Auto-tuning MPI Collective Operations on Large-Scale Parallel Systems

Wenxu Zheng*, Jianbin Fang*✉, Juan Chen*, Feihao Wu*, Xiaodong Pan*, Hao Wang*,
Xiaole Sun*, Yuan Yuan*, Min Xie*, Chun Huang*, Tao Tang*, Zheng Wang†‡
*College of Computer, National University of Defense Technology, China
†School of Computing and Communications, Lancaster University, United Kingdom
‡Xi'an University of Posts & Telecommunications, China
Email: {zhengwenxu17, j.fang, juanchen, wufeihao16, panxiaodong17, wanghao18, sunxiaole18, yuanyuan,
chunhuang, xiemin, taotang84}@nudt.edu.cn, z.wang@lancaster.ac.uk

***Abstract*—MPI libraries are widely used in applications of high performance computing. Yet, effective tuning of MPI colletives on large parallel systems is an outstanding challenge. This process often follows a trial-and-error approach and requires expert insights into the subtle interactions between software and the underlying hardware. This paper presents an empirical approach to choose and switch MPI communication algorithms at runtime to optimize the application performance. We achieve this by first modeling offline, through microbenchmarks, to find how the runtime parameters with different message sizes affect the choice of MPI communication algorithms. We then apply the knowledge to automatically optimize *new unseen* MPI programs. We evaluate our approach by applying it to NPB and HPCC benchmarks on a 384-node computer cluster of the Tianhe-2 supercomputer. Experimental results show that our approach achieves, on average, 22.7% (up to 40.7%) improvement over the default setting.**

***Keywords*-MPI, collective communication; auto-tuning**

I. Introduction

Message Passing Interface (MPI) is a de facto standard for programming large-scale high-performance computing (HPC) systems [2]. It provides an extensive set of tuning parameters to allow programmers to customize the MPI environment to match the application requirements and the underlying hardware. While flexible, choosing the optimal MPI parameter settings is challenging as the number of the possible options is huge and improper parameters can have a significantly negative impact on the resulting performance.

Prior studies have shown that the best communication algorithm for a collective operation highly depends on the message to be transferred [30, 36, 38]. As there is no "one-size-fits-all" algorithm, the mainstream MPI implementations, e.g., MPICH [3], provide a set of tuneable parameters for configuring how collective operations can be performed. Because of the large number of runtime parameter configurations, as well as the differences of application features (i.e. number of processes, size of problem and count of collective functions) and underlying hardware (i.e. interconnection, communication bandwidth and node infrastructure) in various system platform, manually selecting the "optimal" application-specific configuration involves an extremely large search space and thus is a challenging job.

A naïve method to obtain the optimal configuration is to enumerate all the possible configurations and choose the optimal setting of runtime parameters. However, the large search space means an exhaustive search is often prohibitively expensive in practice. As a compromise, researchers manually construct heuristics to find a sub-optimal solution in a quick manner. However, building a good heuristic requires intensive expert involvement. This makes it difficult to keep a timely heuristic updated when the underlying hardware or MPI implementation has changed.

This paper aims to find a way to automatically construct MPI optimization heuristics. We do so by first empirically characterizing the impact of communication algorithms on the performance of MPI collective operations using microbenchmarks, through tuning the runtime parameters. Based on the knowledge extracted from the microbenchmarks, we then develop a model to automatically choose, from a set of available algorithms available to an operation, which communication algorithm to use at runtime. Our key insight is that most real-world applications contain various MPI collective operations [8], and each of their best-performing communication algorithms is largely depending on the message size. If we can find, from microbenchmarks, the correlation between the message size and the optimal operation-specific algorithm, we can then apply and transfer this knowledge to optimize *new*, *unseen* MPI programs.

In this paper, we consider eight commonly used MPI collective operations, or primitives (see Table I), provided by MPICH, a widely used open source MPI implementation, and a total of 19 communication algorithms. We use the OSU MPI benchmark (OMB) suite [4] to characterize MPI program behaviours. This microbenchmark suite contains all the collective operations. To understand how the operation-specific communication algorithms affect the performance, we use a profiling tool, Integrated Performance Monitoring (IPM) [1], to capture the size of communicated messages during program execution time. We then build a mapping from the message size to the best-performing communication algorithm. The extracted knowledge can then be applied to *unseen* real-world applications. Such an approach requires little human involvement as the microbechmark profiling and

