Received June 23, 2021, accepted July 21, 2021, date of publication July 30, 2021, date of current version August 10, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3101689

Efficient and Accurate Selection of Optimal Collective Communication Algorithms Using Analytical Performance Modeling

EMIN NURIYEV AND ALEXEY LASTOVETSKY
School of Computer Science, University College Dublin, Dublin 4, D04 V1W8 Ireland

Corresponding author: Emin Nuriyev (emin.nuriyev@ucdconnect.ie)

This work was supported by the Science Foundation Ireland (SFI) under Grant 14/IA/2474.

ABSTRACT The performance of collective operations has been a critical issue since the advent of Message Passing Interface (MPI). Many algorithms have been proposed for each MPI collective operation but none of them proved optimal in all situations. Different algorithms demonstrate superior performance depending on the platform, the message size, the number of processes, etc. MPI implementations perform the selection of the collective algorithm empirically, executing a simple runtime decision function. While efficient, this approach does not guarantee the optimal selection. As a more accurate but equally efficient alternative, the use of analytical performance models of collective algorithms for the selection process was proposed and studied. Unfortunately, the previous attempts in this direction have not been successful. We revisit the analytical model-based approach and propose two innovations that significantly improve the selective accuracy of analytical models: (1) We derive analytical models from the code implementing the algorithms rather than from their high-level mathematical definitions. This results in more detailed and relevant models. (2) We estimate model parameters separately for each collective algorithm and include the execution of this algorithm in the corresponding communication experiment. We experimentally demonstrate the accuracy and efficiency of our approach using Open MPI broadcast and gather algorithms and two different Grid'5000 clusters and one supercomputer.

INDEX TERMS Message passing, collective communication algorithms, communication performance modeling, MPI.

I. INTRODUCTION

The message passing interface (MPI) [1] is the de-facto standard, which provides a reliable and portable environment for developing high-performance parallel applications on different platforms. Since the release of the first version of MPI, it provides a flexible communication layer including a mechanism for collective operations. MPI collective operations are classified into the following categories [1]: 1) **All-To-All** *(MPI_Allgather, MPI_Alltoall, MPI_Allreduce, MPI_Barrier)*; 2) **All-To-One** *(MPI_Gather, MPI_Reduce)*; 3) **One-To-All** *(MPI_Bcast, MPI_Scatter)*; 4) **Other** *(MPI_Scan, MPI_Exscan)*.

Rabenseifner [2] shows that collective operations consume more than eighty percent of the total execution time of a typical MPI application. Therefore, a significant amount of research has been invested into optimisation of MPI collectives. Those researches have resulted in a large number of algorithms, each of which comes up optimal for specific message sizes, platforms, numbers of processes, and so forth. Mainstream MPI libraries provide multiple collective algorithms for each collective routine. For example, MPICH [3] employs three broadcast algorithms to implement *MPI_Bcast*. In Open MPI library [4], the broadcast routine is built up with six different algorithms. However, none of the algorithms is optimal in all situations. Thus, there is a problem of selection of the optimal algorithm for each call of a collective routine, which normally depends on the platform, the number of processes, the message size and so forth.

There are two ways how this selection can be made in the MPI program. The first one, MPI_T interface [1], is provided by the MPI standard and allows the MPI programmer to select

The associate editor coordinating the review of this manuscript and approving it for publication was Pavlos I. Lazaridis.

the collective algorithm explicitly from the list of available algorithms for each collective call at run-time. It does not solve the problem of optimal selection delegating its solution to the programmer. The second one is transparent to the MPI programmer and provided by MPI implementations. It uses a simple *decision function* in each collective routine, which is used to select the algorithm at runtime (see Listing 1). The decision function is empirically derived from extensive testing on the dedicated system. For example, for each collective operation, both MPICH and Open MPI use a simple decision routine selecting the algorithm based on the message size and number of processes [5]–[7]. The main advantage of this solution is its efficiency. The algorithm selection is very fast and does not affect the performance of the program. The main disadvantage of the existing decision functions is that they do not guarantee the optimal selection in all situations.

```
int bcast_intra_dec_fixed(void *buff, int count, MPI_Datatype *datatype, int root, MPI_comm *comm)
{
  const size_t small_message_size = 2048;
  const size_t intermediate_message_size = 370728;
  const double a_p16  = 3.2118e-6;
  const double b_p16  = 8.7936;
  const double a_p64  = 2.3679e-6;
  const double b_p64  = 1.1787;
  const double a_p128 = 1.6134e-6;
  const double b_p128 = 2.1102;

  int communicator_size;
  size_t message_size, dsize;

  communicator_size = MPI_comm_size(comm);
  MPI_Type_size(datatype, &dsize);
  message_size = dsize * (unsigned long)count;

  if ((message_size < small_message_size) || (count <= 1)) {
    return binomial_tree_bcast(. . .);
  } else if (message_size < intermediate_message_size) {
    return split_binary_tree_bcast(. . .);
  } else if (communicator_size < (a_p128 * message_size + b_p128)) {
    return chain_bcast(. . . );
  } else if (communicator_size < 13) {
    return split_binary_tree_bcast(. . .);
  } else if (communicator_size < (a_p64 * message_size + b_p64)) {
    return chain_bcast(. . .);
  } else if (communicator_size < (a_p16 * message_size + b_p16)) {
    return chain_bcast(. . .);
  }
  return chain_bcast(. . .);
}
```

LISTING 1. **Open MPI decision function for MPI_Bcast.**

As an alternative approach, the use of analytical performance models of collective algorithms for the selection process has been proposed and studied. In the case of success, the analytical performance modelling approach, being as efficient as the existing decision functions approach, would guarantee the optimal selection in all situations. This approach was first proposed in [8]. In this work, several point-to-point communication models, such as Hockney [9], LogP [10], LogGP [11], PLogP [12], are used to build analytical performance models of collective algorithms. The analytical performance models are then used in decision functions for selection of the optimal algorithm. Unfortunately, the analytical performance models proposed in this work could not reach the level of accuracy sufficient for selection of the optimal algorithm.

In this paper, we revisit the model-based approach and propose a number of innovations that significantly improve the selective accuracy of analytical models to the extent that allows them to be used for accurate selection of optimal collective algorithms. Our analytical modelling approach is based on the following two innovations. First, while previous attempts to build analytical performance models of collective algorithms only take into account their high-level mathematical definition, we derive our analytical models from their high-level mathematical definition but also taking into account the properties of the algorithms, which have a significant impact on their performance and can only be extracted from the implementation code. For example, a high-level model does not detail whether the point-to-point communications used in the algorithm are blocking or non-blocking, or whether they use the *rendezvous* or *eager* protocol. Therefore, when you derive an analytical performance model, you have to assume such properties anyway, but different assumptions will lead to different performance models. Our approach is not to assume these properties arbitrarily but extract them from the implementation, making the informed decision relevant for the implementation where the model will be used. Another example is segmentation. Open MPI collective algorithms widely use the message segmentation technique, and if we do not take into account this property in the derived performance models, they will reflect not the actual algorithms but some other algorithms that are not implemented in Open MPI.

Second, we propose to estimate the model parameters separately for each collective algorithm and carefully design the communication experiments for their estimation. More specifically, we design a specific communication experiment for each collective algorithm, so that the algorithm itself would be involved in the execution of the experiment. Moreover, the execution time of this experiment must be dominated by the execution time of this collective algorithm. Then, we conduct a number of experiments on the target platform for a range of numbers of processors and message sizes and accurately measure their execution times. From these experiments, we derive a sufficiently large number of equations with the model parameters as unknowns. Finally, we use a solver to find the values of the model parameters.

We applied our approach to collective algorithms implemented in Open MPI. As a result, we managed to build a detailed analytical performance model for each collective algorithm and successfully use the models for selection of the optimal one. The accuracy of our solution has been validated on the Grid'5000 [13] platform and a Cray XC40 supercomputer [14].

The main contributions of this paper can be summarized as follows:

- We propose and implement a new analytical performance modelling approach for MPI collective algorithms, which derives the models from the code implementing the algorithms.
- We propose and implement a novel approach to estimation of the parameters of analytical performance models of MPI collective algorithms, which estimates the parameters separately for each algorithm and includes the modelled collective algorithm in the communication experiment, which is used to estimate the model parameters.
- We experimentally validate the proposed approach to selection of optimal collective algorithms on two different clusters of the Grid'5000 platform and a Cray XC40 supercomputer.

The rest of the paper is structured as follows. Section II reviews the existing approaches to performance modelling and algorithm selection problems. Section III describes our approach to construction of analytical performance models of MPI collective algorithms by deriving them from the MPI implementation. Section IV presents our method to measure analytical model parameters. Section V presents experimental validation of the proposed approach. Section VI discusses limitations of the work and how they can be mitigated. Section VII concludes the paper.

II. RELATED WORK

In order to select the optimal algorithm for a given collective operation, we have to be able to accurately compare the performance of the available algorithms. Analytical performance models are one of the efficient ways to express and compare the performance of collective algorithms. In this section, we overview the state of the art in analytical performance modelling and in measurement of model parameters. We also overview the state of the art in methods applying machine-learning models to the problem of selection of optimal MPI algorithms.

A. ANALYTICAL PERFORMANCE MODELS OF MPI COLLECTIVE ALGORITHMS

All analytical models of collective algorithms use point-to-point communication models as building blocks. The most popular point-to-point communication models used in collective models are the Hockney model [9], LogP [10], LogGP [11], and PLogP [12]. In our work, we use the Hockney model, which estimates the time $T(m)$ of sending a message of size m between two nodes as $T(m) = \alpha + \beta \cdot m$, where α and β are the message latency and the reciprocal bandwidth respectively.

Thakur *et al.* [5] propose analytical performance models of several collective algorithms for *MPI_Allgather, MPI_Bcast, MPI_Alltoall, MPI_Reduce_scatter, MPI_Reduce*, and *MPI_Allreduce* routines using the Hockney model. The parameters of the models, α and β, are assumed to be the same for all algorithms, message sizes and numbers of processes. The authors find their models not accurate enough for the task of selection of optimal collective algorithms. They conclude that in order to improve the accuracy of their analytical models, we have to assume that α and β depend on the message size and the number of processes. They do not propose models improved this way though. In our work, we stick to the assumption of independence of model parameters on the message size and the number of processes. Instead, we improve the accuracy of our models by deriving them from the implementation of the modelled algorithms. In addition, we assume that α and β may depend on the algorithm. Thus, our approach to improving the accuracy of models of collective algorithms is to make them more algorithm and implementation specific.

Chan *et al.* [15] build analytical performance models of *Minimum-spanning tree* algorithms and *Bucket* algorithms for MPI_Bcast, MPI_Reduce, MPI_Scatter, MPI_Gather, MPI_Allgather, MPI_Reduce_scatter, MPI_Allreduce collectives and later extend this work for multidimensional mesh architecture in [16]. The proposed models are built using high-level theoretical descriptions of the algorithms. Therefore, the authors conclude that while the models can be used for analysis of theoretical complexity of the algorithms, they are not accurate enough for the task of estimation and comparison of their practical performance.

An analytical performance model of a new *reduction* algorithm is proposed for a non-power-of-two number of processes by Rabenseifner *et al.* [17]. The model uses a traditional high-level mathematical description of the algorithm. The aim of the model is to understand and express the complexity of the algorithm. Like in all previous models, its level of abstraction is too high to reach the accuracy required for comparison of the practical performance of the proposed reduction algorithm with its counterparts.

A general analytical performance model for *tree-based* broadcast algorithms with message segmentation has been proposed by Patarasuk *et al.* [18]. Unlike traditional models, this model introduces a new parameter, *Maximum nodal degree* of the tree. The purpose of this model is restricted to theoretical comparison of different tree-based broadcast algorithms. Accurate prediction of the execution time of the broadcast algorithms and methods for measurement of the model parameters, including the maximal nodal degree of the tree, are out of the scope of their work.

