary tree** algorithms. However, a problem that this model had to face was the situation in which more than one communication channel was used in the same NBFT. Figures 1b and 1c show this exact situation where links in red and green are used to indentify two different communication channels. As explained, this model takes into account 4 possible communication channels, ordered according to:

$$c_i \geq c_j \quad \Leftrightarrow \quad T^{c_i}_{p2p}(m) \geq T^{c_j}_{p2p}(m) \quad , \forall m,\ \forall i,j \in \{0,1,2,3\} \tag{8}$$

Therefore, this model assumes that:

$$T^{c_i}_{p2p}(m) = Q_{i,j}(m) \cdot T^{c_j}_{p2p}(m) \quad , \forall m,\ \forall i,j \in \{0,1,2,3\} |\ i > j \tag{9}$$

where $Q_{i,j}(m)$ is a platform-dependent but algorithm-independent parameter ($Q_{i,j}(m) \geq 1$) representing the *ratio of delayes* between any two communication channels c_i and c_j. This parameter is easily estimated using the fitted values for the point-to-point communication in equation 5 as follows:

$$Q_{i,j}(m) = \frac{\widehat{T}^{c_i}_{p2p}(m)}{\widehat{T}^{c_j}_{p2p}(m)} \quad , \forall m,\ \forall i,j \in \{0,1,2,3\} |\ i > j \tag{10}$$

Therefore, the execution time T_{NBFT} of a NBFT **A** using multiple channels is calculated as the time $T^{c_{\max}}_{NBFT}$ of the NBFT **B** that only uses the highest-order channel, obtained from **A** by formal replacement of of each group of $Q_{\max,j}(m)$ point-to-point communications in channel c_j with a single point-to-point communication in channel $c_{\max}$, where $c_{\max}$ is the highest-order channel used in **A** while c_j can be any lower-order channel. Formally, by labelling with N_i the number of point-to-point communication through channel c_i, this yields the following equation:

$$T_{NBFT}(P,m) = \begin{cases} T^{c_0}_{NBFT}(P,m) & \text{if } c_{\max} = c_0 \\ T^{c_1}_{NBFT}(N_1 + \lfloor \frac{N_0}{Q_{1,0}(m)} \rfloor + 1, m) & \text{if } c_{\max} = c_1 \\ T^{c_2}_{NBFT}(N_2 + \lfloor \frac{N_1}{Q_{2,1}(m)} \rfloor + \lfloor \frac{N_0}{Q_{2,0}(m)} \rfloor + 1, m) & \text{if } c_{\max} = c_2 \\ T^{c_3}_{NBFT}(N_3 + \lfloor \frac{N_2}{Q_{3,2}(m)} \rfloor + \lfloor \frac{N_1}{Q_{3,1}(m)} \rfloor + \lfloor \frac{N_0}{Q_{3,0}(m)} \rfloor + 1, m) & \text{if } c_{\max} = c_3 \end{cases} \tag{11}$$

where again, each $T^{c_i}_{NBFT}(P,m)$ is computed according to equation 4, using the fitted values for the point-to-point communications and the *parallelization factor* found in section 3.
This concretely solves the problem of computing NBFT latencies with multiple concurrent communication channels and practically enables the model to predict the execution time of the collective operation algorithms in all possible scenarios.

5 Linear Model

In addition to the point-to-point communication model presented in the previous section, a linear model has been also build to attempt the latency prediction by fitting the collected data with a simple linear regression approach. Contrarily to the previous model, this linear model is independent from the single point-to-point latency measurements and only takes into account the data collected from the benchmarks of the different algorithms.
The main assumption is that, for every possible processes `map-by` allocation and any possible message size, the latency can be described as a linear function of the number of processes involved in the collective operation. This is a reasonable assumption since the execution time of either a **broadcast** or a **reduce** operation is expected to increase with the number of processes involved. Driven by the measured data however, different fitting functions have been used depending on the specific mapping policy fixed in the test.
For data collected with the `map-by core` policy, a categorical variable[2] `z` has been added to the model in order to identify the active number of sockets during the test. This is done since the `core` allocation fills the sockets of the 2 nodes in a sequential order. Hence, each time a new socket starts being populated with processes, the overall latency is expected to increase with a much higher pace due to the added communications between the sockets. Formally, for any message size m, the linear model can be explained by the following equation:

