Performance Analysis of MPI Collective Operations *

Jelena Pješivac–Grbović, Thara Angskun, George Bosilca,
Graham E. Fagg, Edgar Gabriel Jack J. Dongarra
Innovative Computing Laboratory, Computer Science Department,
University of Tennessee, 1122 Volunteer Blvd., Suite 413,
Knoxville, TN 37996-3450, USA
{pjesa, angskun, bosilca, fagg, egabriel, dongarra}@cs.utk.edu

Abstract

Previous studies of application usage show that the performance of collective communications are critical for high performance computing and are often overlooked when compared to the point-to-point performance. In this paper we attempt to analyze and improve collective communication in the context of the widely deployed MPI programming paradigm by extending accepted models of point-to-point communication, such as Hockney, LogP/LogGP, and PLogP. The predictions from the models were compared to the experimentally gathered data and our findings were used to optimize the implementation of collective operations in the FT-MPI library.

1 Introduction

Previous studies of application usage show that the performance of collective communications are critical to high performance computing (*HPC*). Profiling study [12] showed that applications spend more than eighty percent of a transfer time in collective operations. Given this fact, it is essential for MPI implementations to provide high-performance collective operations. However, collective operation performance is often overlooked when compared to the point-to-point performance. Collective

*This material is based upon work supported by the Department of Energy under Contract No. DE-FG02-02ER25536.

operations (*collectives*) encompass a wide range of possible algorithms, topologies, and methods. The optimal[1] implementation of a collective for a given system depends on many factors, including for example, physical topology of the system, number of processes involved, message sizes, and the location of the root node (where applicable). Furthermore, many algorithms allow explicit segmentation of the message that is being transmitted, in which case the performance of the algorithm also depends on the used segment size. Some collective operations involve local computation (e.g. reduction operations), in which case we also need to consider local characteristics of each node as they could affect our decision on how to overlap communication with computation.

Simple, yet time consuming way to find even a semi-optimal implementation of an individual collective operation is to run an extensive set of tests over a parameter space for the collective on a dedicated system. However, running such detailed tests even on relatively small clusters (32 - 64 nodes), can take a substantial amount of time [15][2]. If one were to analyze all of the MPI collectives in a similar manner, the tuning process could take days. Still, many of current MPI implementations use "extensive" testing to determine switching points between the algorithms. The decision of which algorithm to use is semi-static and based on predetermined parameters that do not model all possible target systems.

Alternatives to the static decisions include running a limited number of performance and system evaluation tests. This information can be combined with predictions from parallel communication models to make run-time decisions to select near-optimal algorithms and segment sizes for given operation, communicator, message size, and the rank of the root process.

There are many parallel communicational models that predict performance of any given collective operation based on standardized system parameters. Hockney [9], LogP [5], LogGP [1], and PLogP [10] models are frequently used to analyze parallel algorithm performance. Assessing the parameters for these models within local area network is relatively straightforward and the methods to approximate them have already been established and are well understood [6],[10].

The major contribution of this paper is the direct comparison of Hockney, LogP, LogGP, and

[1]We define "optimal implementation" in the following way: given a set of available algorithms for the collective, optimal implementation will use the best performing algorithm for the particular combination of parameters (message size, communicator size, root, etc.)

[2]For example, profiling the linear scatter algorithm on 8 nodes took more than three hours[15].

PLogP based parallel communication models applied to MPI collective operations. Indirectly, this work was used to implement and optimize the collective operation subsystem of the FT-MPI [7] library.

The rest of this paper proceeds as follows. Section 2 discusses related work. Section 3 examines parallel communication models of interest; Section 4 discusses the Optimized Collective Communication (OCC) library and explains some of the algorithms it currently provides; Section 5 provides details about the collective algorithm modeling; Section 6 presents the experimental evaluation of our study; and Section 7 is discussion and future work.

2 Related Work

Performance of MPI collective operations has been an active area of research in recent years. Important aspect of collective algorithm optimizations is understanding the algorithm performance in terms of different parallel communication models.

Thakur et al. [14] and Rabenseifner et al. [13] use Hockney model to analyze the performance of different collective operation algorithms. Kielmann et al. [11] use PLogP model to find optimal algorithm and parameters for collective operations incorporated in the MagPIe library. Bell et al. [2] use extensions of LogP and LogGP models to evaluate high performance networks. Bernaschi et al. [3] analyze the efficiency of reduce-scatter collective using LogGP model. Vadhiyar et al. [15] used a modified LogP model which took into account the number of pending requests that had been queued.

3 Parallel Communication Models

The parallel communication models (PCMs) are a basis for design and analysis of parallel algorithms. Good model includes the smallest possible number of parameters, but is able to capture complexity of the run-time system across a number of possible criteria sufficiently well. Since MPI collective operations consist of communication and computation part of the algorithm, we need to model both network and computation aspects of the system.