Pjevsivac-Grbovic *et al.* [8] study selection of optimal collective algorithms using analytical performance models for *barrier*, *broadcast*, *reduce* and *alltoall* collective operations. Analytical performance models are built using the Hockney, LogP/LogGP, and PLogP point-to-point communication models. Additionally, the splitted-binary broadcast algorithm has been designed and analysed with different performance models in this work. The models are built up with the traditional approach using high-level mathematical definitions of the collective algorithms. In order to predict the cost of a collective algorithm by analytical formula, model parameters are measured using point-to-point communication experiments. After experimental validation of their modelling approach, the authors conclude that the proposed models are not accurate enough for selection of optimal algorithms.

Lastovetsky *et al.* [19] propose a point-to-point communication model for heterogeneous clusters. The model assumes that time to transmit a message between two nodes in a heterogeneous cluster is composed of the network transmission delay, source and destination processing delays. The analytical performance model of the binomial broadcast algorithm is built up using this model taking into account the impact of message passing protocols. While the predicted execution time of the binomial broadcast algorithm was close to the experimentally measured time, its use for comparison of practical performance of broadcast algorithms has never been studied.

Hofler *et al.* [20] propose a new algorithm using the hardware multicast support that performs the MPI_Bcast operation in a constant time. Switch-based InfiniBand cluster systems support hardware multicast operation. Analytical performance modelling of that type of algorithms is out of the scope of this paper.

B. MEASUREMENT OF MODEL PARAMETERS

One of the uses of analytical communication performance models is for theoretical analysis of the complexity of collective algorithms. In such purely theoretical studies, the authors do not pay much attention to methods of measurement of model parameters. However, if a model is intended for accurate prediction of the execution time of the communication algorithm on each particular platform, a well-defined experimental measurement method of the model parameters will be as important as the theoretical formulation of the model. Different measurement methods may give significantly different values of the model parameters and therefore either degrade or improve the model's prediction accuracy.

In general, a typical measurement method consists of a well-defined set of communication experiments, each of which is used to obtain an equation with model parameters as unknowns on one side of the equation and the measured execution time of the experiment on the other side. The full system of such equations is then solved to find the values of the model parameters for each particular platform. Existing measurement methods predominantly consist of *point-to-point* communication experiments, which are used to obtain a system of *linear* equations. In this subsection, we overview some notable works in this area.

Hockney [9] presents a measurement method to find the α and β parameters of the Hockney model. The set of communication experiments consists of point-to-point round-trips. The sender sends a message of size m to the receiver, which immediately returns the message to the sender upon its receipt. The time $RTT(m)$ of this experiment is measured on the sender side and estimated as $RTT(m) = 2 \cdot (\alpha + m \cdot \beta)$. These round-trip communication experiments for a wide range of message size m produce a system of linear equations with α and β as unknowns. To find α and β from this system, the linear least-squares regression is used.

Culler *et al.* [21] propose a method of measurement of parameters of the LogP model, namely, L, the upper bound on the latency, o_s, the overhead of processor involving sending a message, o_r, the overhead of processor involving receiving a message, and g, the gap between consecutive message transmission. The measurement method relies on the Active Messages (AM) protocol [22] and consists of the following four communication experiments:

- In the first experiment, the sender issues a small number of messages, N_s, consecutively without receiving any reply. The time of this experiment is measured on the sender side and estimated as $T_s = N_s \cdot o_s$. Thus, from this equation o_s can be found as $o_s = T_s/N_s$.
- In the second experiment, the sender issues a large number of messages, N_l ($N_l \gg N_s$), consecutively. Time to send a message increases due to arriving replies during sending a message. When the capacity limit of the network is reached, the send request will eventually stall. Thus, the time to send N_l messages in one direction can be estimated as $T_l = N_l \cdot g$, and g is found from this linear equation as $g = T_l/N_l$. The time of this experiment is again measured on the sender side.
- The third experiment is designed to find o_r. The sender issues N_l messages in one direction with $\triangle$ amount of time between messages. The delay $\triangle$ is introduced in order to make sure that the reply from the receiver has reached the sender side and therefore the time to process the reply by the sender can be accurately estimated as o_r. The time of this experiment is measured on the sender side and estimated as $T' = N_l \cdot (o_s + \triangle + o_r)$. Since $\triangle$ and o_s are known, o_r can be found from this linear equation as $o_r = T'/N_l - o_s - \triangle$.
- The fourth experiment performs a round-trip of a single message. The time of this experiment is measured on the sender side and estimated as $RTT = 2 \cdot (o_s + L + o_r)$. From this linear equation, L can be found as $L = RTT/2 - o_s - o_r$.

Kielmann *et al.* [12] propose a method of measurement of parameters of the PLogP (Parametrized LogP) model. PLogP defines its model parameters, except for latency L, as functions of message size. The method consists of the following four communication experiments:

- The first experiment is designed to measure $g(0)$. The sender sends N consecutive zero-byte messages followed by a single empty reply from the receiver. Network saturation is achieved by increasing the number of messages, N. It is assumed that when the network is saturated, the time T to send a large number of zero-byte messages can be estimated as $T = N \cdot g(0)$, and g can be found by solving this linear equation as $g(0) = T/N$. The time of this experiment is measured on the sender side.

- The second experiment is designed to measure $o_s(m)$. The sender starts the clock, sends a single message of size m, and then stops the clock. The time of this experiment is estimated as $T_s(m) = o_s(m)$.
- The third experiment is designed to measure $o_r(m)$. The sender sends a zero-byte message to the receiver, waits for $\triangle$ time ($\triangle > T_s(m)$), starts the clock, receives a message of size m and then stops the clock. The receiver receives the zero-byte message from the sender and sends back a message of size m. The time $T_r(m)$ measured on the sender side is estimated as $T_r(m) = o_r(m)$.
- The fourth experiment is designed to measure L and $g(m)$. It consists of two round-trips, with a zero-byte message and a message of size m respectively. The time of the first round-trip is estimated as $RTT(0) = 2(L + g(0))$, and the time of the second round-trip is estimated as $RTT(m) = 2 \cdot L + g(0) + g(m)$. Both times are measured on the sender side. L and $g(m)$ are then found from this system of two linear equations as $L = RTT(0)/2 - g(0)$ and $g(m) = RTT(m) - RTT(0) + g(0)$.

Hoefler *et al.* [23] develop a method to measure parameters of the LogGP model. LogGP extends the LogP model by adding a G parameter, the gap per byte for long messages. The building block of the method is a *ping-ping* round-trip, where the sender sends N consecutive messages of size m with delay d to the receiver, the receiver first receives all these messages and then sends them back to the sender, which also receives them all. The execution time of each communication experiment of the method, $PRTT(N, d, m)$, depends on parameters N, d and m of the experiment and measured on the sender side (PRTT stands for Parametrized Round-Trip Time). Three particular ping-ping round-trip experiments are used to obtain equations involving the LogGP model parameters as unknowns:

- The first experiment executes a round-trip of a single message ($N = 1$) of size m without delay ($d = 0$). The time of this experiment, $PRTT(1, 0, m)$, is estimated as $PRTT(1, 0, m) = 2 \cdot (o_s + L + o_r + (m - 1) \cdot G)$.
- The second experiment executes a ping-ping round-trip that issues N consecutive messages of size m without delay ($d = 0$). The time of this experiment, $PRTT(N, 0, m)$, is estimated as $PRTT(N, 0, m) = PRTT(1, 0, m) + (N - 1) \cdot G_{all}$, where G_{all} is a cumulative hardware gap, estimated as $G_{all} = G \cdot (m - 1) + g$.
- The third experiment executes a ping-ping round-trip that issues N consecutive messages of size m with delay $d > 0$. The time of this experiment, $PRTT(N, d, m)$, is estimated as $PRTT(N, d, m) = PRTT(1, 0, m) + (N - 1) \cdot \max\{o_s + d, G_{all}\}$.

Now model parameters g, G, L, o_r and o_s are found as follows:

- From equations obtained from the first and second experiments, the linear equation $G \cdot (m - 1) + g = \frac{PRTT(N,0,m) - PRTT(1,0,m)}{N-1}$, involving two unknown parameters g and G, can be derived. By repeating these experiments for a wide range of message size m, a system of m linear equations with g and G as unknowns is produced. To find g and G from this system, the linear least-squares regression can be used.
- From the first experiment with $m = 1$, the equation $PRTT(1, 0, 1) = 2 \cdot (o_s + L + o_r)$ can be derived, giving $L = PRTT(1, 0, 1)/2 - (o_s + o_r)$. However, the authors argue that due to the overlap of processor overheads and network latency, L should be more accurately estimated as $L = PRTT(1, 0, 1)/2$.
- In order to measure o_r, the measurement method proposed by Kielmann [12] is used.
- Finally, o_s is found from the linear equation $o_s + d_G = \frac{PRTT(N,d_G,m) - PRTT(1,0,m)}{N-1}$, which is derived from the first and third experiments, as $o_s = \frac{PRTT(N,d_G,m) - PRTT(1,0,m)}{N-1} - d_G$. Here, parameter $d = d_G$ of the third experiment is determined empirically to guarantee that $d_G > G_{all}$.

Rico-Gallego *et al.* [31] propose a detailed method for measurement of parameters of the τ-Lop model on a multi-core cluster. τ-Lop assumes that the cost of transmission of a message of size m is estimated as $T^c_{p2p}(m) = o^c(m) + \sum_{j=0}^{s-1} L^c_j(m, \tau_j)$, where $o^c(m)$ is the overhead of protocols and software stack, $L^c_j(m, \tau_j)$ is the time to transfer a message of size m through channel c at the j-th step of the transmission, with τ_j contending transfers ($L^c_j(0, \tau_j) = 0$), and s is the number of steps of the message transmission. For each communication channel, *shared memory* or *network*, experimental measurement of $o^c(m)$ is designed separately using the following round-trip experiments:

- The first experiment executes a round-trip of a message of size m under the *Eager* protocol for shared memory and network. The time of the experiment is estimated as $RTT^c(0) = 2 \cdot (o^c(m) + \sum_{j=0}^{s-1} L^c_j(0, 1))$. For each channel, $o^c(m)$ is found as $o^c(m) = RTT^c(0)/2$.
- The second experiment executes a round-trip of a message of size m under the *Rendezvous* protocol for shared memory and network. The time of the experiment is estimated as $Ping^c(0) = o^c(m) + \sum_{j=0}^{s-1} L^c_j(0, 1)$. Therefore, $o^c(m) = Ping^c(0)$.
- The third set of experiments exchange messages of size m between processes using *MPI_Sendrecv* routine in a ring shape. Process P_i sends a message to P_{i+1} and receives a message from P_{i-1}. Then, *MPI_Wait* is called to complete both transmissions. L^0 and L^1 are estimated by the execution of these experiments in different channels respectively.

From this overview, we can conclude that the state-of-the-art methods for measurement of parameters of communication performance models are all based on *point-to-point* *communication experiments*, which are used to derive a system of equations involving model parameters as unknowns. In this work, we propose to use *collective communication experiments* in the measurement method in order to improve the predictive accuracy of analytical models of collective algorithms.

The only exception from this rule is a method for measurement of parameters of the LMO heterogeneous communication model [32], [33]. LMO is a communication model of heterogeneous cluster, and the total number of its parameters is significantly larger than the maximum number of independent point-to-point communication experiments that can be designed to derive a system of independent linear equations with the model parameters as unknowns. To address this problem and obtain the sufficient number of independent linear equations involving model parameters, the method additionally introduces simple collective communication experiments, each using three processors and consisting of a one-to-two communication operation (scatter) followed by a two-to-one communication operation (gather). The experiments are implemented using the MPIBlib library [34]. This method however is not designed to improve the accuracy of predictive analytical models of communication algorithms.

C. SELECTION OF COLLECTIVE ALGORITHMS USING MACHINE LEARNING ALGORITHMS

Machine learning (ML) techniques have also been tried to solve the problem of selection of optimal MPI algorithms.