$$T^m_{\text{latency}}(P) = \beta_0 + \beta_1 P + \sum_{i=1}^{3} (\beta_{2,i} z_i + \beta_{3,i}(P \cdot z_i)) \tag{12}$$

where z_i is the i^{th} binary dummy variable produced from the categorical variable $\mathbf{z}$ to identify wheter or not the i^{th} socket has been involved in the test[3]. Mathematically, the effect of this variable is to change the intercept and the slope of the linear fit, depending on the number of sockets involved. Therefore, this fit produces a segmented curve in the latency plot, where the slope of the curve changes at the points where a new socket starts being populated. While few can be said about the intercept, the i^{th} slope $\beta_1 + \beta_{3,i}$ of this fit can be interpreted as the increase in latency of the specific algorithm being tested, due to the addition of one process in the i^{th} socket. Hence, contrarily to the previous model, paramenters of this model are both platform and algorithm-dependent.
A similar fit has then been done for the `map-by socket` policy, but using a categorical variable `z` that identifies the active number of nodes in each test. Therefore a similar equation to 12 has been used, but having only two dummy variables. The slope in the segmented curve of this fit therefore changes only when a new node is being used in the test. The `map-by node` instead, did not require any categorical variable since in this case all the sockets of the 2 nodes are populated in a round-robin fashion. Also in these two cases the fit parameters assume the same meaning an caracteristics as the previous one.
As described by equation 12 the fits have been done with respect to the number of processes P. This is true in the case of the **linear** and **chain** algorithm, according to the expectation that the overall execution time is in some way linearly related to the number of processes. For the **binary tree** algorithm however, the execution time is expected to be logarithmically related to the number of processes, hence the $\log_2(P)$ has instead been used as the independent variable in the fit.
The data collected for the **default** algorithms have also been fitted with a linear model using the same approach and keeping P as the independent variable. However, since as explained in section 2 the algorithm for this case in not known a priori, the objective here has just been to check to which extent the linear model well represents the data.

[2] Then converted to 3 dummy variables to perform the actual fit.
[3] i ranges from 1 to 3 since indexing is zero-based and socket 0 is always involved in any test.