3.1 Modeling Network Performance

In modeling communication aspects of collective algorithms, we employ the models most-frequently used by the message-passing community:

Hockney Model

Hockney model [9] assumes that the time to send a message of size m between two nodes is $\alpha + \beta m$, where α is the latency for each message, and β is the transfer time per byte or reciprocal of network bandwidth. We altered Hockney model such that α and β are the function of message size. Congestion cannot be modeled using this model.

LogP/LogGP Models

LogP model [5] describes a network in terms of latency, L, overhead, o, gap per message, g, and number of nodes involved in communication, P. The time to send a message between two nodes according to LogP model is $L + 2o$. LogP assumes that only constant-size, small messages are communicated between the nodes. In this model, the network allows transmission of at most $\lfloor L/g \rfloor$ messages simultaneously.

LogGP [1] is an extension of the LogP model that additionally allows for large messages by introducing the gap per byte parameter, G. LogGP model predicts the time to send a message of size m between two nodes as $L + 2o + (m-1)G$.

In both LogP and LogGP model, the sender is able to initiate a new message after time g.

PLogP Model

PLogP model [10] is an extension of the LogP model. PLogP model is defined in terms of end-to-end latency L, sender and receiver overheads, $o_s(m)$ and $o_r(m)$ respectively, gap per message $g(m)$, and number of nodes involved in communication P. In this model sender and receiver overheads and gap per message depend on the message size.

Notion of latency and gap in PLogP model slightly differs from that of the LogP/LogGP model. Latency in PLogP model includes all contributing factors, such as copying data to and from network

interfaces, in addition to the message transfer time. Gap parameter in PLogP model is defined as the minimum time interval between consecutive message transmissions or receptions, implying that at all times $g(m) \geq o_s(m)$ and $g(m) \geq o_r(m)$.

Time to send a message of size m between two nodes in the PLogP model is $L + g(m)$. If $g(m)$ is a linear function of message size m and L excludes the sender overhead, then the PLogP model is identical to LogGP model which distinguishes between sender and receiver overheads.

3.2 Modeling Computation

We model the time spent in computation on message of size m as γm, where γ is computation time per byte. This linear model ignores effects caused by the access patterns and cache behavior, but is able to provide us with lower limit on time spent in computation.

4 Optimized Collective Communication Library

We have developed a framework for method verification and performance testing known as the Optimized Collective Communication library (OCC). OCC is an MPI collective library built on top of MPI's point-to-point operations. OCC consists of three modules: methods[3], verification, and performance-testing modules. The methods module provides a simple interface for addition of new collective algorithms. The verification module provides basic verification tools for the existing methods. The performance module provides measurement tools for the library.

Currently, the methods module contains various implementations of the following subset of MPI collective operations: MPI_Barrier, MPI_Bcast, MPI_Reduce, MPI_Scatter, and MPI_Alltoall.

4.1 Virtual Topologies

MPI collective operations can be classified as either one-to-many/many-to-one (single producer or consumer) or many-to-many (every participant is both producer and consumer) operations. For example, Broadcast, Reduce, Scatter(v), and Gather(v) follow one-to-many communication pattern,

[3]A method is defined by a control sequence (algorithm) and parameters for that control sequence, such as virtual topology and segment size.

while Barrier, Alltoall, Allreduce, and Allgather(v) employ many-to-many communication.

Generalized version of the one-to-many/many-to-one type of collectives can be expressed as *i)* receive data from preceding node(s), *ii)* process data (optional), *iii)* send data to succeeding node(s). The data flow for this type of algorithms is unidirectional. Virtual topologies can be used to determine the preceding and succeeding nodes in the algorithm.

OCC library currently supports five different virtual topologies: flat-tree/linear, pipeline (single chain), binomial tree, binary tree, and k-chain tree. Our tests show that given a collective operation, message size, and number of processes, each of the topologies can be beneficial for some combination of parameters.

4.2 Available Algorithms

This section describes the available methods in OCC for barrier, broadcast, reduce and alltoall operations. Due to space constraints and since it is outside of the scope of this paper, we are not discussing the algorithms in great details.

Barrier

Barrier is a collective operation used to synchronize a group of nodes. It guarantees that by the end of the operation, the remaining nodes have at least entered the barrier. We implemented four different algorithms for the Barrier collective: flat-tree/linear fan-in-fan-out, double ring, recursive doubling, and Bruck [4] algorithm.