In [35], applicability of the quadtree encoding method to this problem is studied. The goal of this work is to select the best performing algorithm and segment size for a particular collective on a particular platform. The approach is based on the following steps. (1) Collective algorithms are executed on a particular platform to collect detailed performance data. (2) The decision map is built for the collective on a particular platform by analyzing the performance data. It is assumed that the decision map covers all message and communicator sizes. (3) The quadtree is initialized using the decision map. (4) The decision function source code is generated from the initialised quadtree. For example, Linear tree, Binary tree, Binomial tree, Split-Binary, and Chain tree broadcast algorithms are profiled with a maximum of 50 processes. The experimental results show that mean performance penalty reaches 74% and 37% and maximum performance penalty reaches 391% and 743% on different platforms respectively. While the study shows some level of applicability of the quadtree encoding algorithm to the problem, collection of detailed profiling data of collectives for all message sizes and communicator sizes is a very expensive procedure. Besides, for some message sizes and communicator sizes the penalty of the decision function is too high. Taking into account that decision trees are considered weak learners [36], the decision function will perform poorly on unseen data.

Applicability of the C4.5 algorithm to the MPI collective selection problem is explored in [37]. The C4.5 algorithm [38] is a decision tree classifier, which is employed to generate a decision function, based on a detailed profiling data of MPI collectives. The same steps are followed to build the decision tree using the C4.5 algorithm as in the quadtree encoding method presented above. The same weaknesses are shared by the decision trees built by the quadtree encoding algorithm and by the C4.5 algorithm. While the accuracy of the decision function built by the C4.5 classification algorithm is higher than that of the decision function built by quadtree encoding algorithm, still, the performance penalty is higher than 50%.

Most recently Hunold *et al.* [39] studied the applicability of six different ML algorithms for selection of optimal MPI collective algorithms. The basic idea of their approach is to create a regression model for every collective algorithm that is available for a given collective operation, predicting the execution time of the collective algorithm. The constructed regression models are then used at run time to select the algorithm that minimizes the execution time for unseen configurations. The ML algorithms employed to build the regression models are Random Forests, Neural Networks, Linear Regressions, XGBoost, K-nearest Neighbor, and generalized additive models (GAM). The configuration is characterised by the message size, the number of nodes, and the number of processes per node. The approach is evaluated using MPI_Bcast, MPI_Allreduce and MPI_Alltoall collectives. In the experimental evaluation, the number of nodes varies between 4 and 36, and the number of processes per node varies between 1 and 32. The experimental results show two things. First, it is very expansive and difficult to build a regression model even for a relatively small cluster. There is no clear guidance on how to do it to achieve better results. Second, even the best regression models do not accurately predict the fastest collective algorithm in most of the reported cases. Moreover, in many cases the selected algorithm performs worse than the default algorithm, that is, the one selected by a simple native decision function.

To the best of the authors' knowledge, the works outlined in this subsection are the only research done in MPI collective algorithm selection using ML algorithms. The results show that the selection of the optimal algorithm without any information about the semantics of the algorithm yields inaccurate results. While the ML-based methods treat a collective algorithm as a black box, we derive its performance model from the implementation code and estimate the model parameters using statistical techniques. The limitations of the application of the statistical techniques (AI/ML) to collective performance modelling and selection problem can be found in a detailed survey [40].

III. IMPLEMENTATION-AWARE ANALYTICAL MODELS OF BROADCAST AND GATHER ALGORITHMS

In this work, we propose to derive analytical models of MPI collective algorithms from their implementations rather than from high-level mathematical definitions, and use the derived models at runtime for selection of the optimal algorithms.

TABLE 1. List of collective algorithms used in open MPI 3.1.

Collective routine	Collective algorithm
Broadcast	Linear tree [24], Chain tree [18], Binary [24], Split-binary [24], K-Chain tree [24], Binomial [5], [24]
Gather	Linear [24], Linear with synchronisation [24], Binomial [24]
Allgather	Ring [5], Recursive doubling [5], Bruck [25], Neighbor exchange [26]
Barrier	Flat tree [24], Double Ring [24], Recursive doubling [24], Bruck [25]
Scater	Linear [27], Binomial [24]
Alltoall	Linear [24], Pairwise exchange [24], Bruck [25]
Reduce	Flat tree [24], Chain [24], [28], Binomial [24], Binary [24], Rabenseifner [17], [29]
Reduce-scatter	Reduce-scatterv [24], Recursive halving [24], Ring [24]
Allreduce	Recursive doubling [24], Ring [24], Ring with segmentation [24], Rabenseifner [17], [29]
Scan	Linear [24], [30], Linear with segmentation [24], Binomial [24]

We present this approach by applying it to Open MPI and its *broadcast* and *gather* collective algorithms. While the algorithms are Open MPI specific, the proposed modelling approach itself is general and can be applied to other MPI implementations and collective algorithms. The complete list of Open MPI 3.1 collective algorithms can be found in TABLE 1. In this work, we cover in detail the *broadcast* and *gather* algorithms.

As stated in Section I, we propose a new approach to analytical performance modelling of collective algorithms. While the traditional approach only takes into account high-level mathematical definitions of the algorithms, we derive our models from their implementation. This way, our models take into account important details of their execution having a significant impact on their performance. In this section, we present our analytical modelling approach by applying it to broadcast and gather collective algorithms implemented in Open MPI. This approach could be similarly applied to other collective algorithms as well as other MPI implementations such as MPICH. Analytical models of the broadcast and gather collective algorithms implemented in Open MPI are derived in Sections III-A and III-B.

To model point-to-point communications, we use the Hockney model, which estimates the time $T_{p2p}(m)$ of sending a message of size m between two processes as $T_{p2p}(m) = \alpha + \beta \cdot m$, where α and β are the latency and the reciprocal bandwidth respectively. For segmented collective algorithms, we assume that $m = n_s \cdot m_s$, where n_s and m_s are the number of segments and the segment size respectively. We assume that each algorithm involves P processes ranked from 0 to $P - 1$.

We deliberately pick Hockney, the simplest possible model, despite it does not separate the contributions of CPUs and network in the communication cost and therefore is less accurate than LogP and its derivatives. Our intention in this work is to investigate to which extent taking into account the properties of collective algorithms, which can only be extracted from the implementation, and learning the general communication parameters of the models (such as Hockney's α and β, or L, o, g, and G for LogP/LogGP) separately for each algorithm, will be able to improve the models of the collective algorithms for the purpose of comparison of their relative performance. This approach to improvement of the models has never been investigated. At the same time, the approach using different basic communication models, such as Hockney, LogP, PLogP, to improve the selective accuracy of the models of collective algorithms has been thoroughly studied by Pjesivac-Grbovic *et al.* [24]. Their work has shown that only the use of more accurate general communication models does not help improve the selective accuracy of analytical models of collective algorithms. Therefore, in our work we contrast their approach with ours and use the simplest possible general model but more relevant implementation-aware definitions of the algorithms.

We fully realise the restrictions of Hockney, especially, when it comes to modelling the effects of network congestion. In this work, however, we only consider one-process-per-CPU configurations of MPI programs. With such configurations, we observed that the effects of network congestion were not very significant on our experimental platforms, and our models of collective algorithms worked very well. At the same time, we found the network congestion to be much more impactful when we use one-process-per-core configurations, resulting in degradation of the selective accuracy of our Hockney-based models. Therefore, in order to improve the predictive accuracy of our models for such configurations, more accurate point-to-point communication

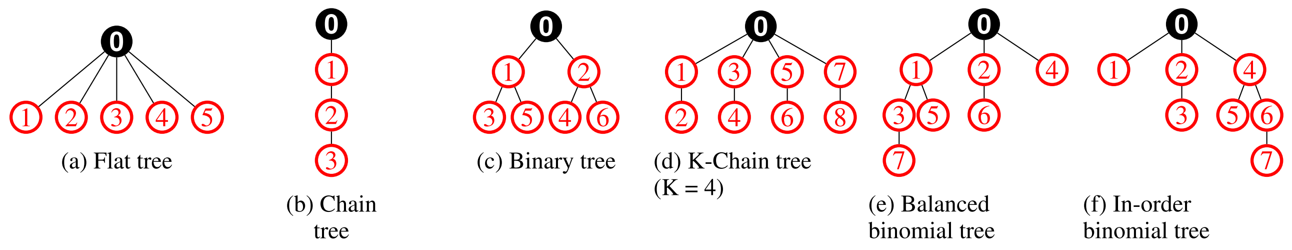


FIGURE 1. **Virtual topologies for collective algorithms.**

models, including models from the LogP family, should be considered.

A. BROADCAST ALGORITHMS

During the broadcast (*MPI_Bcast*), one process, called *root*, sends the same data to all processes in the communicator. At the end of the operation, the root buffer is copied to all other processes. In this section, we build analytical performance models of broadcast algorithms implemented in Open MPI. All broadcast algorithms implemented in Open MPI, except for the linear tree broadcast algorithm, are implemented using message segmentation. The main purpose of message segmentation is to enable higher bandwidth utilization. If a segment size is smaller than *eager limit* then it avoids the *rendezvous* protocol. Therefore, we only build analytical models of broadcast algorithms with message segmentation assuming the buffered mode of *send* operations. Models of segmented broadcast algorithms employing the *rendezvous* (synchronous) mode would have no practical application in Open MPI as they assume a configuration with a segment size being not small enough to avoid the *rendezvous* protocol, which does not make much sense.

1) LINEAR (FLAT) TREE ALGORITHM

The algorithm employs a single level tree topology shown in FIGURE 1a where the root node has $P - 1$ children. In Open MPI, the linear broadcast algorithm is implemented using blocking *send* and *receive* operations. The algorithm transmits the whole message from root to the leaves without message segmentation. Regardless of communication mode (buffered or not), because of blocking communication, each next *send* only starts after the previous one has been completed. Therefore, the execution time of the linear tree broadcast algorithm will be equal to the sum of execution times of $P - 1$ send operations:

$$T^{blocking}_{linear\_bcast}(P, m) = (P - 1) \cdot (\alpha + m \cdot \beta). \quad (1)$$

In Open MPI, this linear tree algorithm is one of the six algorithms available for implementation of the *MPI_Bcast* routine. There is another linear tree broadcast algorithm, which cannot be chosen to implement *MPI_Bcast*, but only used as a building block in other tree-based broadcast algorithms implementing *MPI_Bcast*, namely, in the *binomial tree*, *binary tree*, *k-chain tree*, and *chain tree* broadcast algorithms (see Algorithm 1 for more details). That linear

Algorithm 1 Tree-Based Segmented Broadcast Algorithm

```
if ($rank == root$) then
    for $i \in 0..n_s - 1$ do
        for $child \in list\ of\ children$ do
            MPI_Isend(segment[$i$], child, ...)
        end for
        MPI_Waitall(...)
    end for
else if ($intermediate\ nodes$) then
    for $i \in 0..n_s - 1$ do
        MPI_Irecv($segment[i]$)
        MPI_Wait(...)
        for $child\ in\ list\ of\ children$ do
            MPI_Isend($segment[i]$, $child$, ...)
        end for
        MPI_Waitall($children$)
    end for
else if ($leaf\ nodes$) then
    for $i \in 0..n_s - 1$ do
        MPI_Irecv($segment[i]$, ...)
        MPI_Wait(...)
    end for
end if
```

tree algorithm is implemented using *non-blocking* send and receive operations.

