Contention-Aware Performance Prediction of Parallel Programs

Pranjal Singh

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Project Report- CS396 (Undergraduate Project - II)
Supervisor - Prof. Preeti Malakar

Abstract. Message Passing Interface (MPI) is a standard for distributed-memory parallel programming and is widely adopted. Large-scale parallel programs with high compute requirements are also quite communication-intensive. One of the important research problems in this domain is the prediction of execution time of these parallel programs. The execution time comprises computation and communication times which is hard to predict due to variation in network parameters. Moreover, the performance of an application also depends on its neighborhood due to shared interconnects. This adds congestion in the network which is hard to quantify. We modify the Improved Profiler for MPI (IPMPI) to use historical and/or real-time network health to predict performance of collective communications in MPI. We extend the Hockney model for point-to-point messages into a piecewise function based on experimental data and use it to model collective communications in the presence of network congestion. We validate our model on a Beowulf cluster with irregular/unpredictable network load and also with artificial congestion. We obtain predictions with accuracy up to 80% for two collectives, with and without contention.

1 Introduction

1.1 Parallel Programs

Owing to the slowdown in improvements in microprocessor technologies, parallel computing has taken centrestage in recent years. Manufacturers have been increasing the logical core count consistently over the past two decades and parallel programs have become ubiquitous.

This requires programs to be rewritten to run in parallel on multiple cores - whether scientific simulations, transactions or network switching. The easier case is when a program is "parallelizable" - or can be broken into parts without significant dependencies. The harder case is that there are dependencies, and some form of communication is required between different execution entities (threads or processes). Usually, explicit hardware support is needed for communication - interprocess communication constructs within a machine, or a network, if a program is made to run on multiple machines.

A node (used interchangeably with machine/PC) has multiple cores, and all modern architectures and operating systems permit sharing of memory segments across processes, which may be running on different cores. Some series of processors may have over a hundred logical cores, and networks are fabricated alongside the cores themselves (Network-on-Chip architectures). This is because of the limited scalability of shared-memory systems - the overhead becomes unaffordable. However, these mechanisms are usually transparent to the programmer.

Programs/computation are distributed across multiple nodes if needed. Communication between nodes (inter-node communication) is usually via a LAN, and is significantly slower than intra-node communications.

1.2 MPI

MPI, or Message Passing Interface is a standard for distributed-memory parallel programming. It is widely used in parallel computing and is available on nearly all computation clusters and supercomputers. There are several implementations of the MPI standard (MPICH, MPAVICH, Intel MPI, etc) [6] [7]. Its popularity is largely due to its portability (it is a standard, not a library or a platform).

MPI implementations are responsible for launching processes, placement of nodes (if not specified) and setting up communication links between processes, taking into account the relative placement of each pair. These steps are transparent to the MPI programmer, making parallel programming convenient and platform-independent.

MPI consists of point-to-point messages and *collective* operations, which are operations involving many processes in a particular pattern. The collectives we test the model against are:

- Broadcast - one of the processes (the root) sends a message to all ranks (including itself) in the specified communicator.
- Allgather - each process in a communicator sends a message to all other processes. At the end, each process has the output/message from all processes.
- Alltoall - Each process a message each to all processes. At the end, the ith process has the ith message from each rank.

MPI also contains several message send modes (blocking, synchronous, buffered, nonblocking, ready, etc). We do not discuss these as the differences are not relevant to this work. Throughout, we use the standard send mode (`MPI_Send`) and blocking `MPI_Recv`.

1.3 Performance Prediction

Scheduling decisions in supercomputers/clusters make assumptions on the running time of the programs requested. However, predictions rarely achieve reliable accuracy. Communication may take varying amounts of time, depending on the network health and other random disturbances. Processes usually do not run synchronously (for sufficiently long programs) as a programmer might imagine. Thus, performance prediction is an important research direction. Approaches to performance modelling can be grouped into statistical and analytical modelling.

1.3.1 Network Contention

Parallel tasks usually run on systems alongside tens to hundreds of other jobs, on shared network switches and interconnects. The communication requirements of other jobs generate *external congestion* for network resources in the network. Network packets may have to wait in queues at switches for extended durations in such scenarios. In addition, *internal congestion* is generated by other processes of the job of interest - there may be hundreds of messages being sent in a single step in a collective. External congestion is considered a more complex entity, as it is transparent to the user. Unsurprisingly, researchers have failed to quantify contention - it does not behave as a mathematical function, in addition to being dependent on innumerable other entities. (We do not attempt to quantify it either, we only take it into account when modelling messages sent by our processes.)

External congestion is important in performance prediction, because (a) it is arguably the largest source of variation, and (b) over a reasonable-sized timeframe, it does not vary significantly, making it more predictable than other variations, if any. (The set of jobs in the cluster can be assumed to be fixed over short durations.) In our experiments, we observe slowdowns with artificial congestion.