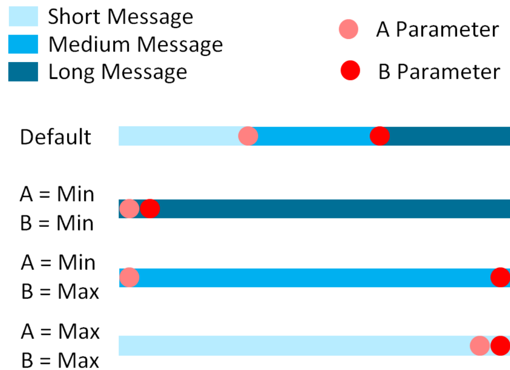


Figure 1. Choosing communication algorithms by changing runtime parameters. In this example, by setting the runtime parameters, A and B, we can change the communication algorithm for an MPI collective operation.

characterization can be automatically performed offline. This means whenever the hardware or MPI library implementation has changed, we can simply re-run this process to update the heuristic. This approach is a new way for building optimal heuristics for tuning MPI collective operation.

We evaluate our approach by applying it to the NPB and HPCC parallel benchmark suites on a 384-node cluster of the Tianhe-2 supercomputer. Experimental results show that our approach achieves, on average, 22.7% (up to 40.7%) improvement over the default setting.

This paper makes the following contributions:

- We empirically quantify how the message size and the choice of communication algorithms affect the performance of commonly used MPI collective operations on a sub-system of the Tianhe-2 supercomputer.
- We propose a novel approach to automatically construct heuristics to choose the optimal communication algorithm for a given collective, taking into consideration of the process number and the problem size. And the approach gives significantly performance improvement (up to 40.7%) on real-world MPI applications, without modification to the program source code.

The rest of this paper is organized as follows. Section 2 introduces the background. Section 3 illustrates the experimental setup, which contains the benchmark and platform instruction. Section 4 shows the performance analysis of our micro-benchmark results with the best configuration. Section 5 evaluates how we apply our model onto several typical real-world applications. Section 6 shows related work and Section 7 concludes the work.

II. Background

A. Problem Scope

The MPI standard defines a set of collective communication operations, each come with a range of algorithm choices. In this work, we target the MPICH, an open source implementation of the MPI standard. We consider eight collective operations offered by MPICH, namely *alltoall*, *allgather*, *allreduce*, *bcast*, *reduce*, *gather*, *reduce_scatter* and *scatter*.

The communication algorithm used for a collective operation is determined by a set of operation-specific runtime parameters, defined in Table I. Figure 1 gives a concrete example on how the change of runtime parameter values affect the choice of communication algorithms. In this example, we assume two runtime parameters, *A* and *B*, which are used to determine whether a specific message is short, medium or long, to switch the corresponding communication algorithm.

In this work, we are interested in choosing the right parameter settings to help the MPICH runtime to choose the optimal algorithm for a given collective operation. We do so by characterizing how the communication message size and the number of parallel processes affect the choice of parameter settings. Note that the optimal parameter settings also depend on the underlying hardware; hence we wish to find a way to straightforwardly model the relation between the application features (i.e., message size and process number) and MPI communication algorithms.

B. MPI Collective Operations

We now describe each of the eight collectives aforementioned.

MPI_Allgather aims to gather data from all tasks and distribute the combined data to all task. It differs from *MPI_Gather* in that *allgather* distributes the data onto all the involved processes. The conventional implementation of *allgather* uses a ring method so that the data from each process is sent around a virtual process ring. Due to the high latency of this approach, researchers have developed two new algorithms for short message, i.e., Recursive doubling and Bruck [36]. Typically, the former works very well for power-of-two cases, while Bruck works for other process configurations. In the following context, we set the power-of-two numbers of process, and thus we only use the *Recursive doubling* approach for short messages.

MPI_Alltoall defines that each process sends/receives data to/from every other process. The conventional *alltoall* uses MPI_Isends and MPI_Irecvs to send and receive data without scheduling communication in each process. As an optimization, the Bruck algorithm is applied to the less-than-8KB messages with at least eight participating processes. For long messages, the Pairwise Exchange algorithm is used for power-of-two cases.

MPI_Reduce performs a global reduction operation reducing values from all processes to the root process. The *Binomial Tree* algorithm does well for short messages, while for long message, a better algorithm is proposed by Rabenseifner [31], which implements a long-message reduce with recursive-halving algorithm followed by a gather with binomial tree algorithm to the root. In addition,

MPI_Allreduce can be implemented in a straightforward manner by performing a reduce operation followed by a broadcast. However, for short messages, it uses a recursive doubling algorithm, similar to *MPI_Allgather*. And in the Rabenseifner's algorithm, the *gather* becomes an *allgather* with recursive doubling algorithm for long message.