In this latter case, $P - 1$ non-blocking *send*s will run on the *root* concurrently. Therefore, the execution time of the linear broadcast algorithm using non-blocking point-to-point communications and buffered mode, $T^{nonblock}_{linear\_bcast}(P, m)$, can be bounded as follows:

$$T_{p2p}(m) \leq T^{nonblock}_{linear\_bcast}(P, m) \leq (P - 1) \cdot T_{p2p}(m). \quad (2)$$

We will approximate $T^{nonblock}_{linear\_bcast}(P, m)$ as

$$T^{nonblock}_{linear\_bcast}(P, m) = \gamma(P, m) \cdot (\alpha + m \cdot \beta), \quad (3)$$

where

$$\gamma(P, m) = \frac{T^{nonblock}_{linear\_bcast}(P, m)}{T_{p2p}(m)}. \quad (4)$$

We will use this approximation when deriving analytical performance models of the remaining five broadcast algorithms implemented in Open MPI. As we can see from Algorithm 1, the non-blocking version of linear tree broadcast

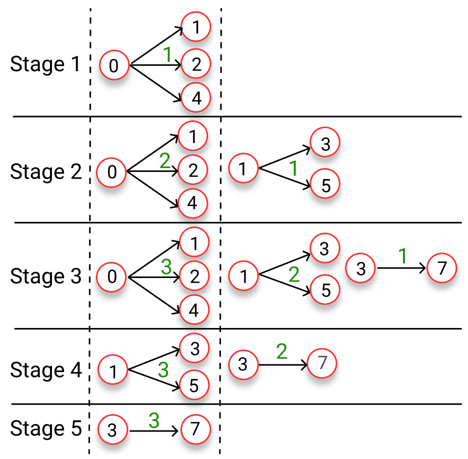


FIGURE 2. **Execution stages of the binomial tree broadcast algorithm, employing the non-blocking linear broadcast ($P = 8$, $n_s = 3$). Nodes are labelled by the process ranks. Each arrow represents transmission of a segment. The number over the arrow gives the index of the broadcast segment.**

is used in these five algorithms for transmission of a single message segment. In this paper, we assume the same fixed segment size in all segmented algorithms. Therefore, in the rest of the paper we define γ as a function of P only, $\gamma(P)$. From Formula 2, we can derive that $T^{nonblock}_{linear\_bcast}(2, m) = T_{p2p}(m)$ and, hence, $\gamma(2) = 1$.

2) BINOMIAL TREE ALGORITHM

In Open MPI, the algorithm employs *balanced* binomial tree (FIGURE 1e). The binomial tree broadcast algorithm is segmentation-based and implemented as a combination of linear tree broadcast algorithms using non-blocking *send* and *receive* operations.

FIGURE 2 shows the stages of execution of the binomial tree broadcast algorithm. Each stage consists of parallel execution of a number of linear broadcast algorithms using non-blocking communication. The linear broadcast algorithms running in parallel have a different number of children. Therefore, the execution time of each stage will be equal to the execution time of the linear broadcast algorithm with the maximum number of children. The execution time of the whole binomial broadcast algorithm will be equal to the sum of the execution times of these stages.

In Open MPI, the binomial tree broadcast algorithm employs the balanced binomial tree virtual topology. Therefore, the number of stages in the binomial broadcast algorithm can be calculated as

$$N_{steps} = \lfloor log_2 P \rfloor + n_s - 1. \quad (5)$$

Thus, the time to complete the binomial tree broadcast algorithm can be estimated as follows:

$$T_{binomial\_bcast}(P, m, n_s) = \sum_{i=1}^{\lfloor log_2 P \rfloor + n_s - 1} \max_{1 \le j \le \min(\lfloor log_2 P \rfloor, n_s)} T^{nonblock}_{linear\_bcast}(P_{ij}, \frac{m}{n_s}), \quad (6)$$

where P_{ij} denotes the number of nodes in the j-th linear tree of the i-th stage.

Using the property of the binomial tree and Formula 3, we have

$$T_{binomial\_bcast}(P, m, n_s) = (n_s \cdot \gamma(\lceil \log_2 P \rceil + 1) \sum_{i=1}^{\lfloor \log_2 P \rfloor - 1} \gamma(\lceil \log_2 P \rceil - i + 1) - 1) \cdot (\alpha + \frac{m}{n_s} \cdot \beta). \quad (7)$$

3) CHAIN TREE ALGORITHM

Each internal node in the chain tree topology has one child (Fig 1b). In Open MPI, the chain tree algorithm is segmentation-based and implemented using non-blocking point-to-point communication. While the height of the chain tree equal to $P - 1$, the algorithm will be completed in $P + n_s - 2$ steps, each consisting of a varying number of concurrent non-blocking point-to-point communications (technically, Open MPI employs concurrent non-blocking linear tree broadcast algorithms, but in this case each linear broadcast will be equivalent to a point-to-point communication). Therefore, the execution time of the chain tree algorithm can be estimated as

$$T_{chain\_bcast}(P, m, n_s) = (P + n_s - 2) \cdot (\alpha + \frac{m}{n_s} \cdot \beta). \quad (8)$$

4) SPLIT-BINARY TREE ALGORITHM

In Open MPI, the split-binary tree algorithm is segmentation-based and implemented using blocking point-to-point communication. The algorithm consists of two phases – *forwarding* and *exchange*. In the first phase, the message of size m is split into two equal parts in the root, which are then sent to the left and right subtrees respectively using message segmentation. After completion of the first phase, each node in the left subtree contains the first half of the message and each node in the right subtree – the second half of the message. Because of segmentation, each node will receive $\frac{n_s}{2}$ segments during the first phase.

As the balanced binary tree virtual topology is employed in the split-binary tree algorithm, each node in the left subtree will have a matching pair in the right subtree and vice versa. In the second phase, each pair of matching nodes in the left and right subtrees exchange their halves of the message. The execution time of the split-binary tree broadcast will be equal to the sum of the execution times of the first and the second phases. As the height of the balanced binary tree is equal to $\lfloor \log_2 P \rfloor$, we have

$$T_{split\_binary\_bcast}(P, m, n_s) = 2 \cdot (\lfloor \log_2 P \rfloor + \frac{n_s}{2} - 1) \cdot (\alpha + \frac{m}{n_s} \cdot \beta) + (\alpha + \frac{m}{2} \cdot \beta) \quad (9)$$

5) BINARY TREE ALGORITHM

In the binary tree virtual topology each internal process has two children (FIGURE 1c). In Open MPI, the binary tree broadcast algorithm is segmentation-based and uses the balanced binary tree topology (FIGURE 1c). The root broadcasts each segment to its children using the non-blocking linear tree broadcast algorithm. Upon receipt of the next segment, each internal node acts similarly. As the binary tree used in this algorithm is balanced, all the non-blocking linear broadcasts will have the same execution time, namely,

$$T^{nonblock}_{linear\_bcast}(3, m_s) = \gamma(3) \cdot (\alpha + \frac{m}{m_s} \cdot \beta).$$

As the height of the balanced binary tree is equal to $\lfloor \log_2 P \rfloor$, the algorithm will be completed in $(\lfloor \log_2 P \rfloor + n_s - 1)$ steps, each consisting of a varying number of concurrent non-blocking linear broadcasts, involving 3 processes. Therefore,

$$T_{binary\_bcast}(P, m, n_s) = \gamma(3) \cdot (\lfloor \log_2 P \rfloor + n_s - 1) \cdot (\alpha + \frac{m}{n_s} \cdot \beta). \quad (10)$$

6) K-CHAIN TREE ALGORITHM

In Open MPI, the K-chain tree algorithm is implemented using non-blocking communication and message segmentation. In the K-chain tree, the root node has $K (K = 4)$ children, while the internal nodes have a single child each (FIGURE 1d). As the height of the tree is $\lfloor \frac{P-1}{K} \rfloor$, the algorithm takes $\lfloor \frac{P-1}{K} \rfloor + n_s - 1$ steps to complete. At each step, a varying number of non-blocking linear tree broadcast algorithms will be executed concurrently (one at the first step, K at the last step, and up to $K \times (\lfloor \frac{P-1}{K} \rfloor - 1) + 1$ algorithms for intermediate steps). Note, that while Open MPI employs concurrent non-blocking linear tree broadcast algorithms, in this case the most of the linear broadcasts will be equivalent to non-blocking point-to-point communications.

The execution time of the K-chain tree algorithm will be equal to the sum of the execution times of its steps. The execution time of each step will be equal to the maximum execution time of the concurrently executed linear broadcasts. For the first n_s steps, this maximum time will be the time of the linear broadcast involving the root of the whole K-chain tree, which is estimated as $\gamma(K + 1) \cdot (\alpha + \frac{m}{n_s} \cdot \beta)$ according to Formula 3. For each of the remaining $\lfloor \frac{P-1}{K} \rfloor - 1$ steps, all concurrently executed linear broadcasts will be equivalent to non-blocking point-to-point communications, the time of which is $\alpha + \frac{m}{n_s} \cdot \beta$. Thus, the total execution time of the K-chain tree algorithm will be estimated as

$$T_{k\_chain\_bcast}(P, m, n_s) = (\lfloor \frac{P-1}{K} \rfloor + \gamma(K + 1) \cdot n_s - 1) \cdot (\alpha + \frac{m}{n_s} \cdot \beta). \quad (11)$$

B. GATHER ALGORITHMS

MPI_Gather is a *many-to-one* MPI operation. MPI_Gather takes data elements from all processes of the communicator and gathers them in one single process which is called *root*. In this section, we derive analytical formulas of the gather algorithms implemented in Open MPI.

1) LINEAR WITHOUT SYNCHRONISATION

In the Open MPI implementation of the *linear without synchronisation* gather algorithm, the root receives messages from its $P - 1$ children using blocking receive operations. Therefore, the execution time of this gather algorithm can be estimated as the sum of the execution times of $P - 1$ blocking receive operations, that is,

$$T_{linear\_gather}(P, m) = (P - 1) \cdot (\alpha + m \cdot \beta). \quad (12)$$

2) LINEAR WITH SYNCHRONISATION

The Open MPI implementation of the *linear with synchronisation* gather algorithm employs both blocking and non-blocking communications. The messages gathered from the children are all identically split into two equal parts. In order to receive all these parts from its $P - 1$ children, the root executes a loop, at i-th iteration of which it receives both halves of the message from the i-th child by performing the following steps: 1) it first posts a non-blocking receive for the first part; 2) then it sends a zero-byte message using a blocking send, signalling the child to start sending the message parts; 3) then the root posts a non-blocking receive for the second half of the message; 4) finally, it blocks itself waiting for the completion of the previously posted non-blocking receives.