1.4 Contributions

We attempt to develop a model for performance prediction of MPI programs under network contention using a lookup-based approach, trying to keep the lookup overhead minimal. We did not find models that test this approach, but there is a considerable body of research in directions similar to ours (in particular, network-aware collective algorithm selection).

In order to do this, we put the Hockney model [1] to the test and find some modifications that makes the model better predict communication times. We did not find research that explicitly does so, although there is literature on MPI send modes. We do not extensively test these modifications, as we instead aim to predict the execution time of an entire MPI program.

To this end, we develop two models to estimate the time needed for a single P2P communication, of which the first uses a direct lookup, and the second uses the modified Hockney model. We directly sum the time needed for each step in each collective of a program, and find the

output to be a somewhat accurate estimate of the running time of a program. We conclude with a discussion of the limitations of our work and some aspects that may be looked into in future research.

2 Related Work

There is little research on performance prediction of entire programs. However, a lot of research has gone into modelling P2P messages. Our second model incorporates the Hockney model described below. There are other models such as LogP, LogGP, etc that take into account other details and are more accurate, but we do not discuss them as we do not use them in this model. Most approaches to performance modelling use statistical methods and are not relevant to us. We discuss related research directions as well.

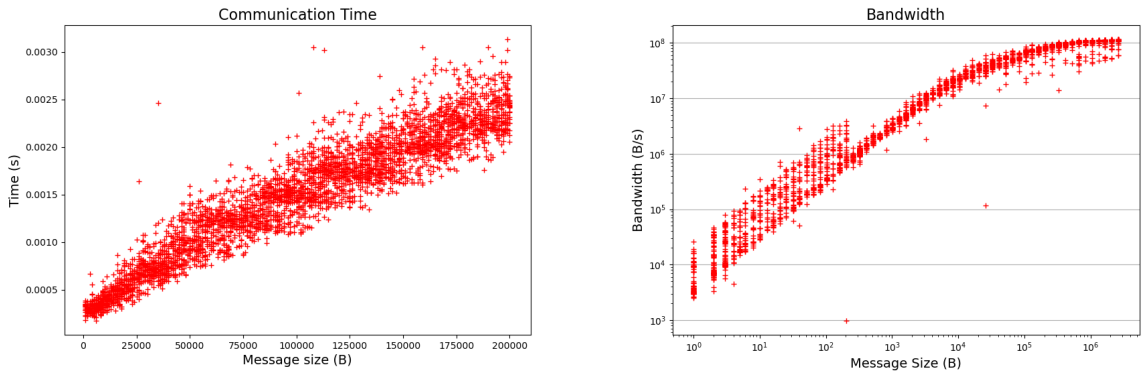
2.1 The Hockney Model

The Hockney model [1] was proposed in 1981 and is the most commonly used model for P2P messages. It divides communication time into latency and the actual send time, proportional to message size. Mathematically,

$$t(n) = \alpha + n \cdot \beta = \alpha + \frac{n}{B}$$

n - message size (Bytes)
α - latency
β - reciprocal bandwidth, seconds per byte
B - Bandwidth

The Hockney model is widely used to develop and analyse parallel algorithms and programs. Communication parameters are commonly estimated using the ping-pong test, where a message is sent to a node, and upon receiving the message, the node sends a message back to the sender. We experimentally verified whether the model is sufficiently accurate for our purposes.



(a) Linear plot

(b) Obtained bandwidth for message sizes

Figure 1: Verifying the Hockney Model

In 1, we note that the bandwidth converges to roughly 10^8 MBps. This is close to the link's capacity of 1 Gbps. For smaller messages, the bandwidth is significantly smaller. These details are taken into account in the next section.

A better way to verify whether the plots are really linear over the range (the range is too large to be plotted directly) is to subtract the theoretical bandwidth (or assume any constant bandwidth) from the observed communication time.

We obtain a nearly constant plot for latency on assuming a bandwidth of 1 Gbps, as seen in 2. We make a logarithmic plot to see how the model fares over the whole range of message sizes, and observe that it is nearly constant except in the 10 KB - 100 KB range. Overall, the latency increases only by a factor of about 10 for message sizes from 1 B to 1 MB. We conclude that the Hockney model is accurate enough for out purposes, but modify it based on our observations to increase the accuracy.