MPI_Reduce_scatter reduces and scatters the results among all processes, instead of storing the data only on the root process as *MPI_Reduce* does. Note that the *scatter* here is a variant (i.e. scatterv), that the number of data sent to each process is uneven. The conventional algorithm for *reduce_scatter* is to use a binomial tree. To avoid the large overhead for lone messages, the Pairwise exchange algorithm has been introduced. Note that, this communication algorithm differs depending on whether the reduction operation is commutative (e.g., MPI_SUM, MPI_MAX) or not.

MPI_Bcast is a simple case of the one-to-all collectives, which is implemented based on three communication algorithms selected according to the message size. For short message, binomial tree algorithm is widely employed. For long message, the function executes a scatter followed by an allgather.

C. Runtime Parameters

Like other MPI implementations, MPICH offers two set of runtime parameters to customize the software environment.

Platform-specific parameters are related to the underlying architecture of a target platform, which are specifically designed for functionality rather than performance. They are used to enable or disable certain features, such as the error checking, thread level and network module, including the switching of on/off several MPI functions.

Function-specific parameters are used to select the communication algorithms applied to a specific MPI function. These algorithms can be selected according to the message size and the function-specific runtime parameters. In this work, we focus on tuning these runtime parameters to match applications and parallel systems.

Recall that our goal is to determine the optimal runtime parameter values to help the system choose the appropriate communication algorithm. We do so by modeling the correlation between the application features and the parameter settings in a empirical manner. In the following section, we will introduce our approach for obtaining the **best configuration** of runtime parameter settings according to the message size and process number.

III. Experiment Setup

We now describe the experiment setup, including platforms, MPI runtime parameters, benchmarks and profiling methodology.

Table I
Runtime parameters and their default values

	Runtime Parameters	Defaults
`Alltoall`	MPICH_ALLTOALL_SHORT_MSG_SIZE	256
	MPICH_ALLTOALL_MEDIUM_MSG_SIZE	32K
`Allgather`	MPICH_ALLGATHER_SHORT_MSG_SIZE	80K
	MPICH_ALLGATHER_LONG_MSG_SIZE	512K
`Allreduce`	MPICH_ALLREDUCE_SHORT_MSG_SIZE	2K
`Bcast`	MPICH_BCAST_SHORT_MSG_SIZE	12K
	MPICH_BCAST_LONG_MSG_SIZE	512K
`Reduce`	MPICH_REDUCE_SHORT_MSG_SIZE	2K
`Reduce scatter`	MPICH_REDSCAT_COMMUTATIVE_LONG_MSG_SIZE	512K
`Gather`	MPICH_GATHER_INTER_SHORT_MSG_SIZE	2K
`Scatter`	MPICH_SCATTER_INTER_SHORT_MSG_SIZE	2K

A. System Hardware and Software

We use a sub-system of the Tianhe-2 supercomputer, which comprises 384 compute nodes with customized interconnection [23, 27]. Each compute node runs a 64-bit GNU/Linux 2.6.32 kernel, with GCC v4.4.7 of the "-O3" compiler option [46] and MPICH v3.1 library, and is equiped with 64GB memory, two Intel Xeon E5-2692 12-core processors running at 2.20 GHz, with 32 KB L1 cache, 256 KB L2 cache and 30 MB L3 cache.

B. Configurable Runtime Parameters

MPICH v3.1 has around 70 runtime parameters, twelve of which are function-specific parameters. Table I gives the collective operations and their configurable parameters targeted in this work. We note that `MPIR_CVAR_BCAST_MIN_PROCS` has little impact on the communication performance, and thus we do not consider this runtime parameter in the remaining context.

As depicted in Figure 1, changing the runtime parameter values can direct the MPI to choose the communication algorithm to use at runtime. For example, *MPI_Alltoall* has two runtime parameters, `MPIR_CVAR_ALLTOALL_SHORT_MSG_SIZE` and `MPIR_CVAR_ALLTOALL_MEDIUM_MSG_SIZE`, determining what communication algorithms to used according to the size of transfering message. For the default cases, when the message size is shorter than 256B, this collective tends to use the short message algorithm, i.e., the Bruck algorithm. When the message size is larger than 32KB with a power-of-two number of processes, it will use the long message algorithm, i.e., the Pairwise algorithm. When the message size falls in between the two parameter values, it will use the medium message algorithm, i.e., the original irecvs/isends algorithm. Our approach implicitly switches the communication algorithm by changing the parameter settings at runtime.