6 Results

In this section, the collected latency data and the results obtained with the two implemented models are presented. As explained in section 3, all the measurements have been repeated for multiple message sizes and the models have been trained accordingly. Although a clear relationship between the message size and the latency has been observed and the models are able to deal with multiple message sizes, this study focuses on the connection between latency and number of processes, hence this section will only present results relative to message size 1 B due to space constraints. Moreover, for the sake of brevity, the values for the R^2 metric, used as measurement of goodness of the models, are available in appendix A.
Figure 4 shows the results obtained for the different algorithms **broadcast** operation with all the possible mapping policies. As anticipated, the default algorithm has only been fitted with the linear model due to the impossibility of knowing the exact algorithm executed before runtime. The difference between the different mapping policies is significant in these three plots. As expected, the `map-by core` plot shows 4 distinct segments where the slope of the measured data changes corresponding to the allocation of a new socket in the underlying architecture. The same pattern repeats for the `socket` allocation but in this case only one change is visible, logically correponding to a new node being allocated. The `node` allocation, instead, shows a single slope as expected. Although not perfect, the linear model is able to predict with enough accuracy the latencies of all three allocations, with an R^2 being respectively 0.92, 0.90 and 0.89, the only exception being the initial trend of the socket allocation that shows a seemingly logarithmic trend.
Regarding the linear algorithm, the linear fit expectedly predicts with high accuracy the latencies in all of the three cases, but good results have also been obtained with the point-to-point model. The latter, in fact, is able to predict seemingly perfectly the initial trends of the core and socket allocation, while showing slight over/under-estimations for the final part of the plots and for the node allocation.
Similar results have been obtained for the chain algorithm, where the point-to-point model predicts almost perfectly the measured latencies up to the first node, but the underestimates the latencies when a second node starts being allocated in the core and socket cases. Here again, the linear fit turns out to be the superior model in terms of accuracy.
Concerning the binary tree algorithm, the logarithmic trend hypotesized in the previous section is confirmed by the measured data. However, this is only true in the case of the node allocation and for the initial part of the plot in the core and socket cases. In fact, when the second node starts being allocated, the latencies start to increment almost linearly with P. For this reason the predictions of the point-to-point model only match the measured data up to the first node, while the linear model has been modified to fit and initial logarithmic trend, plus a final linear trend in those cases, showing the versatility of this model.
Figure 3 then shows the results obtained for the **reduce** operation. In the case of the default algorithm, worse results are obtained by the linear fit in the case of the core allocation with respect to those obtained in the cases of the socket and node allocation, mostly due to the absence of an evident linear trend in the first case.
In the case of the linear algorithm, the measured data for the core and socket mappings showed a strange behavior where an initial expected linear trend up to the first node is then followed by a decrease in latency for increasing number of processes in the second node. The linear model is of course able to fit the data anyway but the phenomenon is not easily explainable, suggesting the more probable possibility of anomalies in the measurement process. The node allocation, instead, shows a clear linear trend that is perfectly fitted by the linear model. However, the point-to-point model completely fails to allign predictions with measured data in this cases, the only exception being the latecies measured for the first socket in the core allocation. Again, this can be due to anomalies in the measurement process, particularly in the measurements of the NBFT latencies for the estimation of γ^c as explained in section 3.
Finally, both the chain and binary tree algorithms present the same aspects obseerved for the broadcast operation with slight differences. The major one being the binary tree algorithm in the case of the node mapping, which does not show any logarithmic trend but rather almost constant and relatively small latency times for increasing numbers of P.