At the same time, upon receipt of a zero-byte signal message from the root, each child will perform two successive standard blocking sends for the first and the second parts of its message. When the size of these parts, $\frac{m}{2}$, is greater than the eager limit, m_{eager}, than the standard blocking sends will follow the *rendezvous* protocol, that is, will be equivalent to *synchronous* sends. Otherwise, they will follow the *eager* protocol, that is, will be equivalent to *buffered* sends. In the first case, the execution of all point-to-point communications will be serialized, and, therefore, the execution time of the linear gather with synchronisation algorithm can be estimated as the sum of the execution times of the employed point-to-point communications:

$$\begin{aligned} T_{linear\_gather\_with\_synch}(P, m) &= (P - 1) \cdot (2 \cdot (\alpha + \frac{m}{2} \cdot \beta)) \\ &= (P - 1) \cdot (2 \cdot \alpha + m \cdot \beta). \end{aligned} \quad (13)$$

Otherwise, when $\frac{m}{2} \leq m_{eager}$, each child will send its half-messages concurrently. Therefore, the execution time of the linear gather with synchronisation algorithm in this case can be estimated as

$$T_{linear\_gather\_with\_synch}(P, m) = (P - 1) \cdot (\alpha + \frac{m}{2} \cdot \beta). \quad (14)$$

3) BINOMIAL ALGORITHM

In Open MPI, the binomial gather algorithm employs the *in-order* binomial tree virtual topology (FIGURE 1f). The leaf nodes and internal nodes use the standard blocking send to send the messages to their parents, which receive the message using the blocking receive. The algorithm will

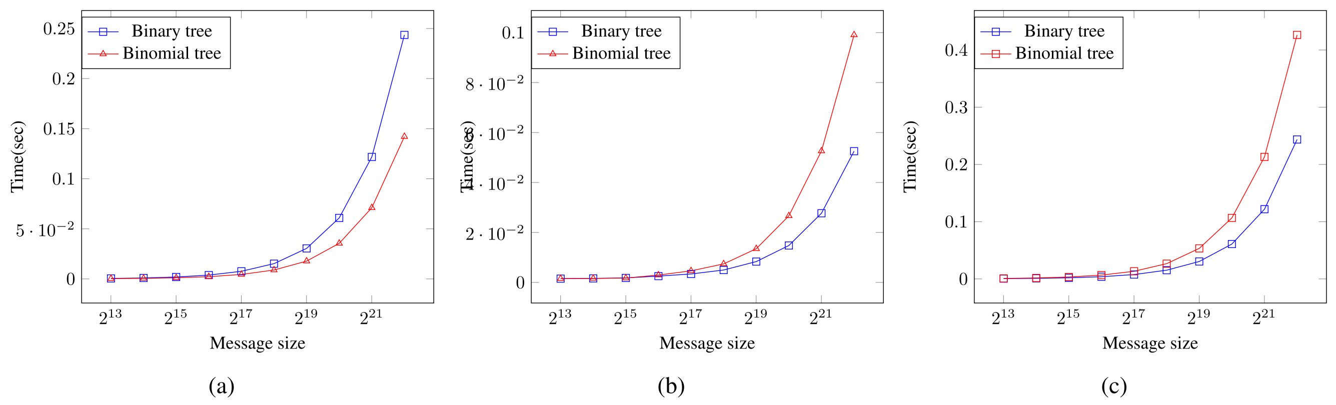


FIGURE 3. Performance estimation of the binary and binomial tree broadcast algorithms by the traditional and proposed analytical models in comparison with experimental curves. The experiments involve ninety processes (P = 90). (a) Estimation by the existing analytical models. (b) Experimental performance curves. (c) Estimation by the proposed analytical models derived from the implementation codes.

be completed in $\lfloor \log_2 P \rfloor$ steps, each performing a set of concurrent blocking receives.

At i-th step, the root will receive a message of size $2^{i-1} \cdot m$ from its i-th child, combining the messages gathered by the latter acting as the root of the i-th subtree during the previous $i-1$ steps ($i = 1, \dots, \lfloor \log_2 P \rfloor$). Given this message size, $2^{i-1} \cdot m$, will be the largest communicated at the i-th step of the algorithm, its execution time can be estimated as

$$\begin{aligned} T_{binomial\_gahter}(P, m) &= \sum_{i=1}^{\lceil \log_2 P \rceil} (\alpha + 2^{i-1} \cdot m \cdot \beta) \\ &= \lceil \log_2 P \rceil \cdot \alpha + (P-1) \cdot m \cdot \beta. \quad (15) \end{aligned}$$

C. COMPARISON OF THE PROPOSED ANALYTICAL MODELS AGAINST THE STATE OF THE ART

In this section, we use the binomial and binary tree algorithms as an example to illustrate that unlike the traditional approaches, the approach based on the derivation of analytical models of collective algorithms from their implementation codes, yields models, which can be used for accurate pairwise comparison of the performance of collective algorithms implementing the same collective operation.

Existing analytical modelling approaches [5], [24], [41] estimate the execution time of the binary and binomial tree broadcast algorithms as follows:

$$\begin{aligned} T_{binomial\_bcast}(P, m) &= \lceil \log_2 P \rceil \cdot (\alpha + m \cdot \beta), \\ T_{binary\_bcast}(P, m) &= 2 \cdot (\lceil \log_2 (P+1) \rceil - 1) \cdot (\alpha + m \cdot \beta). \end{aligned}$$

FIGURE 3 shows the performance of the binary tree and binomial tree algorithms using: a) the estimation by the existing analytical models; b) the experimental results on the Grisou cluster of the Grid'5000 platform; c) the estimation by the analytical models presented in Section III-A. It is evident that while the existing models wrongly predict that the binomial tree algorithm will outperform the binary tree algorithm on the target platform, our models correctly predict the relative performance of these algorithms.

IV. ESTIMATION OF MODEL PARAMETERS

A. INTRODUCTION TO THE ESTIMATION METHOD

In the most general case, the analytical model of an Open MPI collective algorithm uses three platform parameters – α, β, and $\gamma(p)$. The traditional state-of-the-art approach to estimation of α and β would be to find these parameters from a number of point-to-point communication experiments. Namely, the time of a round-trip of a message of size m, $RTT(m)$, is measured for a wide range of m. From these experiments, a system of linear equations with α and β as unknowns is derived. Then, linear regression is applied to find α and β. The found values of α and β would be then used in all analytical predictive formulas.

This approach yields a unique single pair of (α, β) for each target platform. Unfortunately, with α and β found this way, not all our analytical formulas will be accurate enough to be used for accurate selection of the best performing collective algorithm. Using non-linear regression does not improve the situation as the function $RTT(m)$ is typically near linear. Therefore, we propose to estimate the model parameters separately for each collective algorithm. More specifically, we propose to design a specific communication experiment for each collective algorithm, so that the algorithm itself would be involved in the execution of the experiment. Moreover, the execution time of this experiment must be dominated by the execution time of this collective algorithm. Then, we conduct a number of experiments on the target platform for a range of numbers of processors, p, and message sizes, m. From those experiments, we can derive a sufficiently large number of equations with α, β, and $\gamma(p)$ as unknowns, and then use an appropriate solver to find their values.

Unfortunately, when applied straightforwardly, this approach yields a system of *non-linear* equations like the one shown in FIGURE 4. This nonlinearity makes the task of estimation of the parameters mathematically very difficult, because we need to solve a large system of nonlinear equations.

$$\begin{cases} (\frac{P-1}{K} + \gamma(K+1)\cdot n_{s_1} - 1)\cdot(\alpha + \frac{m_1}{n_{s_1}}\cdot\beta) + (P-1)\cdot(\alpha + m_{g_1}\cdot\beta) = T_1 \\ (\frac{P-1}{K} + \gamma(K+1)\cdot n_{s_2} - 1)\cdot(\alpha + \frac{m_2}{n_{s_2}}\cdot\beta) + (P-1)\cdot(\alpha + m_{g_2}\cdot\beta) = T_2 \\ \dots \\ (\frac{P-1}{K} + \gamma(K+1)\cdot n_{s_M} - 1)\cdot(\alpha + \frac{m_M}{n_{s_2}}\cdot\beta) + (P-1)\cdot(\alpha + m_{g_M}\cdot\beta) = T_M \end{cases}$$

$$\Downarrow$$

$$\begin{cases} \alpha + \frac{(\frac{P-1}{K}+\gamma(K+1)\cdot n_{s_1}-1)\cdot\frac{m_1}{n_{s_1}}+(P-1)\cdot m_{g_1}}{\frac{(P-1)\cdot(1+K)}{K}+\gamma(K+1)\cdot n_{s_1}-1}\cdot\beta = \frac{T_1}{\frac{(P-1)\cdot(1+K)}{K}+\gamma(K+1)\cdot n_{s_1}-1} \\ \alpha + \frac{(\frac{P-1}{K}+\gamma(K+1)\cdot n_{s_2}-1)\cdot\frac{m_2}{n_{s_2}}+(P-1)\cdot m_{g_2}}{\frac{(P-1)\cdot(1+K)}{K}+\gamma(K+1)\cdot n_{s_2}-1}\cdot\beta = \frac{T_2}{\frac{(P-1)\cdot(1+K)}{K}+\gamma(K+1)\cdot n_{s_2}-1} \\ \dots \\ \alpha + \frac{(\frac{P-1}{K}+\gamma(K+1)\cdot n_{s_M}-1)\cdot\frac{m_M}{n_{s_M}}+(P-1)\cdot m_{g_M}}{\frac{(P-1)\cdot(1+K)}{K}+\gamma(K+1)\cdot n_{s_M}-1}\cdot\beta = \frac{T_M}{\frac{(P-1)\cdot(1+K)}{K}+\gamma(K+1)\cdot n_{s_M}-1} \end{cases}$$

FIGURE 4. A system of M non-linear equations with α, β, and $\gamma(K+1)$ as unknowns, derived from M communication experiments, each consisting of the execution of the K-Chain tree broadcast algorithm, broadcasting a message of size m_i ($i = 1, \dots, M$) from the root to the remaining $P-1$ processes, followed by the linear gather algorithm without synchronisation, gathering messages of size m_{g_i} ($m_{g_i} < eager\_limit$; $m_{g_i} \neq m_s$) on the root. The execution times, T_i, of these experiments are measured on the root. Given $\gamma(K+1)$ is evaluated separately, the system becomes a system of M linear equations with α and β as unknowns.

Our approach to this problem is the following. As the non-linearity is caused by the multiplicative terms involving $\gamma(p)$, we separate the estimation of $\gamma(p)$ from the estimation of α and β. Namely, we assume that $\gamma(p)$ is algorithm-independent and design a separate communication experiment for its estimation. The values of $\gamma(p)$ found from this experiment are then used as known coefficients in the algorithm-specific systems of equations for α and β. We present this approach in Sections IV-B and IV-C.

B. ESTIMATION OF $\gamma(p)$

The model parameter $\gamma(p)$ appears in the formula estimating the execution time of the linear tree broadcast algorithm with non-blocking communication, which is only used for broadcasting of a segment in the tree-based segmented broadcast algorithms. Thus, in the context of Open MPI, the linear tree broadcast algorithm with non-blocking communication will always broadcast a message of size m_s to a relatively small number of processes.

According to Formula 4,

$$\gamma(p) = \frac{T^{nonblock}_{linear\_bcast}(p, m_s)}{T_{p2p}(m_s)} = \frac{T^{nonblock}_{linear\_bcast}(p, m_s)}{T^{nonblock}_{linear\_bcast}(2, m_s)}.$$

Therefore, in order to estimate $\gamma(p)$ for a given range of the number of processes, $p \in \{2, \dots, P\}$, we need a method for estimation of $T^{nonblock}_{linear\_bcast}(p, m_s)$. We use the following method:

- For each $p \in \{2, \dots, P\}$, we measure on the root the execution time $T_1(p, N)$ of N successive calls to the *linear tree with non-blocking communication* broadcast routine separated by barriers. The routine broadcasts a message of size m_s.
- We estimate $T^{nonblock}_{linear\_bcast}(p, m_s)$ as $T_2(p) = \frac{T_1(p,N)}{N}$.

The experimentally obtained discrete function $\frac{T_2(p)}{T_2(2)}$ is used as a platform-specific but algorithm-independent estimation of $\gamma(p)$.

From our experiments, we observed that the discrete estimation of $\gamma(p)$ is near linear. Therefore, as an alternative for platforms with very large numbers of processors, we can build by linear regression a linear approximation of the discrete function $\frac{T_2(p)}{T_2(2)}$, obtained for a representative subset of the full range of p, and use this linear approximation as an analytical estimation of $\gamma(p)$.

C. ESTIMATION OF ALGORITHM SPECIFIC α AND β

To estimate the model parameters α and β for a given collective algorithm, we design a communication experiment, which starts and finishes on the root (in order to accurately measure its execution time using the root clock), and involves the execution of the modelled collective algorithm so that the total time of the experiment would be dominated by the time of its execution.