In flat-tree/linear fan-in-fan-out algorithm all nodes report to a preselected root; once everyone has reported to the root, the root sends a releasing message to everyone. In the double ring algorithm, a zero byte message is sent from a preselected root circularly to the right. A node can leave Barrier only after it receives the message for the second time. Both linear and double ring algorithm require P communication steps. Bruck algorithm requires $\lceil log_2 P \rceil$ communication steps. At a step k, node r receives zero byte message from and sends message to rank $(r - 2^k)$ and $(r + 2^k)$ node (with wrap around) respectively. The recursive doubling algorithm requires $log_2 P$ steps if P is power of 2, and $\lfloor log_2 P \rfloor + 2$ steps if not. At step k, node r exchanges message with node (r XOR

2^k). If the number of nodes P is not power 2, we need two extra steps to handle extra nodes.

Broadcast

Broadcast operation broadcasts a message from the root process to all processes of the group. At the end of the call, the contents of the root's communication buffer is copied to all other processes. Since all nodes need to receive the same data, nodes who already received the data can be used as new data sources. We implemented the following algorithms for this collective: flat-tree/linear, pipeline, binomial tree, and binary tree. All of the algorithms allow for the message segmentation which allows us to overlap concurrent communications.

In flat-tree/linear algorithm root node sends individual message to all participating nodes. In pipeline algorithm, messages are propagated from the root left to right in a linear fashion. In binomial and binary tree algorithms, messages traverse the tree starting from the root, and going towards the leaf nodes through the intermediate nodes. Implementation of these algorithms overlaps receive and send operations such that if possible, at any given time, we have one incoming and one or many outgoing communication.

Reduce

Reduce operation combines the elements provided in the input buffer of each process in the group using the specified operation, and returns the combined value in the output buffer of the root process.

We have implemented generalized Reduce operation that can use all available virtual topologies: flat-tree/linear, pipeline, binomial tree, binary tree, and k-chain tree. At this time, OCC library works only with predefined MPI operations. As in the case of Broadcast, our actual implementation overlaps communication with computation and other communications.

Alltoall

Alltoall is used to exchange data among the all processes in the group. The operation is equivalent to all processes executing the scatter operation on their local buffer. We have implemented two

algorithms for this function: linear and pairwise exchange.

In linear algorithm for alltoall collective, at any step i, i^{th} node sends a message to all other nodes. The $(i+1)^{th}$ node is able to proceed and start sending as soon as it receives the complete message from the i^{th} node. We allow for segmentation of the message being sent. In pairwise exchange algorithm, at step i, node with rank r sends a message to node $(r+i)^{th}$ and receives message from $(r-i)^{th}$, with wrap around. We do not segment message in this algorithm. At any given step in this algorithm, a single incoming and outgoing communication exist at every node.

5 Modeling Collective Operations

For each of the implemented algorithms we have created a numeric reference model based on a point-to-point communication model (such as the Hockney, LogP/LogGP, and PLogP model). Due to the space constraint we show only formulas for Barrier, Reduce, and Alltoall collectives.

We assume the full-duplex network which allows us exchange and send-receive a message in the same amount of time as completing a single receive. Message segmentation allows us to divide a message of size m into a number of segments, n_s, of segment size m_s. If applicable, the formulas in Tables 1, 2, and 3 account for the message segmentation. In Hockney and PLogP models parameters depend on the message size. It is trivial to obtain LogP predictions from the LogGP formulas by setting the gap-per-byte parameter G to zero.

Table 1 summarizes performance equations for four different Barrier implementations. Table 2 summarizes equations used for predicting Reduce performance. The formulas for Reduce algorithms with $\gamma = 0$ are similar to the equations modeling Broadcast and Scatter performance over the same topology. Alltoall collective is modeled using formulas in Table 3.

6 Results

6.1 Experiment Setup

The measurements were obtained on a dedicated cluster at the SInGR center at University of Tennessee. The cluster consists of 32 Dell Precision 530s nodes, each with Dual Pentium IV Xeon

Barrier	Model	Duration
Flat Tree	*Hockney*	$T = (P-1) \times \alpha$
Flat Tree	*LogP/LogGP*	$T_{min} = (P-2) \times g + 2 \times (L + 2 \times o)$ $T_{max} = (P-2) \times (g+o) + 2 \times (L + 2 \times o)$
Flat Tree	*PLogP*	$T_{min} = P \times g + 2 \times L$ $T_{max} = P \times (g + o_r) + 2 \times (L - o_r)$
Double Ring	*Hockney*	$T = 2 \times P \times \alpha$
Double Ring	*LogP/LogGP*	$T = 2 \times P \times (L + o + g)$
Double Ring	*PLogP*	$T = 2 \times P \times (L + g)$
Recursive Doubling	*Hockney*	$T = \log_2(P) \times \alpha$,if P is exact power of 2 $T = (\log_2(P) + 2) \times \alpha$,otherwise
Recursive Doubling	*LogP/LogGP*	$T = \log_2(P) \times (L + o + g)$,if P is exact power of 2 $T = (\lfloor \log_2(P) \rfloor + 2) \times (L + o + g)$,otherwise
Recursive Doubling	*PLogP*	$T = \log_2(P) \times (L + g)$,if P is exact power of 2 $T = (\lfloor \log_2(P) \rfloor + 2) \times (L + g)$,otherwise
Bruck	*Hockney*	$T = \lceil \log_2(P) \rceil \times \alpha$
Bruck	*LogP/LogGP*	$T = \lceil \log_2(P) \rceil \times (L + o + g)$
Bruck	*PLogP*	$T = \lceil \log_2(P) \rceil \times (L + g)$