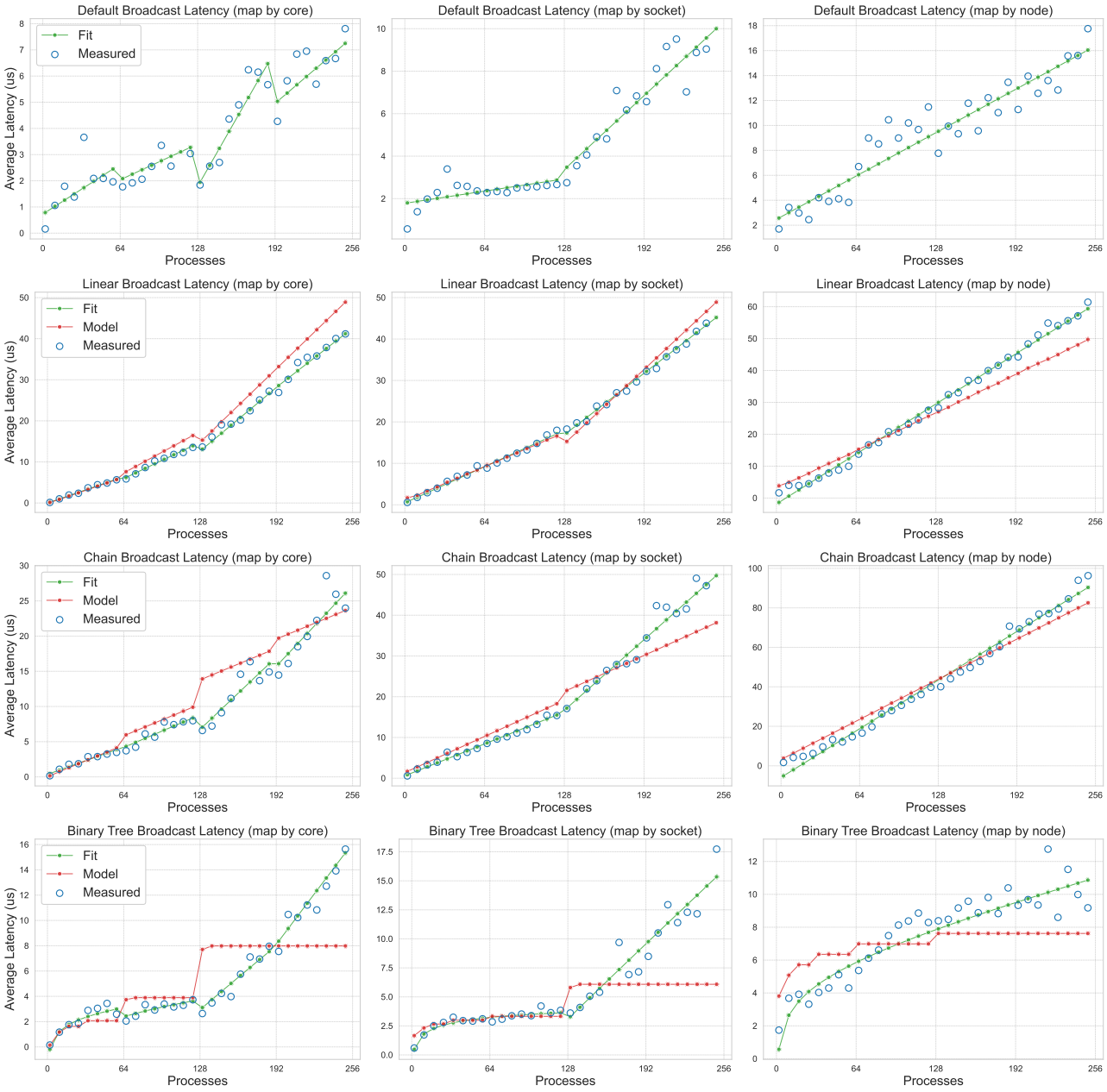


Figure 3: Comparison between point-to-point and linear fit models for different **broadcast** algorithms and mapping policies. In blue the measured data, corresponding to a message size of 1 B. The point-to-point model predictions are shown in red while the linear model predictions are shown in green.