For example, for all broadcast algorithms, the communication experiment consists of a broadcast of a message of size m (where m is a multiple of segment size m_s), using the modelled broadcast algorithm, followed by a *barrier*, which is then followed by a linear-without-synchronisation gather algorithm, gathering messages of size m_g ($m_g < eager\_limit$; $m_g \neq m_s$) on the root. *Barriers* are also called in the beginning and in the end of the experiment, and the clock is started on the root after the first barrier and stopped after the last one. The execution time of this experiment on p nodes, $T_{bcast\_exp}(p, m)$, can be estimated as follows:

$$T_{bcast\_exp}(p, m) = T_{bcast\_alg}(p, m) + T_{linear\_gather}(p, m_g) \quad (16)$$

Using analytical formulas from Section III for $T_{bcast\_alg}(p, m)$ and $T_{linear\_gather}(p, m_g)$, this experiment will yield one linear equation with α and β as unknowns for each combination of p, m and m_g. By repeating this experiment with different p, m and m_g, we obtain a system of linear equations for α and β. Each equation in this system can be represented in the canonical form, $\alpha + \beta \times m_i = T_i$ $(i = 1, \ldots, M)$. Finally, we use the least-square regression to find α and β, giving us the best linear approximation $\alpha + \beta \times m$ of the discrete function $f(m_i) = T_i$ $(i = 1, \ldots, M)$.

FIGURE 4 shows a system of linear equations built for the K-Chain tree broadcast algorithm for our experimental platform. To build this system, we used the same P nodes in all experiments but varied the message sizes $m \in \{m_1, \ldots, m_M\}$ and $m_g \in \{m_{g_1}, \ldots, m_{g_M}\}$. With M different pairs of message sizes, we obtained a system of M equations. The number of nodes, P, was approximately equal to the half of the total number of nodes. We observed that the use of larger numbers of nodes in the experiments will not change the estimation of α and β.

V. EXPERIMENTAL RESULTS AND ANALYSIS

This section presents experimental evaluation of the proposed approach to selection of optimal collective algorithms using Open MPI broadcast and gather operations. In all experiments. we use the default Open MPI configuration (without any collective optimization tuning).

A. EXPERIMENT SETUP

For experiments, we use Open MPI 3.1 in default configuration running on a dedicated Grisou and Gros clusters of the Nancy site of the Grid‘5000 infrastructure [13]. The Grisou cluster consists of 51 nodes each with 2 Intel Xeon E5-2630 v3 CPUs (8 cores/CPU), 128GB RAM, 2 × 558GB HDD, interconnected via 10Gbps Ethernet. The Gros cluster consists of 124 nodes each with Intel Xeon Gold 5220 (18 cores/CPU), 96GB RAM, 894 GB SSD, interconnected via 2 x 25Gb Ethernet.

To make sure that the experimental results are reliable, we follow a detailed methodology: 1) We make sure that the cluster is fully reserved and dedicated to our experiments. 2) For each data point in the execution time of collective algorithms, the sample mean is used, which is calculated by executing the application repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. We also check that the individual observations are independent and their population follows the normal distribution. For this purpose, MPIBlib [34] is used.

In our communication experiments, MPI programs use the one-process-per-CPU configuration, and the maximal total number of processes is equal to 90 on Grisou and 124 on Gros clusters. The message segment size, m_s, for segmented broadcast algorithms is set to 8KB and is the same in all experiments. This segment size is commonly used for segmented broadcast algorithms in Open MPI. Selection of optimal segment size is out of the scope of this paper.

B. EXPERIMENTAL ESTIMATION OF MODEL PARAMETERS

First of all, we would like to stress again that we estimate model parameters for each cluster separately.

Estimation of parameter $\gamma(p)$ for our experimental platforms follows the method presented in Section IV-B. With the maximal number of processes equal to 90 (Grisou) and 124 (Gros), the maximal number of children in the linear tree broadcast algorithm with non-blocking communication, used in the segmented Open MPI broadcast algorithms, will be equal to seven. Therefore, the number of processes in our communication experiments ranges from 2 to 7 for both clusters. By definition, $\gamma(2) = 1$. The estimated values of $\gamma(p)$ for p from 3 to 7 are given in TABLE 6.

After estimation of $\gamma(p)$, we conduct communication experiments to estimate algorithm-specific values of parameters α and β for six broadcast algorithms and three gather algorithms following the method described in Section IV-C. In these experiments we use 40 processes on Grisou and 124 on Gros. The message size, m, varies in the range from 8KB to 4MB in the broadcast experiments, and from 64KB to 1MB in the gather experiments. We use 10 different sizes for broadcast algorithms, $\{m_i\}_{i=1}^{10}$, and 5 different sizes for gather algorithms, $\{m_i\}_{i=1}^{5}$, separated by a constant step in the logarithmic scale, $\log m_{i-1} - \log m_i = const$. Thus, for each collective algorithm, we obtain a system of 10 linear equations with α and β as unknowns. We use the Huber regressor [42] to find their values from the system.

The values of parameters α and β obtained this way can be found in TABLE 4 and TABLE 5. We can see that the values of α and β do vary depending on the collective algorithm, and the difference is more significant between algorithms implementing different collective operations. The results support our original hypothesis that the average execution time of a point-to-point communication will very much depend on the context of the use of the point-to-point communications in the algorithm. Therefore, the estimated values of the α and β capture more than just sheer network characteristics. One interesting example is the Split-binary tree and Binary tree broadcast algorithms. They both use the same virtual topology, but the estimated time of a point-to-point communication, $\alpha + \beta \times m$, is smaller in the context of the Split-binary one. This can be explained by a higher level of parallelism of the Split-binary algorithm, where a significant part of point-to-point communications is performed in parallel by a large number of independent pairs of processes from the left and right subtrees.

C. ACCURACY OF SELECTION OF OPTIMAL COLLECTIVE ALGORITHMS USING THE CONSTRUCTED ANALYTICAL PERFORMANCE MODELS

The constructed analytical performance models of the Open MPI broadcast and gather collective algorithms are designed for the use in the MPI_Bcast and MPI_Gather routines for runtime selection of the optimal algorithm, depending on the number of processes and the message size. While the

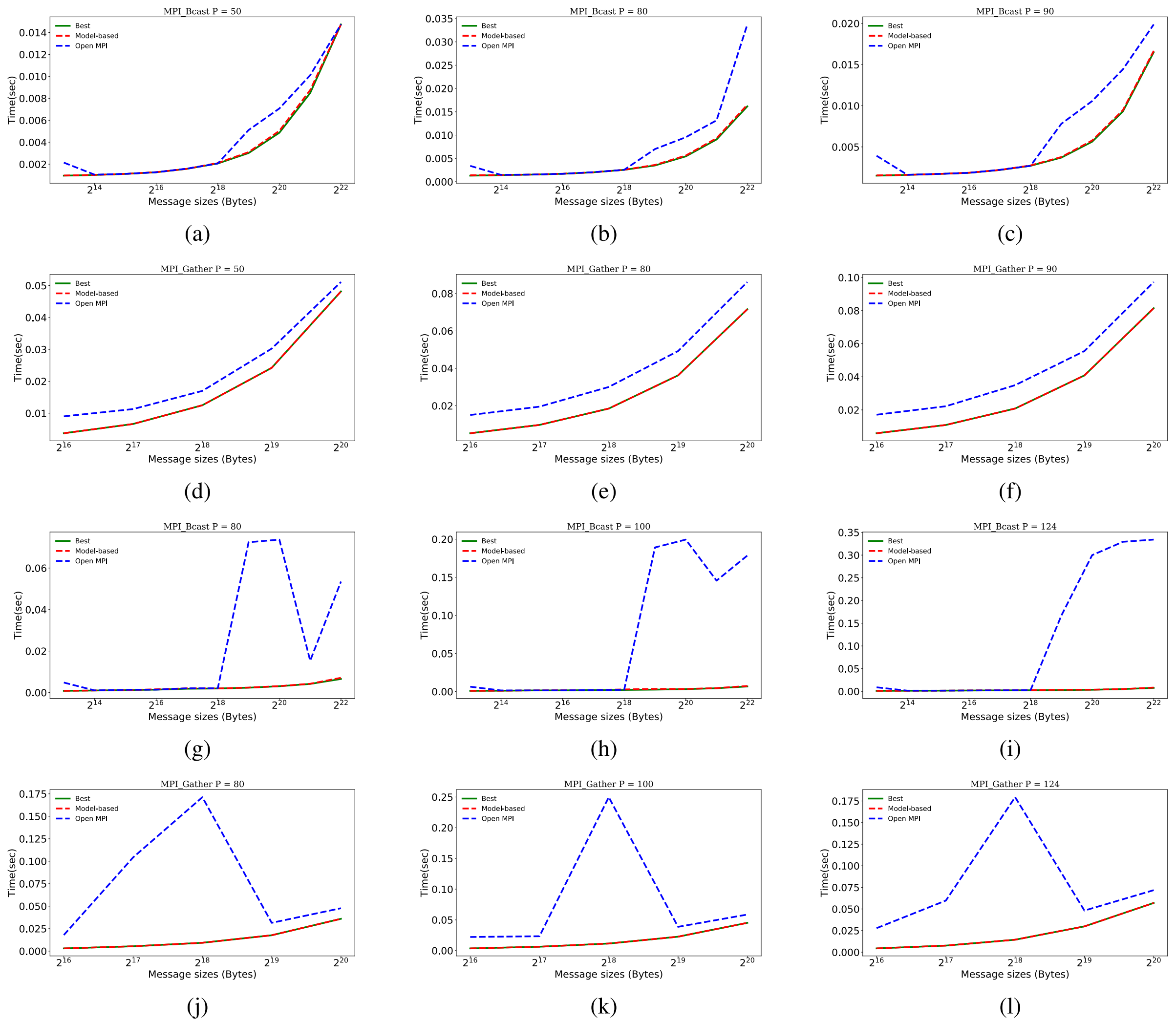


FIGURE 5. **Comparison of the selection accuracy of the Open MPI decision function and the proposed model-based method for MPI_Bcast and MPI_Gather. (5a - 5f) and (5g - 5l) present performance of collectives on Grisou and Gros clusters, respectively.**

efficiency of the selection procedure is evident from the low complexity of the analytical formulas derived in Section III, the experimental results on the accuracy are presented in this section.

FIGURE 5 shows the results of our experiments for MPI_Bcast and MPI_Gather. For both operations, we present results of experiments with 50, 80 and 90 processes on Grisou, and 80, 100 and 124 on Gros.

The message size, m, varies in the range from 8KB to 4MB in the broadcast experiments, and from 64KB to 1MB in the gather experiments. The reason to start from 8KB for MPI_Bcast is that in our experiments we use the same typical segment size of 8KB for all segmented broadcast algorithms employed in Open MPI. Therefore, the message sizes are multiples of 8KB in the experiments. We do not present results for smaller message sizes, where the communication time is latency bound, as this requires the construction of specific models, not including the β parameter.

We start from 64KB in the MPI_Gather experiments for the following reason. In Open MPI, the *rendezvous* point-to-point communication protocol is used for messages sizes $m \geq eager\_limit$ and the *eager* protocol for $m < eager\_limit$. The default value of *eager_limit* is 64KB. We use this default value of 64K in our experimental validation. Unlike broadcast algorithms, Open MPI gather algorithms do not employ message segmentation. This means that these algorithms will use the *eager* protocol for message sizes less than 64K and the *rendezvous* protocol for message sizes greater or equal to 64K. Therefore, in general, for each gather algorithm we have to build two models - one for the *eager* protocol and the other for the *rendezvous* protocol. In our experimental validation, due to a limited access to the experimental platforms, we only build models for one protocol. We picked *rendezvous*-based gather models for validation to balance the predominantly *eager*-based broadcast models.

We use 10 different sizes for broadcast algorithms, $\{m_i\}_{i=1}^{10}$, and 5 different sizes for gather algorithms, $\{m_i\}_{i=1}^{5}$, separated by a constant step in the logarithmic scale, $\log m_{i-1} - \log m_i = const$. To obtain data points for graphs in FIGURE 5, we use the MPIBlib routines [34], which measure

TABLE 2. Comparison of the model-based and Open MPI selections with the best performing MPI_Bcast algorithm. For each selected algorithm, its performance degradation against the optimal one is given in braces.