Table 1: **Analysis of different Barrier algorithms** The predictions for *LogP* and *LogGP* are identical since Barrier calls exchange zero-length messages.

Reduce	Model	Duration
Flat Tree	*Hockney*	$T = n_s \times (P-1) \times (\alpha + \beta m_s + \gamma m_s)$
Flat Tree	*LogP/LogGP*	$T = o + (m_s - 1)G + L +$ $n_s \times \max\{g, (P-1) \times (o + (m_s-1)G + \gamma m_s)\}$
Flat Tree	*PLogP*	$T = L + (P-1) \times n_s \times \max\{g(m_s), o_r(m_s) + \gamma m_s\}$
Pipeline	*Hockney*	$T = (P + n_s - 2) \times (\alpha + \beta m_s + \gamma m_s)$
Pipeline	*LogP/LogGP*	$T = (P-1) \times (L + 2 \times o + (m_s-1)G + \gamma m_s) +$ $(n_s - 1) \times \max\{g, 2 \times o + (m_s-1)G + \gamma m_s\}$
Pipeline	*PLogP*	$T = (P-1) \times (L + \max\{g(m_s), o_r(m_s) + \gamma m_s\}) +$ $(n_s - 1) \times (\max\{g(m_s), o_r(m_s) + \gamma m_s\} + o_s(m_s))$
Binomial	*Hockney*	$T = \lceil n_s \times log_2(P) \rceil \times (\alpha + \beta m_s + \gamma m_s)$
Binomial	*LogP/LogGP*	$T = \lceil log_2 P \rceil \times \begin{pmatrix} (n_s - 1) \times \max\{(m_s-1)G + g, o + \gamma m_s\} + \\ o + L + \max\{(m_s-1)G, \gamma m_s\} \end{pmatrix}$
Binomial	*PLogP*	$T = \lceil log_2 P \rceil \times (L + n_s \times \max\{g(m_s), o_r(m_s) + \gamma m_s\})$
Binary	*Hockney*	$T = (\lceil \log_2(P+1) \rceil + n_s - 2) \times (\alpha + \beta m_s)$
Binary	*LogP/LogGP*	$T = (\lceil log_2(P+1) \rceil - 1) \times ((L + 3 \times o + (m_s-1)G + 2\gamma m_s) +$ $(n_s - 1) \times ((m_s-1)G + \max\{g, 3o + 2 \times \gamma m_s\}))$
Binary	*PLogP*	$T = (\lceil log_2(P+1) \rceil - 1) \times (L + 2 \times \max\{g(m_s), o_r(m_s) + \gamma m_s\}) +$ $(n_s - 1) \times (o_s(m_s) + 2 \times \max\{g(m_s), o_r(m_s) + \gamma m_s\})$

Table 2: **Analysis of different Reduce algorithms**.

Alltoall	Model	Duration
Linear	*Hockney*	$T = P \times (\alpha + \beta m_s) + (P-1) \times (n_s \times P + 1 - \frac{P}{2}) \times \alpha$
Linear	*LogP/LogGP*	$T = P \times (L + 2 \times o) + (P-1) \times (n_s \times P + 1 - \frac{P}{2}) \times (g + (m_s - 1)G)$
Linear	*PLogP*	$T = P \times L + (P-1) \times (n_s \times P + 1 - \frac{P}{2}) \times g(m_s)$
Pairwise exchange	*Hockney*	$T = (P-1) \times (\alpha + \beta m)$
Pairwise exchange	*LogP/LogGP*	$T = (P-1) \times (L + o + (m-1) \times G + g)$
Pairwise exchange	*PLogP*	$T = (P-1) \times (L + g(m))$

Table 3: **Analysis of different Alltoall algorithms**. In pairwise exchange formulas the time required to perform a local copy operation between two buffers is ignored.