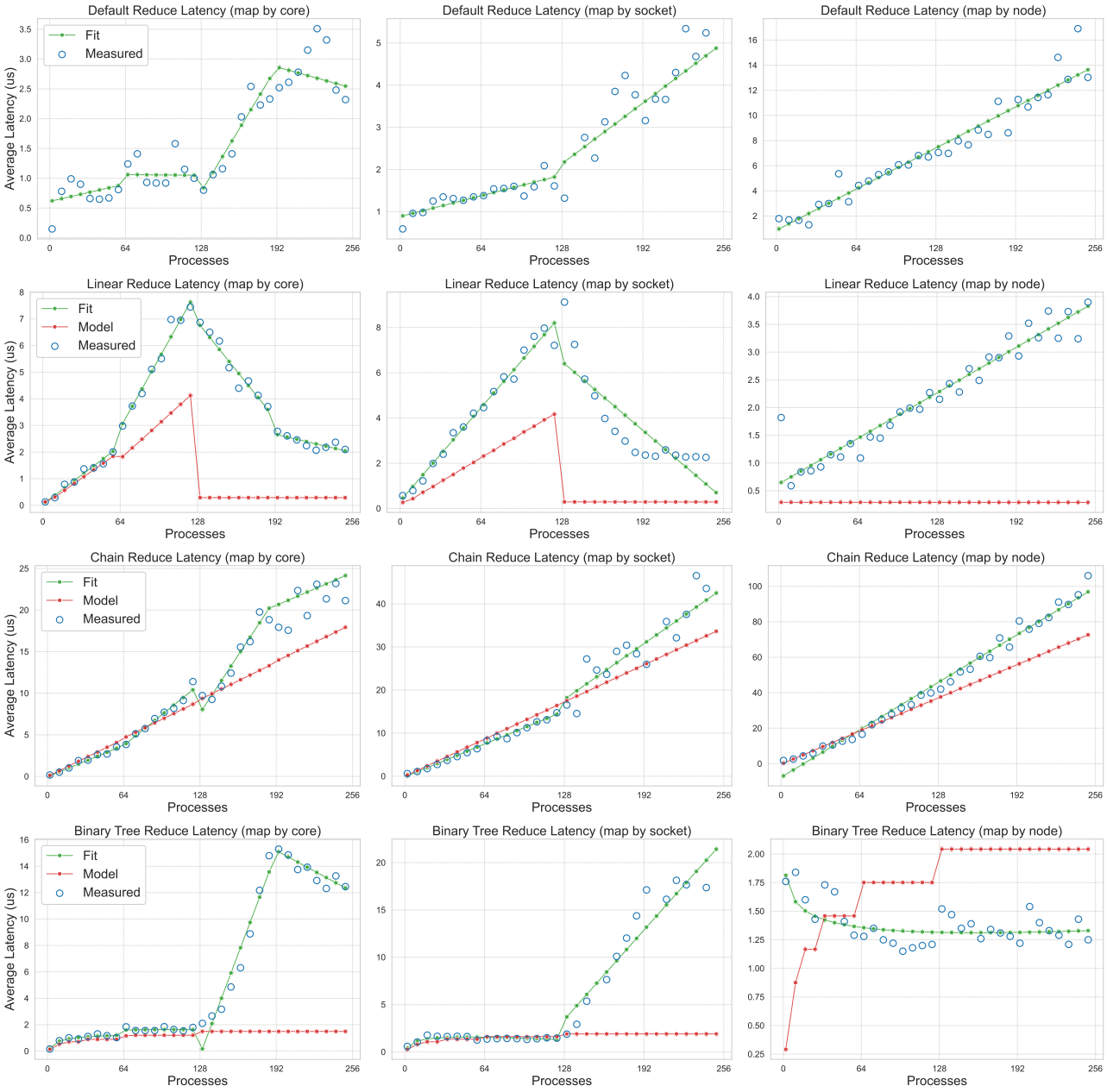


Figure 4: Comparison between point-to-point and linear fit models for different **reduce** algorithms and mapping policies. In blue the measured data, corresponding to a message size of 1 B. The point-to-point model predictions are shown in red while the linear model predictions are shown in green.