P=90, MPI_Bcast, Grisou

m (KB)	Best	Model-based (%)	Open MPI (%)
8	*binomial*	*binary* (3)	*split_binary* (160)
16	*binary*	*binary* (0)	*split_binary* (1)
32	*binary*	*binary* (0)	*split_binary* (0)
64	*split_binary*	*binary* (1)	*split_binary* (0)
128	*binary*	*binary* (0)	*split_binary* (1)
256	*split_binary*	*binary* (2)	*split_binary* (0)
512	*split_binary*	*binary* (2)	*chain* (111)
1024	*split_binary*	*binary* (3)	*chain* (88)
2048	*split_binary*	*binary* (2)	*chain* (55)
4096	*split_binary*	*binary* (1)	*chain* (20)

P=100, MPI_Bcast, Gros

m (KB)	Best	Model-based (%)	Open MPI (%)
8	*binary*	*binomial* (3)	*split_binary* (549)
16	*binomial*	*binomial* (0)	*split_binary* (32)
32	*binomial*	*binomial* (0)	*split_binary* (3)
64	*split_binary*	*binomial* (8)	*split_binary* (0)
128	*split_binary*	*binomial* (8)	*split_binary* (0)
256	*binary*	*binary* (0)	*split_binary* (6)
512	*binary*	*binary* (0)	*chain* (7297)
1024	*split_binary*	*binary* (7)	*chain* (6094)
2048	*split_binary*	*binary* (4)	*chain* (3227)
4096	*split_binary*	*binary* (9)	*chain* (2568)

TABLE 3. Comparison of the model-based and Open MPI selections with the best performing MPI_Gather algorithm. For each selected algorithm, its performance degradation against the optimal one is given in braces.

P=90, MPI_Gather, Grisou

m (KB)	Best	Model-based (%)	Open MPI (%)
64	*binomial*	*binomial* (0)	*linear_sync* (190)
128	*binomial*	*binomial* (0)	*linear_sync* (104)
256	*binomial*	*binomial* (0)	*linear_sync* (68)
512	*binomial*	*binomial* (0)	*linear_sync* (36)
1024	*binomial*	*binomial* (0)	*linear_sync* (19)

P=100, MPI_Gather, Gros

m (KB)	Best	Model-based (%)	Open MPI (%)
64	*binomial*	*binomial* (0)	*linear_sync* (524)
128	*binomial*	*binomial* (0)	*linear_sync* (271)
256	*binomial*	*binomial* (0)	*linear_sync* (2073)
512	*binomial*	*binomial* (0)	*linear_sync* (71)
1024	*binomial*	*binomial* (0)	*linear_sync* (36)

TABLE 4. Estimated values of α and β for the Grisou cluster and Open MPI broadcast and gather algorithms.

Collective algorithm	$\alpha(sec)$	$\beta\ (\frac{sec}{byte})$
Broadcast		
Linear tree	2.2×10^{-12}	1.8×10^{-8}
K-Chain tree	5.7×10^{-13}	4.7×10^{-9}
Chain tree	6.1×10^{-13}	4.9×10^{-9}
Split-binary tree	3.7×10^{-13}	3.6×10^{-9}
Binary tree	5.8×10^{-13}	4.7×10^{-9}
Binomial tree	5.8×10^{-13}	4.8×10^{-9}
Gather		
Linear tree without synchronisation	1.5×10^{-5}	1.5×10^{-9}
Binomial tree	1.2×10^{-4}	8.6×10^{-10}
Linear tree with synchronisation	5.9×10^{-5}	9.4×10^{-10}

TABLE 5. Estimated values of α and β for the Gros cluster and Open MPI broadcast and gather algorithms.

Collective algorithm	$\alpha(sec)$	$\beta\ (\frac{sec}{byte})$
Broadcast		
Linear tree	1.4×10^{-12}	1.1×10^{-8}
K-Chain tree	5.4×10^{-13}	4.5×10^{-9}
Chain tree	4.7×10^{-12}	3.8×10^{-8}
Split-binary tree	5.5×10^{-13}	4.5×10^{-9}
Binary tree	5.8×10^{-13}	4.7×10^{-9}
Binomial tree	1.2×10^{-13}	1.0×10^{-9}
Gather		
Linear tree without synchronisation	1.3×10^{-4}	3.4×10^{-10}
Binomial tree	1.0×10^{-4}	4.2×10^{-10}
Linear tree with synchronisation	9.5×10^{-5}	3.9×10^{-10}

TABLE 6. Estimated values of $\gamma(p)$ on Grisou and Gros clusters.

Number of processes (p)	$\gamma(p)$	
	Grisou	Gros
3	1.114	1.084
4	1.219	1.17
5	1.283	1.254
6	1.451	1.339
7	1.540	1.424

the execution time of each MPI_Bcast / MPI_Gather as follows. They call MPI_Barrier, start a clock on the root, call MPI_Bcast / MPI_Gather, call another MPI_Barrier, and stop the clock. No other operations occur in between. This communication experiment is repeated in a loop until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved, and the sample mean is returned as the measured time.

The graphs show the execution time of the collective operation as a function of message size. Each data point on a blue line shows the performance of the algorithm selected by the Open MPI decision function for the given operation, number of processes and message size. Each point on a red line shows the performance of the algorithm selected by our decision function, which uses the constructed analytical models. Each point on a green line shows the performance of the best Open

MPI algorithm for the given collective operation, number of processes and message size.

TABLE 2 presents selections made for MPI_Bcast using the proposed model-based runtime procedure and the Open MPI decision function. For each message size m, the best performing algorithm, the model-based selected algorithm, and the Open MPI selected algorithm are given. For the latter two, the performance degradation in percentages in comparison with the best performing algorithm is also given. We can see that for the Grisou cluster, the model-based selection either picks the best performing algorithm, or the algorithm, the performance of which deviates from the best no more than 3%. Given the accuracy of measurements, this means that the model-based selection is practically always optimal as the performance of the selected algorithm is indistinguishable from the best performance. The Open MPI selection is near optimal in 50% cases and causes significant, up to 160%, degradation in the remaining cases. For the Gros cluster, the model-based selection picks either the best performing algorithm or the algorithm with near optimal performance, no worse than 10% in comparison with the best performing algorithm. At the same time, while near optimal in 40% cases, the algorithms selected by the Open MPI demonstrate catastrophic degradation (up to 7297%) in 50% cases.

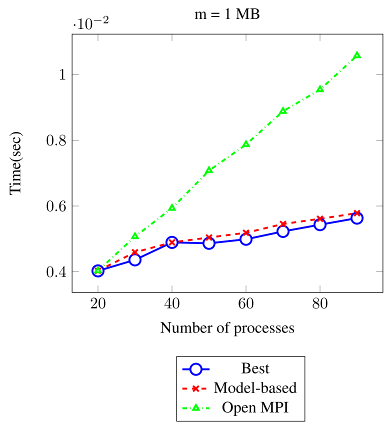
TABLE 3 shows results for MPI_Gather. One can see that the Open MPI selection significantly degrades the performance of MPI_Gather for all message sizes and numbers of processes (up to 190% on Grisou, and up to 2073% on Gros), while our model-based selection always picks the best performing algorithm.

The Open MPI decision functions select the algorithm depending on the message size and the number of processes. For example, the Open MPI broadcast decision function, shown in Listing 1, selects the chain broadcast algorithm for large message sizes. However, from TABLE 2 it is evident that chain broadcast algorithm is not the best performing algorithm for large message sizes on both clusters. From the same table, one can see that the model-based selection procedure accurately picks the best performing binomial tree broadcast algorithm for 16KB and 32KB message sizes on the Gros cluster, where Open MPI only selects the binomial tree algorithm for broadcasting messages smaller than 2KB.

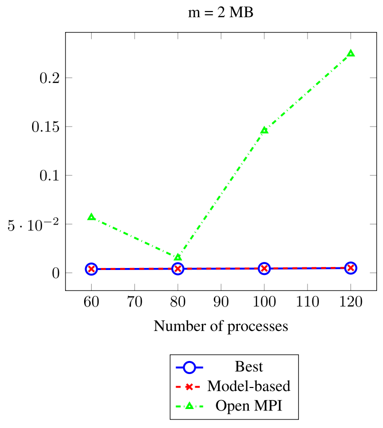
Finally, FIGURE 6 presents experimental results for MPI_Bcast when the message size is fixed but the number of processes is varying.

D. SHAHEEN II

While our experiments on Grid5000 clusters demonstrate the accuracy and the efficiency of the approach, the size of the clusters is relatively small. In this section, we demonstrate that the approach works for larger platforms as well. We present broadcast experiments for Shaheen II [14], a supercomputer owned by King Abdullah University of Science and Technology, Saudi Arabia. It consists of 6174 nodes (197568 cores) each with 2 Intel Haswell CPUs (16 cores per CPU, 2.3GHz), 128 GB of memory per node, Cray Aries



(a)



(b)

FIGURE 6. Comparison of the selection accuracy of the Open MPI decision function and the proposed model-based method for MPI_Bcast where message size is fixed and number of processes is varying. (6a) and (6b) present performance of MPI_Bcast on Grisou and Gros clusters respectively.

interconnect with Dragonfly topology. We could only use up to 512 nodes in our experiments.

We experiment with MPI programs running one process per CPU on 512 nodes. Thus, each program consists of 1024 MPI processes. The message segment size, m_s, for segmented broadcast algorithms is set to 8KB. The estimated values of parameters α and β can be found in TABLE 7.

FIGURE 7 shows the results of our experiments for MPI_Bcast. While P is fixed to 1024, the message size, m, varies from 8KB to 4MB. We use 10 different sizes,

TABLE 7. Estimated values of α and β for Shaheen II and open MPI broadcast algorithms.

Collective algorithm	$\alpha(sec)$	$\beta\,(\frac{sec}{byte})$
Broadcast		
Linear tree	2.6×10^{-12}	2.1×10^{-8}
K-Chain tree	2.5×10^{-13}	2.1×10^{-9}
Chain tree	2.1×10^{-13}	1.8×10^{-9}
Split-binary tree	3.4×10^{-13}	2.8×10^{-9}
Binary tree	3.1×10^{-13}	2.5×10^{-9}
Binomial tree	1.6×10^{-13}	1.3×10^{-9}

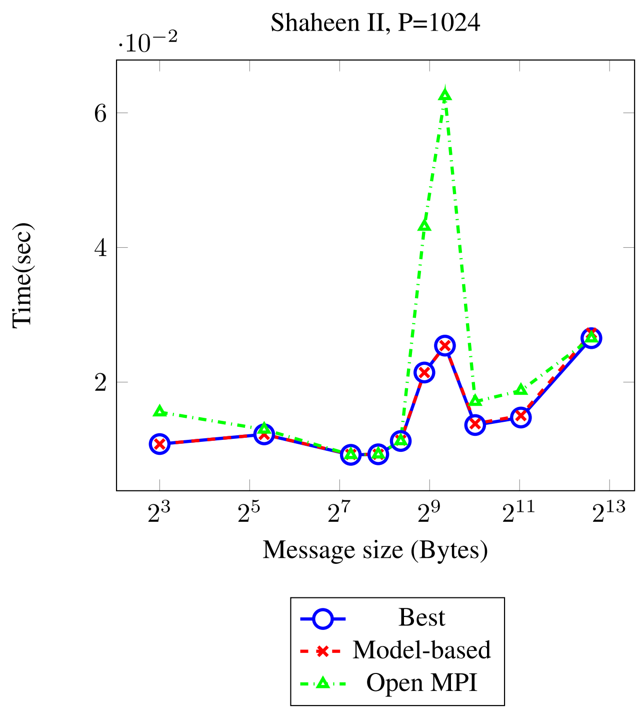


FIGURE 7. Comparison of the selection accuracy of the Open MPI decision function and the proposed model-based method for MPI_Bcast on Shaheen II supercomputer.

TABLE 8. Comparison of the model-based and Open MPI selections with the best performing MPI_Bcast algorithm. For each selected algorithm, its performance degradation against the optimal one is given in braces.