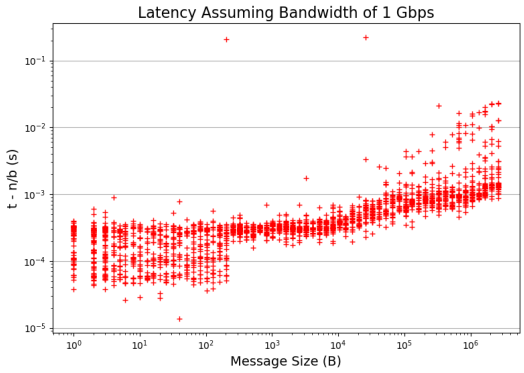


Figure 2: Latency for different message sizes

2.2 Data-based Collective Algorithm Choices

Some MPI collectives can be executed using multiple algorithms (discussed later in greater detail). Efforts have gone into choosing algorithms based on realtime network health rather than the hardcoded decision tree, and have succeeded to some extent. For instance, [3] use statistical (ML-based) methods and achieve speedups of up to 37% as compared to OpenMPI. They also note that IntelMPI is about as fast as the best algorithms (as determined by exhaustive search) which is noteworthy, as it indicates some knowledge of hardware is indispensable in making the optimal choices. Their results strengthen our opinion that timing actual messages is important. However, we do not use statistical approaches.

Nuriyev et al. [5] examine the high-level matheematical expressions in terms of communication parameters and propose two changes - (a) deriving models from the actual MPI source rather than an English-language description, taking into account the send modes for P2P messages (b) estimating Hockney parameters separately for each collective. They achieve speedups of up to 2 times as compared to OpenMPI, in addition to accurately predicting the time the best algorithm takes (best as determined by exhaustive search). A latent contribution of their work is validating the Hockney model, which has faced some criticism [4] in recent years. Their approach, like ours is to model collectives as a sequence of steps and to add the time needed for each step, although without lookup. They do not use statistical methods. They also do not take contention into consideration, which is a arguably our largest contribution.

We note that none of these works attempt to *predict* the running time of entire programs.

2.3 Extensions to the Hockney Model - LogP and LogGP models

WHile the Hockney model has found use in industry and academia, one does find it overly simplistic, or too good to be true. It is a valid question whether it is any more than a successful heuristic. In this section, we briefly discuss the LogP [8] and LogGP models for P2P messages which add another layer of detail to the Hockney model.

The LogP model has four parameters:
L: latency
o: software overhead on the CPU for a send/receive
g: gap - minimum interval between the injection of two messages into the network
P: the number of processors/memory modules

The authors separate the hardware and software overheads for message sending. As the notion of the "time spent in a message" is changed, the gap parameter is introduced to indicate how long the hardware is occupied with sending a message, and the overhead parameter does the

same for the software. The authors use these details to design better algorithms for collectives and thus establish the accuracy of their model. The LogGP model adds a separate parameter equal to the large-message bandwidth, as some architectures have separate support for long messages.

A consequence of this separation is message segmentation - if extraordinarily large messages, larger than the NIC's buffer are sent at a time, then the CPU idles as each packet is sent. All MPI implementations this break messages into appropriately sized segments so as to execute some computation in between.

3 Models

Programs and scripts used may be found at `http://github.com/pranjalsingh4550/mpi`.

3.1 Profiling Programs - Improved Profiler for MPI

MPI implementations use a variety of algorithms for each collective depending on the number of processes in the corresponding communicator and the message size. The latency dominates for small messages, and the bandwidth limits the execution time for large messages, and algorithms are chosen to minimise whichever is dominant.

The choice of algorithms is not known to the programmer directly. Further, the aggregate of all communication is also not known. This information can be used by process-mapping algorithms to generate node mappings that minimise the communication cost, given a particular node allocation. Profilers are used to obtain aggregate information on the actual algorithms used/details of point-to-point messages sent.

Improved Profiler for IPMPI, or IPMPI [2] is a profiler for IPMPI programs, that generates profiles of parallel programs, detailing communication patterns in a MPI program. We use a modified version of IPMPI that generates (a) communication matrices containing the minimum, maximum and sum of messages between two processes (b) a list of steps executed by each process, and the collective to which the step belongs. We modify and use (b) in this work, as described in the next subsection.

Broadly, our approach is to (a) estimate the time needed for each send, (b) find the maximum of these estimates over all the steps that run in parallel and (c) sum the estimates of the time needed for each step.

3.2 Performance Prediction using Historical Data

Experimental data shows significant variation in the performance of individual nodes in the CSEWS cluster. One of the possible reasons is the network topology, wherein two nodes may be connected to the same switch (2 hops) or different switches (4 hops). In practice, it is possible for nodes to be hetereogenous (we ran experiments on identical nodes for simplicity) or network components to be damaged.

To account for such variation, we gathered data on communication time between each pair of nodes in a group of 12 (CSEWS13 to CSEWS24). We timed 56 messages between each pair of nodes in both directions, for each message size from 512 B to 2 MB (in powers of 2). The data showed, for example, that some nodes were upto 10 times slower than the others due to issues with network links. Care was taken not to time any two messages at the same time. (The scripts used were not run simultaneously, even for messages between different sets of nodes.) Messages were sent in a random order (using the GNU shuf tool) rather than by using a simple loop, as it may be possible for the network to have external load for a certain duration within the window.

In our first model, we use the data generated previously as a lookup table, to estimate the time taken by a message. (Not all trial runs were on the same set of nodes, as at times less than

8 of these were up.) We tried incorporating this data into IPMPI to predict the time taken by a particular collective.