7 Conclusions

In this study, two models for latency prediction of the **broadcast** and **reduce** collective operations have been built and compared: the first one based on point-to-point communication, while the second based on a linear model. The two models have been trained and tested on the data measured on the EPYC nodes for different processes allocations as allowed by the `map-by` policy of the MPI library.
As results highligted, the linear model outperformed the model based on point-to-point communications in practically all scenarios. This is not only visually confirmed by the plots but also by the values obtained for the adjusted R^2 of the two models. This is mostly due to the high capability of the linear model to fit the data. However, despite the great results, it's important to remark that the parameters obtained for the linear model are not only platform-dependent, but algorithm-dependent as well. This means that the model has to be trained for every eventual new algorithm for which latency prediction is needed, which reduces it ability to generalize beyond the measured data. Moreover, the connection between the architecture on which the collective operations run and the parameter of the model are less clear than in the case of the point-to-point model. This is also a point of concern for the linear model, as the model might not have been able to reproduce similar predictions had the benchmarks been run on a different infrastructure.
The results obtained with the model based on point-to-point trasmissions, instead, fail in predicting the measured data in many of the observed cases. Leaving aside the anomalous results obtained for the linear algorithm of the reduce operation, this inability of the model to capture the true behaviour of the measurements could be due to multiple factors. In contrast to the linear fit, this model is based on single point-to-point communications, hence the first source of error might be the false assumption that the implicit order of the communication channels based on point-to-point latencies explained by equation 8 is always respected in all situations. Indeed, the results presented above refer to messages of size 1 B for which this condition is valid, however it has been observed that for message sizes higher than 1024 B, socket-to-socket transmissions can actually take more time than node-to-node transmissions, violating the channels order previously introduced. Moreover, this model does not takes into account resources contention among the different channels, hence the possible dependency between the point-to-point transmission and the total number of processes allocated. A second source of error might reside in the way that NBFT using multiple communication channels have been modelled. In fact, it's possible that the sobstitution of groups of lower order point-to-point communication into single higher order communications done in equation 11 is an exceeding simplification of the real process. Perhaps, a different modelling of NBFT using multiple channels could return more accurate results. A third assumption that could not alway reveal to be true might be the fact that the stages of the chain and binary tree algorithm are perfectly serial. Especially in situations where the model overestimates the true latency, possible parallelization of these stages might actually have been the cause of the discrepancy. On the other hand, when the model underestimates the true latency, it's also possible that there exist a limit to the total number of processes communicating at any given time that the model does not take into account. This might also be a possible explanation for the non-logarithmic trends observed in the final part of the binary tree algorithm plots.
Having taken these limitations into account however, the point-to-point model has showed convincing results at least in some situations. Despite a non-optimal performance, the clear advantage of this model compared to the linear fit is that its parameters are platform-dependednt but algorithm-independent, therefore the model offers the possibility of being extended for the prediction of latencies of new algorithms without the need of retraining. This includes the possibility of also simulating the latency of algorithms that have not being implemented yet, before running them on the actual infrastructure.

References

[1] AMD. Epyc 7h12 specifications, 2023. URL: https://www.amd.com/en/processors/epyc.

[2] MPI Forum. Mpi: A message-passing interface standard, 2023. URL: https://www.mpi-forum.org/docs/.

[3] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389–398, 1994. URL: https://www.sciencedirect.com/science/article/pii/S0167819106800219, doi:https://doi.org/10.1016/S0167-8191(06)80021-9.

[4] Emin Nuriyev, Juan-Antonio Rico-Gallego, and Alexey Lastovetsky. Model-based selection of optimal mpi broadcast algorithms for multi-core clusters. *Journal of Parallel and Distributed Computing*, 165:1–16, 2022. URL: https://www.sciencedirect.com/science/article/pii/S0743731522000697, doi:https://doi.org/10.1016/j.jpdc.2022.03.012.

[5] AREA Science Park. Orfeo documentation, 2023. URL: https://orfeo-doc.areasciencepark.it/.

[6] Ohio State University. Osu micro-benchmarks, 2023. URL: http://mvapich.cse.ohio-state.edu/benchmarks/.

[7] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. Memory performance of amd epyc rome and intel cascade lake sp server processors. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, page 165175, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3489525.3511689.

A Appendix

A1 Broadcast R^2 values

Algorithm	map-by core	map-by socket	map-by node
Default	—	—	—
Linear	0.929	0.986	0.916
Chain	0.839	0.895	0.959
Binary	0.534	0.321	0.421

Table 1: R^2 values for the **broadcast** predictions with the point-to-point model.

Algorithm	map-by core	map-by socket	map-by node
Default	0.924	0.901	0.899
Linear	0.995	0.996	0.995
Chain	0.971	0.984	0.987
Binary	0.976	0.956	0.891

Table 2: R^2 values for the **broadcast** predictions with the linear model.

A2 Reduce R^2 values

Algorithm	map-by core	map-by socket	map-by node
Default	—	—	—
Linear	0	0	0
Chain	0.734	0.882	0.809
Binary	0	0	0

Table 3: R^2 values for the **reduce** predictions with the point-to-point model.

Algorithm	map-by core	map-by socket	map-by node
Default	0.893	0.877	0.912
Linear	0.988	0.882	0.941
Chain	0.952	0.966	0.976
Binary	0.987	0.961	0.363

Table 4: R^2 values for the **reduce** predictions with the linear model.