2.4 GHz processors, 512 KB Cache, 2 GB Ram, connected via Gigabit Ethernet.

Model Parameters

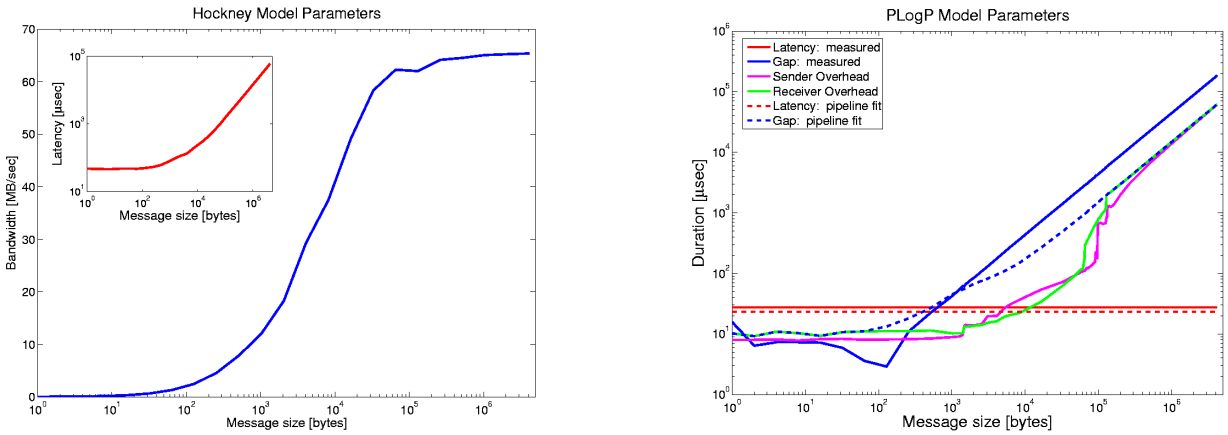
We measured model parameters using different MPI implementation. Most of the collected data was generated using FT-MPI [7], MPICH-1, and MPICH-2 [8][4]. Parameter values measured using MPICH-1 had higher latency and gap values with lower bandwidth than both FT-MPI and MPICH-2. FT-MPI and MPICH-2 had similar values for these parameters.

Hockney model parameters were measured directly using point-to-point tests. Figure 1(a) shows the measured values.

To measure PLogP model parameters we used the `logp_mpi` software suite provided by Kielmann et al. [10]. Parameter values were obtained by averaging the values obtained between different communication points in the same system. Measured values of LogP and LogGP were obtained from the PLogP values as explained in [10].

For LogP, LogGP, and PLogP models, we also experimented with direct parameter fitting to the experimental data, and applying those parameter values to model other collective operations. Parameter fitting was done under the assumption that sender and receiver overheads do not depend on the network behavior and that they can be measured correctly using the `log_mpi` library. In this paper, fitted parameters were obtained by analyzing the performance of non-segmented pipelined broadcast algorithm over various sizes for communicators and messages. We chose to fit model

[4]In our measurements we used MPICH-1.2.6 and MPICH-2 0.97



(a) (b)

Figure 1: **Parameter values**: (a) Hockney model and (b) PLogP model (MPICH-2).

parameters to non-segmented pipelined broadcast's data because the communication pattern of this algorithm (linear sending and receiving message) is the closest match to the point-to-point tests used to measure model parameters in the `logp_mpi` and similar libraries.

Figure 1(b) depicts measured and fitted parameter values for the PLogP model. Values of LogP and LogGP parameters are summarized in Table 4.

LogP/LogGP		**Measured**		**Fitted Pipeline**	
Latency	L	26.68	$[\mu sec]$	30.45	$[\mu sec]$
Overhead	o	8.15	$[\mu sec]$	8.15	$[\mu sec]$
Gap	g	15.6	$[\mu sec]$	8.683	$[\mu sec]$
Gap-per-byte	G	0.044	$[\frac{\mu sec}{byte}]$	0.015	$[\frac{\mu sec}{byte}]$

Table 4: **LogP/LogGP model parameters** (MPICH-2).

Performance Tests

Our performance measuring methodology is described in Table 5. Report-to-root step is necessary to minimize the effects of pipelining. The steps described above are performed on communicators of the same size, and the root node is changed for every measurement.

Each of the collected data points is an average value of 10-20 measurements in which minimum and maximum values are excluded, and the standard deviation was less than 5% of the remaining

```
measure RTR time (time to report to root on this communicator)
for every test on this communicator
   create send and receive buffers
   for each sample point
      for each function
          START measuring on root
          repeat numtimes times
             call function
             report to root
          STOP measuring on root
          save time = (STOP - START - numtimes*RTR)/numtimes;
   for each function
      find minimum, average, and standard deviation of saved times
```

Table 5: **Performance measuring methodology**

points.