The first approach was to use gather estimates of the time taken in each step, (using `MPI_Reduce`) where each process would use lookup tables to construct an array containing the time required in each step. This also requires knowledge of the number of steps globally in the collective, as some nodes participate in some or none of the steps in a collective (using `MPI_Allreduce`). This introduces an unaffordable overhead in the profiler.

The previous approach was modified to add the data from lookup tables to the profiles generated directly - this can be done locally, and does not add significantly to the (already large) overhead of file I/O. A separate script runs through the profiles generated and finds the maximum in each step.

3.3 Accounting for Network Contention - Prediction using Realtime Network Health

The limitations of the above approach are

- Messages of all reasonable sizes cannot be timed. Some interpolation is necessary
- Gathering data between each pair of nodes may be prohibitively expensive. We had about 1,04,000 datapoints collected over a few days for as few as 12 nodes. Supercomputer jobs may run on thousands of processes, and it is impractical to time messages between each pair.
- Changes in network components or links render such data useless.
- Most importantly, historical data does not account for internal or external congestion in the network. External congestion is entirely a function of the other jobs on the system, and may vary on an hourly basis.

In the second model, we aim to reduce the overhead needed to gather historical data, such that it can be gathered at the time of running an application.

The first change we make is to reduce the number of parameters in the historical data - by letting go of the destination. While the historical data generated earlier showed that the outlier nodes were slow to recieve messages, as they were to send messages, the queueing of packets probably occurs in the network switches rather than in the sender for small messages.

The second feature of this model is that communication times are linearly interpolated using the communication times of only 5 message sizes - 8 KB, 16 KB, 64 KB, 128 KB and 1 MB (using the Hockeny model). These messages are timed by an program that sends a message between each pair of nodes. These messages are also sent in a random fashion, rather than by a loop.

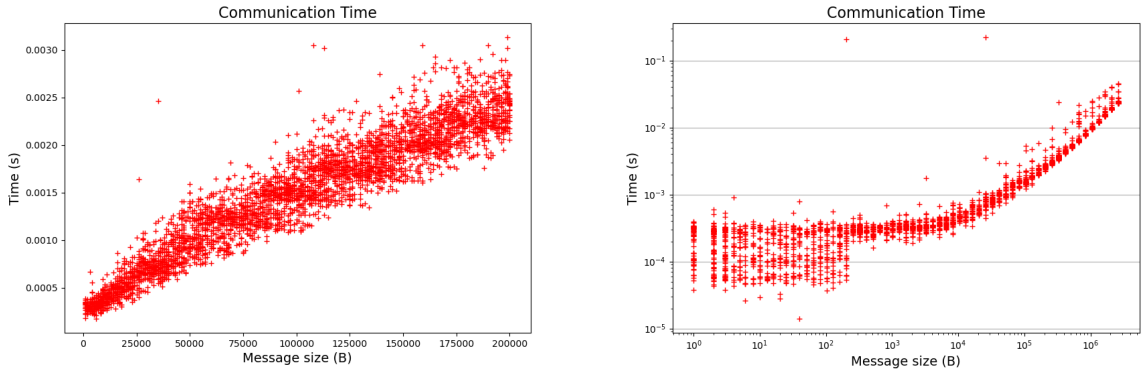
We time messages from each source (to 7 other destinations) and use the medians of the send time for 5 different message sizes (28 messages for each message size and 4 messages per destination per message size), and assume the destination does not make a difference. Thus, if there are a few slow nodes, then the medians obtained for them would be large, but for the remaining nodes, send/receive times from all other nodes are clubbed and the outliers do not significantly increase the obtained median. As before, messages are sent in a random order. Messages are also explicitly synchronised using `MPI_Barrier`.

3.3.1 Modifications to the Hockney Model

Linear interpolation is done using a piecewise linear function with three regions: 0 - 8 KB, 8 - 128 KB and $>$ 128 KB. These values coincide with the short-eager and eager-rendezvous

thresholds for P2P messages. We arrive on these values based on experimental observations, as shown in 3a and 3b.

Intranode messager were observed to be much faster, and the time taken does not depend on network health. We conclude that intranode messages are not of significance in this model. We use historical data to estimate communication times for simplicity.



(a) Linear Plot

(b) Logarithmic Plot

Figure 3: Communication time for P2P messages

Firstly, we observe in 3a that there is a change in the derivative between 60 KB and 120 KB. The eager-rendezvous (message sending modes) threshold is usually 128 KB in MPI implementations, so we let 128 KB be one of the thresholds in our model for P2P messages.