P=1024, MPI_Bcast, Shaheen II

m (KB)	**Best**	**Model-based (%)**	**Open MPI (%)**
8	*binomial*	*binomial* (0)	*split_binary* (44)
40	*binary*	*binary* (0)	*split_binary* (6)
152	*split_binary*	*split_binary* (0)	*split_binary* (0)
232	*binary*	*binary* (0)	*binary* (0)
328	*split_binary*	*binary* (1)	*split_binary* (0)
472	*binary*	*binary* (0)	*chain* (101)
648	*binary*	*binary* (0)	*chain* (146)
1032	*split_binary*	*binary* (1)	*chain* (25)
2088	*split_binary*	*binary* (1)	*chain* (27)
4840	*binary*	*binary* (0)	*chain* (16)

$m(KB) \in \{8, 40, 152, 232, 328, 472, 648, 1032, 2088, 4840\}$. The plot shows the execution time of the broadcast operation as a function of the message size. Each data point on the green line shows the performance of the algorithm selected by the Open MPI decision function for the given number of processes and message size. Each point on the red line shows the performance of the algorithm selected by our decision function, which uses the constructed analytical models. Each point on the blue line shows the performance of the best broadcast algorithm for MPI_Bcast.

TABLE 8 presents selections made for MPI_Bcast using the proposed model-based runtime procedure and the Open MPI decision function. For each message size m, the best performing algorithm, the model-based selected algorithm, and the Open MPI selected algorithm are given. For the latter two, the performance degradation in percentages in comparison with the best performing algorithm is also given. We can see that the model-based selection picks the best performing algorithm, or the algorithm, the performance of which deviates from the best no more than 1%. Given the accuracy of measurements, this means that the model-based selection is practically always optimal as the performance of the selected algorithm is indistinguishable from the best performance. The Open MPI selection is near optimal in 30% cases and causes significant, up to 146%, degradation in the remaining cases.

VI. DISCUSSION

In this section, we briefly discuss some limitations of the presented work and their impact.

First, we assume that communication is congestion-free. Network congestion is important for many distributed memory applications. To capture the performance of the collective algorithms with network congestion, the latter should be modelled using additional parameter(s) in the models. Network congestion was however negligible in our experimental setups, using one-process-per-CPU configurations of MPI programs. Currently, we are working on modelling network congestion to accurately capture the performance of collective algorithms in situations where network congestion has a notable impact on their execution time.

Second, our approach assumes that collectives are implemented through calls to point-to-point communication operations. We do not consider MPI implementations that exploit hardware collective support to perform certain collectives, for example, multicast, in O(1).

Third, we assume that the segment size, m_s, is fixed and the same in all collective algorithms. This limitation can be eased by making the segment size another decision variable with values from a small discreet set, say, $\{8K, 16K, 32K, 64K, 128K\}$. For each collective algorithm, we can build a separate model for each value of the segment size and use the models at runtime to select the fastest combination of the algorithm and segment size for each collective operation.

Fourth, we assume that the values of model parameters, such as α and β, do not depend on the number of processes, P, executing the algorithm. While this assumption did not negatively affect the selective accuracy of the models in our experimental setups, it may not be the case for larger platforms, able to run tens of thousands of MPI processes. For such platforms, one possible solution could be to break the total range of the number of processes into several segments and find the values of model parameters separately for each segment. There are other, more general possible solutions, but in order to study any possible solution, a regular access to a large-scale platform is needed. Unfortunately, the authors do not have such an access.

VII. CONCLUSION

In this paper, we proposed a novel model-based approach to automatic selection of optimal algorithms for MPI collective operations, which proved to be both efficient and accurate. The novelty of the approach is two-fold. First, we proposed to derive analytical models of collective algorithms from the code of their implementation rather than from high-level mathematical definitions. Second, we proposed to estimate model parameters separately for each algorithm, using a communication experiment, where the execution of the algorithm itself dominates the execution time of the experiment.

We also developed this approach into a detailed method and applied it to Open MPI 3.1 and its MPI_Bcast and MPI_Gather operations. We experimentally validated this method on two different clusters and one supercomputer and demonstrated its accuracy and efficiency. These results suggest that the proposed approach can be successful in the solution of the problem of accurate and efficient runtime selection of optimal algorithms for MPI collective operations.

ACKNOWLEDGMENT

Experiments presented in this article were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

REFERENCES

[1] *A Message-Passing Interface Standard*. Accessed: Mar. 8, 2021. [Online]. Available: https://www.mpi-forum.org/
[2] R. Rabenseifner, "Automatic profiling of MPI applications with hardware performance counters," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Lecture Notes in Computer Science), vol. 1697. Berlin, Germany: Springer, 1999, pp. 35–42.
[3] *MPICH—A Portable Implementation of MPI*. Accessed: Mar. 8, 2021. [Online]. Available: http://www.mpich.org/
[4] *Open MPI: Open Source High Performance Computing*. Accessed: Mar. 8, 2021. [Online]. Available: https://www.open-mpi.org/
[5] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 1, pp. 49–66, 2005.
[6] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds. Berlin, Germany: Springer, 2004, pp. 97–104.
[7] G. E. Fagg, J. Pjesivac-Grbovic, G. Bosilca, T. Angskun, J. Dongarra, and E. Jeannot, "Flexible collective communication tuning architecture applied to open MPI," in *Proc. Euro PVM/MPI*, 2006.
[8] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Comput.*, vol. 10, no. 2, pp. 127–143, 2007.
[9] R. W. Hockney, "The communication challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel Comput.*, vol. 20, no. 3, pp. 389–398, 1994.
[10] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," *ACM Sigplan Notices*, vol. 28, no. 7, pp. 1–12, Jul. 1993.
[11] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the LogP model—One step closer towards a realistic model for parallel computation," USA, Tech. Rep., 1995.
[12] T. Kielmann, H. E. Bal, and K. Verstoep, "Fast measurement of LogP parameters for message passing platforms," in *Parallel and Distributed Processing*, J. Rolim, Ed. Berlin, Germany: Springer, 2000, pp. 1176–1183.
[13] *Grid5000*. Accessed: Mar. 8, 2021. [Online]. Available: http://www.grid5000.fr
[14] *Shaheen II Supercomputer*. Accessed: Mar. 8, 2021. [Online]. Available: https://www.hpc.kaust.edu.sa/content/shaheen-ii
[15] E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. van de Geijn, "On optimizing collective communication," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2004, pp. 145–155.
[16] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: Theory, practice, and experience," *Concurrency Comput., Pract. Exp.*, vol. 19, no. 13, pp. 1749–1783, Sep. 2007.
[17] R. Rabenseifner and J. L. Träff, "More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Germany: Springer, 2004, pp. 36–46.
[18] P. Patarasuk, A. Faraj, and X. Yuan, "Pipelined broadcast on Ethernet switched clusters," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, Apr. 2006, p. 10.
[19] A. Lastovetsky, I.-H. Mkwawa, and M. O'Flynn, "An accurate communication model of a heterogeneous cluster based on a switch-enabled Ethernet network," in *Proc. 12th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, vol. 2, 2006, p. 6.
[20] T. Hoefler, C. Siebert, and W. Rehm, "A practically constant-time MPI broadcast algorithm for large-scale infiniband clusters with multicast," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Mar. 2007, pp. 1–8.
[21] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa, "LogP performance assessment of fast network interfaces," *IEEE Micro*, vol. 16, no. 1, pp. 35–43, Feb. 1996.
[22] T. V. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proc. 19th Annu. Int. Symp. Comput. Archit.*, 1992, pp. 256–266.
[23] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGP in theory and practice—An in-depth analysis of modern interconnection networks and benchmarking methods for collective operations," *Simul. Model. Pract. Theory*, vol. 17, no. 9, pp. 1511–1521, Oct. 2009.
[24] J. Pjesivac-Grbovic, "Towards automatic and adaptive optimizations of MPI collective operations," Ph.D. dissertation, Univ. Tennessee, Knoxville, TN, USA, Dec. 2007.
[25] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 11, pp. 1143–1156, Nov. 1997.
[26] J. Chen, L. Zhang, Y. Zhang, and W. Yuan, "Performance evaluation of allgather algorithms on terascale Linux cluster with fast Ethernet," in *Proc. 8th Int. Conf. High-Perform. Comput. Asia–Pacific Region (HPCASIA)*, 2005, p. 6 .
[27] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker, *MPI: The Complete Reference: The MPI Core*, vol. 1. Cambridge, MA, USA: MIT Press, 1998.
[28] J. Worringen, "Pipelining and overlapping for MPI collective operations," in *Proc. 28th Annu. IEEE Int. Conf. Local Comput. Netw. (LCN)*, Oct. 2003, pp. 548–557.
[29] R. Rabenseifner, "Optimization of collective reduction operations," in *Computational Science—ICCS 2004*. Springer, 2004, pp. 1–9.
[30] P. Sanders and J. L. Träff, "Parallel prefix (scan) algorithms for MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Germany: Springer, 2006, pp. 49–57.
[31] J.-A. Rico-Gallego, J.-C. Díaz-Martín, and A. L. Lastovetsky, "Extending τ-lop to model concurrent MPI communications in multicore clusters," *Future Gener. Comput. Syst.*, vol. 61, pp. 66–82, Aug. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X16300346
[32] A. Lastovetsky and V. Rychkov, "Building the communication performance model of heterogeneous clusters based on a switched network," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2007, pp. 568–575.
[33] A. Lastovetsky and V. Rychkov, "Accurate and efficient estimation of parameters of heterogeneous communication performance models," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 2, pp. 123–139, May 2009.

[34] A. Lastovetsky, V. Rychkov, and M. O'Flynn, "MPIBlib: Benchmarking MPI communications for parallel computing on homogeneous and heterogeneous clusters," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2008, pp. 227–238.

[35] J. Pješivac-Grbović, G. E. Fagg, T. Angskun, G. Bosilca, and J. J. Dongarra, "MPI collective algorithm selection and quadtree encoding," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2006, pp. 40–48.

[36] T. G. Dietterich and E. B. Kong, "Machine learning bias, statistical bias, and statistical variance of decision tree algorithms," Dept. Comput. Sci., Oregon State Univ., Corvallis, OR, USA, Tech. Rep., 1995.

[37] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "Decision trees and MPI collective algorithm selection problem," in *Euro-Par 2007 Parallel Processing*, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds. Berlin, Germany: Springer, 2007, pp. 107–117.

[38] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann, 1993.

[39] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, "Predicting MPI collective communication performance using machine learning," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2020, pp. 259–269.

[40] U. Wickramasinghe and A. Lumsdaine, "A survey of methods for collective communication optimization and tuning," 2016, *arXiv:1611.06334*. [Online]. Available: http://arxiv.org/abs/1611.06334

[41] K. Hasanov and A. Lastovetsky, "Hierarchical redesign of classic MPI reduction algorithms," *J. Supercomput.*, vol. 73, no. 2, pp. 713–725, Feb. 2017, doi: 10.1007/s11227-016-1779-7.

[42] P. J. Huber, "Robust estimation of a location parameter," in *Breakthroughs in Statistics*. Springer, 1992, pp. 492–518.

EMIN NURIYEV received the B.Sc. and M.Sc. degrees in applied mathematics from Baku State University, in 2005 and 2007, respectively, and the Ph.D. degree in computer science from University College Dublin, in 2021. His main research interests include algorithms and models for high-performance computing.

ALEXEY LASTOVETSKY received the Ph.D. degree from Moscow Aviation Institute, in 1986, and the D.Sc. (Habilitation) degree from Russian Academy of Sciences, in 1997. His main research interests include high performance heterogeneous computing and energy efficient computing. He has published over 150 articles in refereed journals, edited books, and international conferences. He has authored two monographs–*Parallel Computing on Heterogeneous Networks* (Wiley, 2003) and *High Performance Heterogeneous Computing*, with J. Dongarra, (Wiley, 2009).