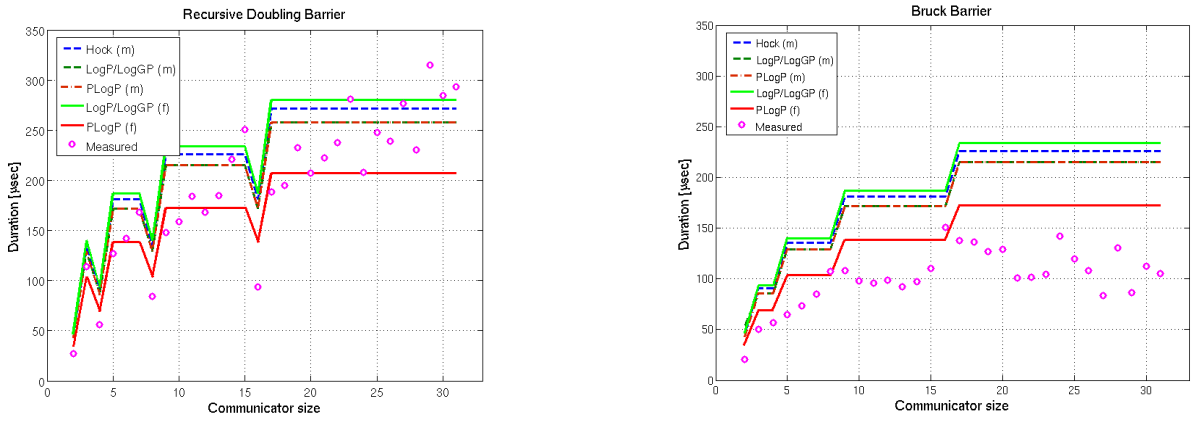
6.2 Empirical results

We executed performance tests for different Barrier, Broadcast, Reduce, Scatter, and Alltoall collective operations implementations using FT-MPI, MPICH, and MPICH-2. We analyzed the algorithm performance and the optimal implementation of different collective operations. When predicting performance of collectives that exchange actual data (message size $>$ 0b) we did not consider pure LogP predictions, but used LogGP instead.

We found that the worst case for the algorithm performance is often too pessimistic. For example, in flat-tree/linear fan-in-fan-out barrier algorithm, root posts $(P-1)$ non-blocking receive operations, followed by MPI_Waitall call. At the same time, non-root nodes send a zero-byte message to the root. The worst case for the algorithm performance assumes that the messages will be delivered to the root in sequential fashion. MPI implementations can optimize this process and deliver all messages at the same time with very small overhead. Our experience with the MPI implementations was that the algorithms performance was generally closer to the best case scenario. Thus, in our results, we chose to model algorithm performance using the best case scenario.

Figure 2 illustrates measured and predicted performance of recursive doubling and bruck barrier algorithms. Experimental data for both algorithms, while exhibiting trends, is not uniform. One of the possible explanations for this is that the "report-to-root" step in performance measurement

takes an amount of time that is comparable to the duration of both algorithms. Thus, variations in the time it takes to perform "report-to-root" on a given communicator can affect the reported value for the barrier more significantly than for other collective operations which take proportionally more time. Since algorithms exchanges zero-length messages, predictions of LogP and LogGP are identical. For measured parameters, predictions of LogP/LogGP and PLogP are very similar. This is not the case for fitted parameters because the fitted latency for LogP/LogGP model is around 20% higher than the measured one and the fitted latency for PLogP model is around 20% lower than the measured one. However, given the experimental data, we believe that the models were able to capture gross performance of these two Barrier algorithms.

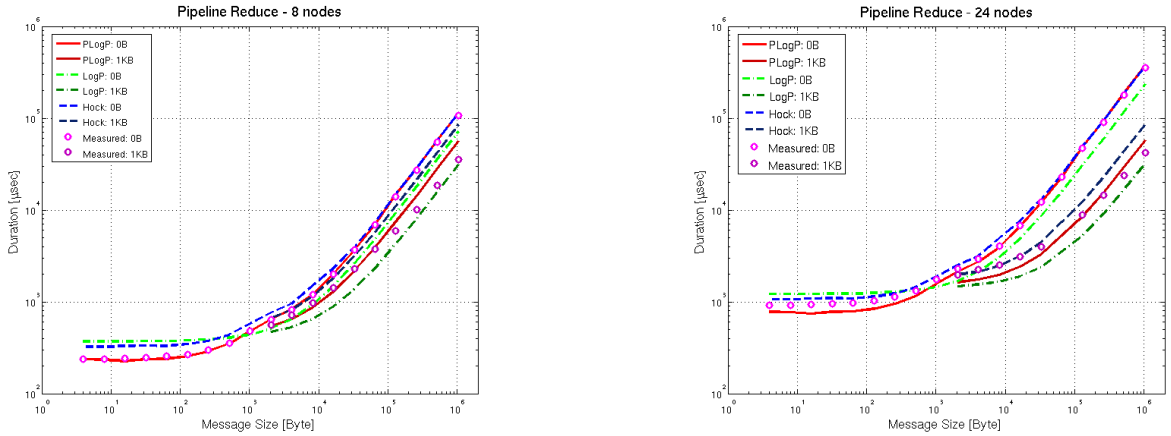


(a) (b)

Figure 2: **Performance of Barrier algorithms**: (a) Recursive Doubling and (b) Bruck. Experimentally measured values are represented using circles. Dashed lines show predictions made using measured parameter values for all three models, while solid lines represent the predictions of LogP/LogGP and PLogP using the fitted parameters (MPICH-2).