The first plot does not show the variation of communication times for small messages due to its limited granularity. On plotting a logarithmic plot (3b), we also note that there is close to no variation in communication time for messages shorter than 10 KB. This corresponds to short message mode. Hence, we further divide the domain at 8 KB and assume that short messages require a constant time. The time needed for collectives with messaage sizes of 4 KB and 8 KB (described in the next section) shows some variation, but we choose to trade accuracy for simplicity. The exact model is:

0 - 8 KB	Assumed to be constant - median time for 4 KB message
8 - 128 KB	Linear interpolation using medians for 16 and 64 KB messages
$\geq$ 128 KB	Linear interpolation using medians for 128 KB and 1 MB messages

Of the two approaches in the previous section, this model was made to run using only the second approach. The results are presented and discussed in the next section.

The model does not have the drawbacks of the previous model as described above. In our experiments, we observed that working estimates of communication time are obtained in less than 2 seconds, and with artificial contention, less than 4 s. Hence, the overhead of this model is sufficiently small to run the program at runtime and obtain communication parameters/an estimate of network health. The program we use to do this sends multiple messages in parallel (one between each pair) and thus accounts for internal congestion. One drawback that remains to be investigated is whether one message should be sent between each pair or a message in either direction. Collectives such as `MPI_Alltoall`, `MPI_Allgather` and some algorithms for `MPI_Bcast` send messages to/from each process. If there are unidirectional communication links, then dozens of packets may be sent for a 1 MB message, in which case a significant overhead may appear.

4 Results

We validated these models against Broadcast, Alltoall and Allgather on 8 nodes on the department's CSEWS cluster with 1 process per node (PPN) and 2 PPN, i.e. 8 and 16 processes. Owing to the small overhead of finding realtime communication parameters, we did the same under the same conditions of artificial network load and tested the second model under artificial network congestion also.

In the cases where artificial congestion is injected, different sets of nodes were used to generate the noise. In each plot, predictions and 5 - 10 observations for three different permutations of the list of hosts have been plotted. This is to account for the non-uniformity described earlier.

Wherever there are greater than 1 processes per node (PPN), we explicitly specify using command-line argument that adjacent ranks are to be placed on the same node (using the option `-host A:2,B:2,C:2,D:2`. By default, if the number of hosts (n) specified is less than the number of processes, the ith rank is placed on the $i\%n$th node. This does not leverage faster intranode messages, as heirarchies in trees/recursive doubling are such that adjacent ranks form subtrees in the algorithms used. Note, however that this should not have a bearing on the accuracy of predictions. Our models are agnostic to algorithms and rank placement.

The models are tested on the department's CSEWS cluster on nodes CSEWS1 to CSEWS32.

4.1 MPI Broadcast

MPICH uses three algorithms for broadcast depending on the message size and the number of processes (communicator size) - binomial tree, scatter followed by recursive doubling allgather or scatter followed by ring allgather (for ≥ 8 processes). Using MPI, we found the threshold for the first two to be 12 KB, and for the latter two to be 512 KB. This does not have a bearing on our scripts/modifications, since they make no assumptions about the algorithms used. The accuracy of predictions may differ, however.

Some optimisations are used for Broadcast on ≥ 8 processes within a node, which we observed when validating the first model. We do not investigate those.