Our experiments show that the segmented pipeline algorithms perform well for large messages. The formulas for modeling segmented pipelined reduce given in Table 2 agree with that observation: as the number of segments n_s increases, the term that depends on number of processes P becomes less significant. In the limiting case, segmented pipeline reduce takes a constant time for a given message size (m) and number of segments (n_s) regardless of number of processes (P). Figure 3 illustrates measured and predicted performance of segmented pipeline algorithm for reduce on communicators of size eight and twenty-four processes. The version of the method without

segmentation is accurately modeled using PLogP and Hockney models, and less successfully with LogP/LogGP model. The duration of the pipelined reduce algorithm with segments of size 1K are most accurately modeled using the PLogP model. Hockney model overestimates the measured value, meanwhile the LogP/LogGP underestimates them respectively.



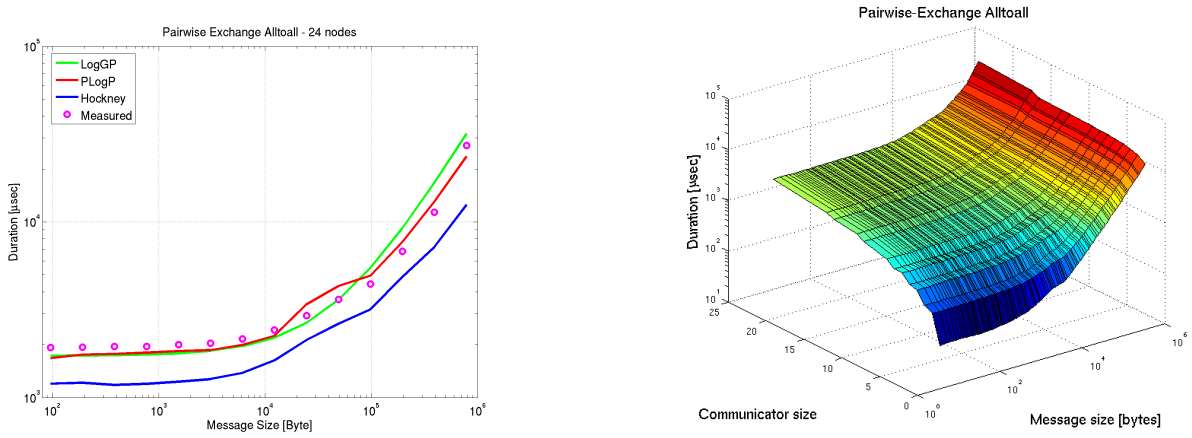
(a) (b)

Figure 3: **Performance of Segmented Pipelined Reduce algorithm**: (a) 8 nodes and (b) 24 nodes. Fitted parameter values were used to make predictions for LogP/LogGP and PLogP models (MPICH-2).

Figure 4 demonstrates the performance of pairwise-exchange alltoall algorithm. The alltoall class of collectives can cause network flooding even when we attempt to carefully schedule communication between the nodes. Hockney model does not have the notion of network congestion and this is one of the possible reasons why it significantly underestimates the performance of the algorithm. While we did not explicitly include congestion on PLogP and LogGP model formulas, they were able to predict the performance reasonably well.

Figure 5 summarizes the measured and predicted optimal implementation of the broadcast collective for 31 processes. Experimental data suggests that for small message sizes ($\leq 1Kb$), the optimal broadcast implementation would use either binary or binomial tree algorithms[5] without segmentation. For large messages ($\geq$ 1MB), segmented pipeline should be used. For intermediate-size messages, optimal algorithm to use is the binary tree algorithm with small segment size.

[5]The difference in performance of the binomial tree and the binary tree broadcast without segmentation for small message sizes was less than error of the measurement (5%)



(a) (b)

Figure 4: **Performance of Pairwise Exchange Alltoall algorithm**: (a) Measured performance and predictions on 24 nodes, and (b) Measured performance on 2 to 24 nodes. The message size represents the total send buffer size (FT-MPI).

Optimal Broadcast implementation according to the PLogP and LogGP models utilize the same methods but with slightly shifted switching points between the algorithms. Hockney model is not able to demonstrate benefit from using segmented pipeline over the binary tree algorithm, since it does not have the concept of gap per message. Due to system overheads, the time a node must wait between sending consecutive large messages over the network can be higher than the latency for that message size in the Hockney model.