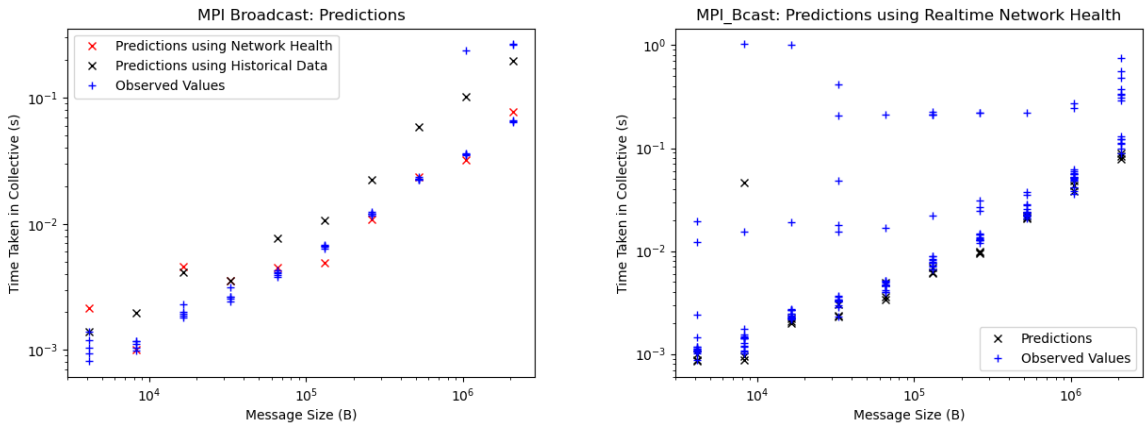


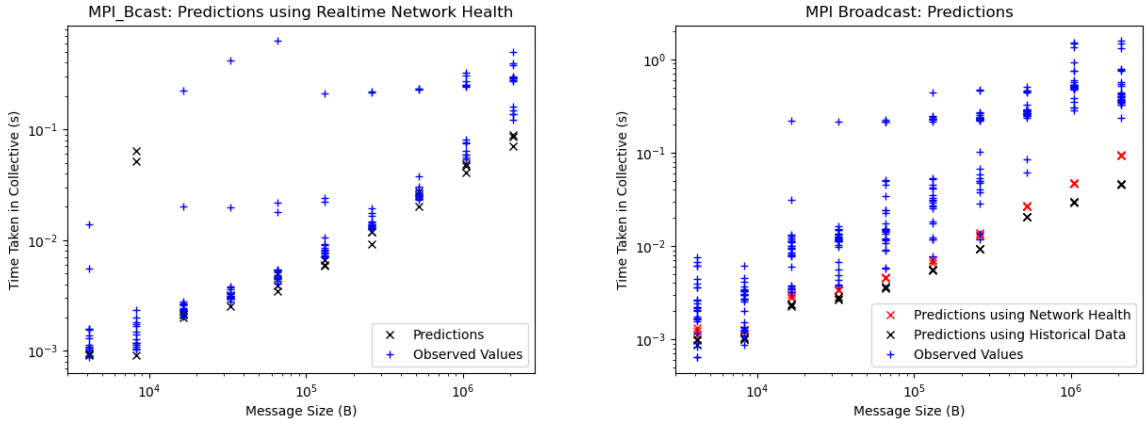
Figure 4: `MPI_Broadcast` on 8 nodes (1 PPN)

Figure 4 show predictions of our models for Broadcast on 8 processes without artificial contention. We draw the reader's attention to the variation in the observed values in the two plots, which is because these experiments were run on different days. Values in the first plot are lower than 300 ms, while some observations in the second plot are as high as 1 s.

In the first case, the first model (using historical data) overpredicts by 30% to 185% as compared to the median for each message size. This is exceptional and surprising, as we did not see overprediction in other cases. This may be because an outlier node was used, for which

a connection issue was later resolved. In general, we tried to avoid using such nodes. The second model's predictions vary from 74% to 205% of the actual values, with most under 130%. Overprediction in the second model may be either because we ignore the fact that slow nodes require longer to be sent a message to, from a regular node (refer to discussion of the second model), or because there was some disturbance in the network after the network health was measured. We believe these overpredictions are exceptions rather than failures of our models because overprediction was not observed in the other cases.

In the second (noisier) case, the second model's predictions are between 59% and 103% of the medians of the actual values (one outlier was omitted). The first model could not be run when obtaining the second plot, since less than 8 of the nodes for which we have historical data were live at the time. With broadcast, our models see limited success. Some possible improvements are discussed in the next section.



(a) 8 nodes, 1 PPN

(b) 8 nodes, 2 PPN

Figure 5: `MPI_Broadcast` with artificial contention

Figure 5 shows the predictions of the models under artificial contention. With 8 processes, most predictions of the second model are between 0.6 and 1.03 times the median, but in some cases there is underprediction by 70-80%, for large message sizes in particular. Note that the predictions are still reasonably close to the minimum oberved value for the large and small/medium messages. Another observation we make in this plot is that the outliers form clear trends. It is possible for observed data to be less noisy for particular node allocations. (Recall that we assigned ranks randomly to nodes.) It is possible that our model does better in such cases.

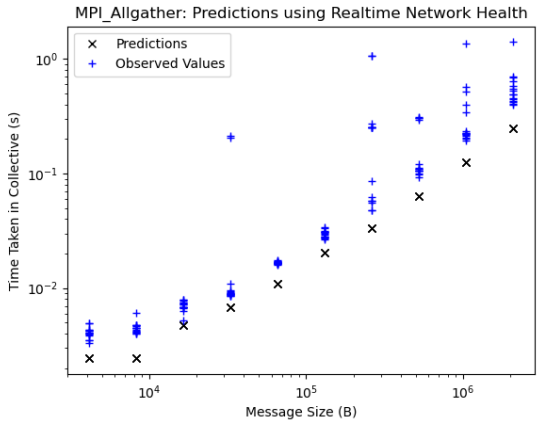
In the second case (16 processes), the first model predicts 10% to 45% of the actual values, because of the contention. The second model has marginally higher predictions, ranging from 10% to 60%. There is a jump in observed values at 512 KB, which is probably because a different algorithm is used. This is part of a larger trend we observe later - underpredictions rise when the number of processes increases, under artificial contention.

Predictions using realtime network health are generally more accurate when the observed communication times have less noise. This could be because the number of outliers in the 8 or so simultaneous messages may be large. This is discussed in detail in the next section.

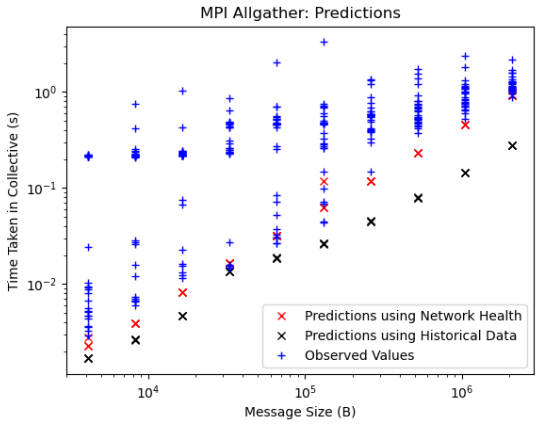
The second model fares better against contention, although the predictions are not accurate. But this is evidence that we have succeeded in making the model sensitive to contention.

4.2 Allgather

There are two algorithms for Allgather: recursive doubling and ring allgather. The threshold for ring allgather is $\geq$ 64 KB.



(a) On 8 nodes (1 PPN) without contention



(b) On 8 nodes (2 PPN) with artificial contention

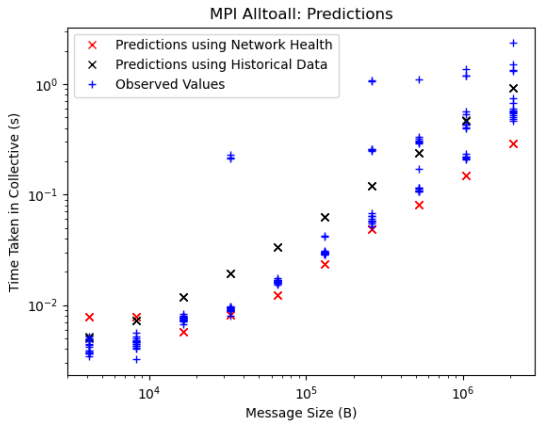
Figure 6: `MPI_Allgather`

We do not plot all 4 combinations as there is not much variation. The observations in the previous subsection hold here too. With 8 processes, the second model predicts 45% to 75% of the observed values. In the second case (16 processes with contention), the first model fails and predicts 2% to 25% of the median of the observed values. We reiterate that the first model does not account for contention in any manner. The second model fares better but is still not accurate - its predictions vary from 3% to 85%. Note the overlap with the accuracy for the former case, without contention. This can be interpreted as the second model predicting with accuracy similar to the first but more noise.

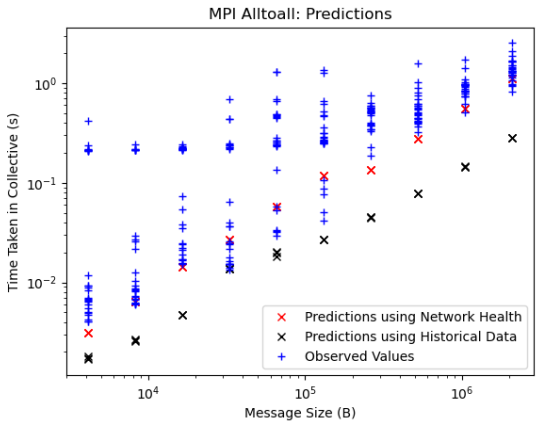
Like in the case of broadcast, in the algorithms for allgather, each rank sends a message in each time step, rather than a message being sent between each group of two. Changing the tokenization methodology in the second model may increase the accuracy.

4.3 All-to-all

Only one algorithm (the trivial one) is used for all to all. Once again, predictions are more ac-



(a) On 8 nodes (1 PPN) without contention



(b) On 8 nodes (2 PPN) with artificial contention

Figure 7: `MPI_Alltoall`

curate for collectives on 8 processes. Unlike the other collectives, the second model overpredicts communication times for small messages. With 8 processes and without artificial contention,

the first model overpredicted by 10% to 100%, and like in the case of broadcase, we suspect it may be because one of the former outliers was used. The second model predicts 0.4 to 1.7 times the median of the actual values. The general trend, however, is to underpredict, for messages larger than 10 KB in particular. This is consistent with what we observe in the other plots.

In the second case, on 16 processes with artificial contention, the first model (historical data) fails and predicts 0.08 to 0.3 times the median. The second model predicts 0.3 to 0.8 times the median of the observed values. The predictions of the second model are usually not lower than the lowest observed value.

The accuracy obtained (80-85%) is similar for allgather and all to all. In conjunction with the fact that in both collectives, each process sends a message in each step, this indicates there is some *regularity* in our predictions. That there is regular underperdiction, is a fact we have briefly discussed earlier.

The plots are discussed in the following section.

5 Discussion

Firstly, we highlight that in most cases, both of our models succeed in predicting the trends observed.

Our models perform better when there is less noise in the network. This is expected of the first model, since noise only serves to increase the actual communication time. This is noteworthy in the case of the second model, since we make explicit efforts to account for noise. One possible explanation is that the bell curve is "flatter" and there is greater deviation from the median. As it is the maximum of up to 8 random variables that matters, there is greater probability that at least one takes a large value. This is discussed in the next subsection.