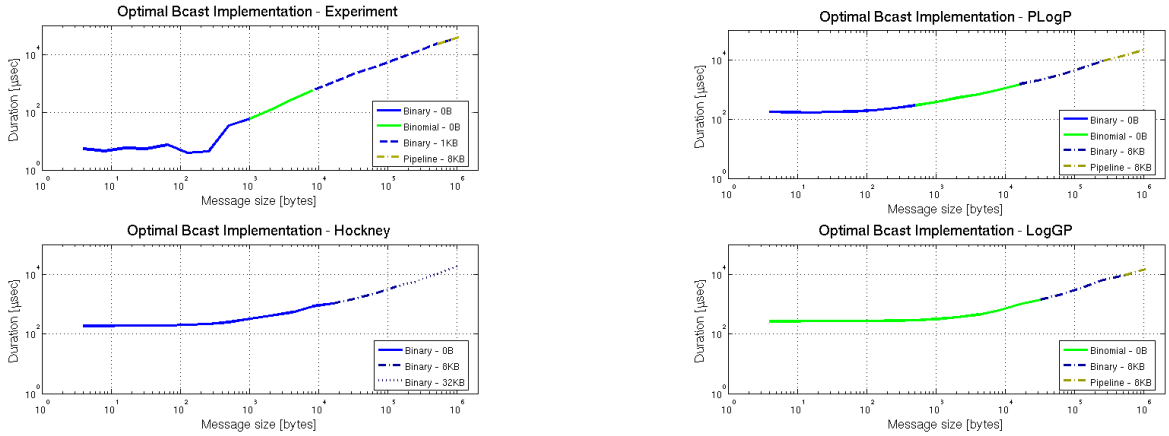


Figure 5: **Optimal implementation of Broadcast algorithm on 31 nodes** (MPICH-2).

7 Discussion and Future Work

We compared the Hockney, LogP, LogGP, and PLogP parallel communication models applied to MPI collective operations. We showed that even when we do not model network congestion directly, all of the models can provide us useful insights into various aspects of the different algorithms and their relative performance. However, Hockney model more frequently than other considered models, failed to correctly predict performance of algorithms. In the case of small and intermediate message sizes, the assumption that it must wait latency time before sending the next message is too pessimistic. For large messages, the assumption that it can start resending as soon as the first byte of the message has reached the recipient is too optimistic. Predictions from the PLogP and LogGP model were sufficiently close that one could use either of the models to reach similar conclusions. Nevertheless, the PLogP model has more flexible parameters, and our analysis suggests that the predictions were the closest to our experimental results.

This work was used to implement and optimize the collective operation subsystem of the FT-MPI library, but can be used as a library for any MPI implementation. In FT-MPI experimental and analytical analysis of collective algorithm performance was used to determine switching points between available methods. At run time, a particular method is thus selected based on the number of processes in the communicator, message size, and the rank of the root process.

We plan to extend this study in the following two directions: addition of new algorithms and collective operations to the OCC library, and making the algorithm selection process at run-time fully automated, rather than hard-coded at compile time.

Acknowledgments

The infrastructure used in this work was supported by the NSF CISE Research Infrastructure program, EIA-9972889. The authors would like to thank Zoran Dimitrijevic for suggestions and helpful comments.

References

[1] 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. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM Press, 1995.

[2] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 28.1. IEEE Computer Society, 2003.

[3] M. Bernaschi, G. Iannello, and M. Lauria. Efficient implementation of reduce-scatter in MPI. *J. Syst. Archit.*, 49(3):89–108, 2003.

[4] 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 Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.

[5] 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. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12. ACM Press, 1993.

[6] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. Assessing fast network interfaces. *IEEE Micro*, 16:35–43, 1996.

[7] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovsky, and J. J. Dongarra. Fault tolerant communication library and applications for high performance computing. In *LACSI Symposium*, 2003.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[9] R. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, March 1994.

[10] T. Kielmann, H. Bal, and K. Verstoep. Fast measurement of LogP parameters for message passing platforms. In J. D. P. Rolim, editor, *IPDPS Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 1176–1183, Cancun, Mexico, May 2000. Springer-Verlag.

[11] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140. ACM Press, 1999.

[12] R. Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proceedings of the Message Passing Interface Developer's and User's Conference*, pages 77–85, 1999.

[13] R. Rabenseifner and J. L. Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In *Proceedings of EuroPVM/MPI*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[14] R. Thakur and W. Gropp. Improving the performance of collective operations in MPICH. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number LNCS2840 in Lecture Notes in Computer Science, pages 257–267. Springer Verlag, 2003. 10th European PVM/MPI User's Group Meeting, Venice, Italy.

[15] S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 3. IEEE Computer Society, 2000.