We also note that the models perform better when there are 8 processes. When 16 processes were used, care was taken to assign adjacent ranks to the same node. The amount of internode communication does not increase significantly, but we still observe increased noise in the actual time needed. This is surprising, but we are unable to explain it. It is of significance, as it may be either the noise that causes mispredictions or an increase in the number of messages sent in a time step.

Experiments showed that upon running a collective on loop without synchronisation, the time taken per collective decreases. On the other hand, if the network health is tested without explicit synchronisation, then the obtained values increase significantly. We are unable to explain this.

5.1 Future Work

One of the assumptions in this work is that medians are representative of the time needed for a particular step. However, on treating the execution time as a random variable, we note that the maximum of a set of identical random variables, (wrongly but for illustration) assuming a normal distribution, is likely to have an expectation larger than the median: the as little as one of the variables needs to take a larger-than-usual value for the maximum to increase. As the number of variables increases, the distribution representing their maximum increases (in theory and practice alike). We plan to model send time as a *probability distribution* in the future, as this would lead to better estimates of the time spent in each step. In fact, this may be one of the reasons both models underpredict when the number of processes increases.

One of the problems that arises is that the size of the sample space (assuming a discrete random variable) grows dramatically - the sum of n random variables, each of which takes t possible values is a random variable with a range of size t^n. This approach is feasible only with some approximation/aggregation at each step.

We did not examine whether mispredictions are correlated with having a large number of processes as well, along with artificial contention. One may run collectives on varying numbers of nodes/processes to this end.

At several points in the previous section, we have made references to the tokenization method used in the second model. In algorithms for many collectives, messages are sent in both directions in a pair of ranks, and in some other cases, a rank sends and receives messages simultaneously, but the source and destination are different (such as in ring allgather). The program we use to estimate network health sends *one message per pair*, and one should expect some underprediction if communication parameters are found this way. Unfortunately, we did not connect these dots sufficiently early so as to incorporate this change. But we expect this will increase the accuracy of our models, and may push the predictions in the 40-85% ranges closer to 100.

6 Conclusions and Future Work

Predicting the performance of parallel programs is an important research direction, arguably more important than predicting sequential program performance (as parallel programs run on shared resource). We attempt to model the time required by collective operations in parallel programs.

We build upon the Hockney model and generalise it to a piecewise linear function, in order to increase the accuracy. Note that the Hockney model is over three decades old, and computing and networking hardware have undergone a sea change since then, and there is research showing that the model is inaccurate. However, our extensions succeed in modelling some collectives, albeit under favourable conditions. The other assumption we make is that MPI collectives can be modelled as a sequence of messages sent in the standard mode. Our experiments show there is merit in this assumption.

We attempt to use tokenization to account for network contention, and partially succeed in accounting for contention with a reasonable overhead. In summary, we find some merit in modelling parallel programs as a sequence of messages in the standard blocking mode.

References

[1] Hockney, R.W., & Jesshope, C.R. (1988). Parallel Computers 2: Architecture, Programming and Algorithms (1st ed.). CRC Press. https://doi.org/10.1201/9780367810672

[2] T. Agrawal and P. Malakar, "IPMPI: Improved MPI Communication Logger," 2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI), Dallas, TX, USA, 2022, pp. 31-40, doi: 10.1109/ExaMPI56604.2022.00009.

[3] S. Hunold, A. Bhatele, G. Bosilca and P. Knees, "Predicting MPI Collective Communication Performance Using Machine Learning," 2020 IEEE International Conference on Cluster Computing (CLUSTER), Kobe, Japan, 2020, pp. 259-269, doi: 10.1109/CLUSTER49012.2020.00036.

[4] William Gropp, Luke N. Olson, and Philipp Samfass. 2016. Modeling MPI Communication Performance on SMP Nodes: Is it Time to Retire the Ping Pong Test. In Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI '16). Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/2966884.2966919

[5] E. Nuriyev and A. Lastovetsky, "Efficient and Accurate Selection of Optimal Collective Communication Algorithms Using Analytical Performance Modeling," in IEEE Access, vol. 9, pp. 109355-109373, 2021, doi: 10.1109/ACCESS.2021.3101689.

[6] MPICH—A Portable Implementation of MPI. Accessed: Nov. 19, 2023. [Online]. Available: http://www.mpich.org/

[7] Open MPI: Open Source High Performance Computing. Accessed: Nov. 19, 2023. [Online]. Available: https://www.open-mpi.org/

[8] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: towards a realistic model of parallel computation. SIGPLAN Not. 28, 7 (July 1993), 1–12. https://doi.org/10.1145/173284.155333

[9] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, Chris Scheiman, LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation, Journal of Parallel and Distributed Computing, Volume 44, Issue 1, 1997, Pages 71-79, https://doi.org/10.1006/jpdc.1997.1346.