C. Benchmarks

Two sets of benchmarks are applied in our experiment. We use the `OSU MPI Benchmark (OMB)` [4], a microbenchmark suite, to characterize the correlation of the

Table II
BENCHMARKS AND THEIR MPI COMMUNICATION OPERATIONS

Benchmarks	MPI Collective Operations
NPB.IS	MPI_Alltoallv, MPI_Alltoall, MPI_Allreduce, MPI_Reduce, MPI_Bcast
NPB.FT	MPI_Alltoall, MPI_Barrier
NPB.CG	MPI_Reduce, MPI_Bcast, MPI_Barrier
NPB.SP	MPI_Allreduce, MPI_Bcast, MPI_Barrier
NPB.BT	MPI_Allreduce, MPI_Reduce, MPI_Bcast, MPI_Barrier
NPB.MG	
NPB.EP	
HPCC.RandomAccess	MPI_Alltoall, MPI_Allreduce, MPI_Reduce, MPI_Bcast
HPCC.FFT	
HPCC.PTRANS	MPI_Allreduce, MPI_Reduce, MPI_Bcast
HPCC.DGEMM	
HPCC.STREAM	
HPCC.Latency/Bandwidth	
HPCC.HPL	MPI_Allreduce, MPI_Bcast

Table III
MESSAGES SIZES OF MPI_ALLTOALL IN DIFFERENT APPLICATION WITH DIFFERENT CONDITIONS(IN BYTES)

App.	no. of proc.	Msg-size(B) in Diff. Class/Prob-size				
		A	B	C	D	E
FT	16	32K	128K	512K	-	-
	32	16K	64K	256K	1M	-
	64	8K	32K	128K	512K	-
	128	4K	16K	64K	256K	1M
	256	2K	8K	32K	128K	512K
	512	1K	4K	16K	64K	256K
	1024	512	2K	8K	32K	128K
IS	16	4	4	4	4	4
	1024	4	4	4	4	4
		2000	2500	3000	4500	6000
MPI-FFT	16	64K	128K	256K	512K	1M
	32	32K	64K	128K	256K	512K
	64	16K	32K	64K	128K	256K
	128	8K	16K	32K	64K	128K
	256	4K	8K	16K	32K	64K
	512	2K	4K	8K	16K	32K
	1024	1K	2K	4K	8K	16K
MPI-R.A.	16	8K	8K	8K	8K	8K
	1024	8K	8K	8K	8K	8K

message size, the number of processes and the runtime parameter settings for the eight operations. We then test our model on two other benchmark suites, `NPB` and `HPCC`, which are two collections of real-world HPC applications.

Table II lists the MPI collectives used in each NPB and HPCC benchmark. In NPB, the applications are written in Fortran except that 'IS' which is written in C. We note that only `FT`, `IS`, `MPIRandomAaccess` and `MPIFFT` contain the *MPI_Alltoall*. The average of five runs each benchmark are reported in the following experiment.

D. Profiling Setup

We profile each of the `OMB` suite with a message size ranging from 1 B to 1 MB. To characterize the performance of MPI collectives, we set the threshold value of MPI runtime parameter to be within 1 B (the minimum) and 1 MB (the maximum). In NPB and HPCC, each benchmark contains various MPI collectives, and the ratio of the collective running time to the total execution time varies [8], which means that the effection of collectives to the general performance is different.

To capture the applications' features, we use the IPM [1] profiling tool to obtain the message sizes at runtime. IPM is a lightweight parallel program analysis instrument, which provides a low-overhead performance profile of the utilization of resource and the performance of a parallel program, such as the count and latency of MPI collective. And it has very slight effect on the code execution. At the same time, by analyzing the application source codes, we can obtain the message sizes of the program. The size of the communication message for different MPI collectives in different programs vaies depending on the problem size and the process number. Table III gives the communication message sizes of the four applications (i.e., FT, IS, MPI-RandomAccess and MPIFFT) for the *MPI_Alltoall*. We find that the message sizes of `FT` and `MPIFFT` change with the problem size and the process scale, while the message sizes of `IS` and `MPIRandomAccess` look constant.

In addition, *MPI_Allreduce*, *MPI_Gather*, *MPI_Bcast*, *MPI_Reduce* and other collectives are executed in the applications, which show the similar feature to *MPI_Alltoall*. In the following section, we evaluate the best configurations obtained from the microbenchmark, focusing on the *MPI_Alltoall*, which appears in `FT`, `MPIFFT`, `MPIRandomAccess`. We do not consider `IS`, because the message size is too small to impact general performance.

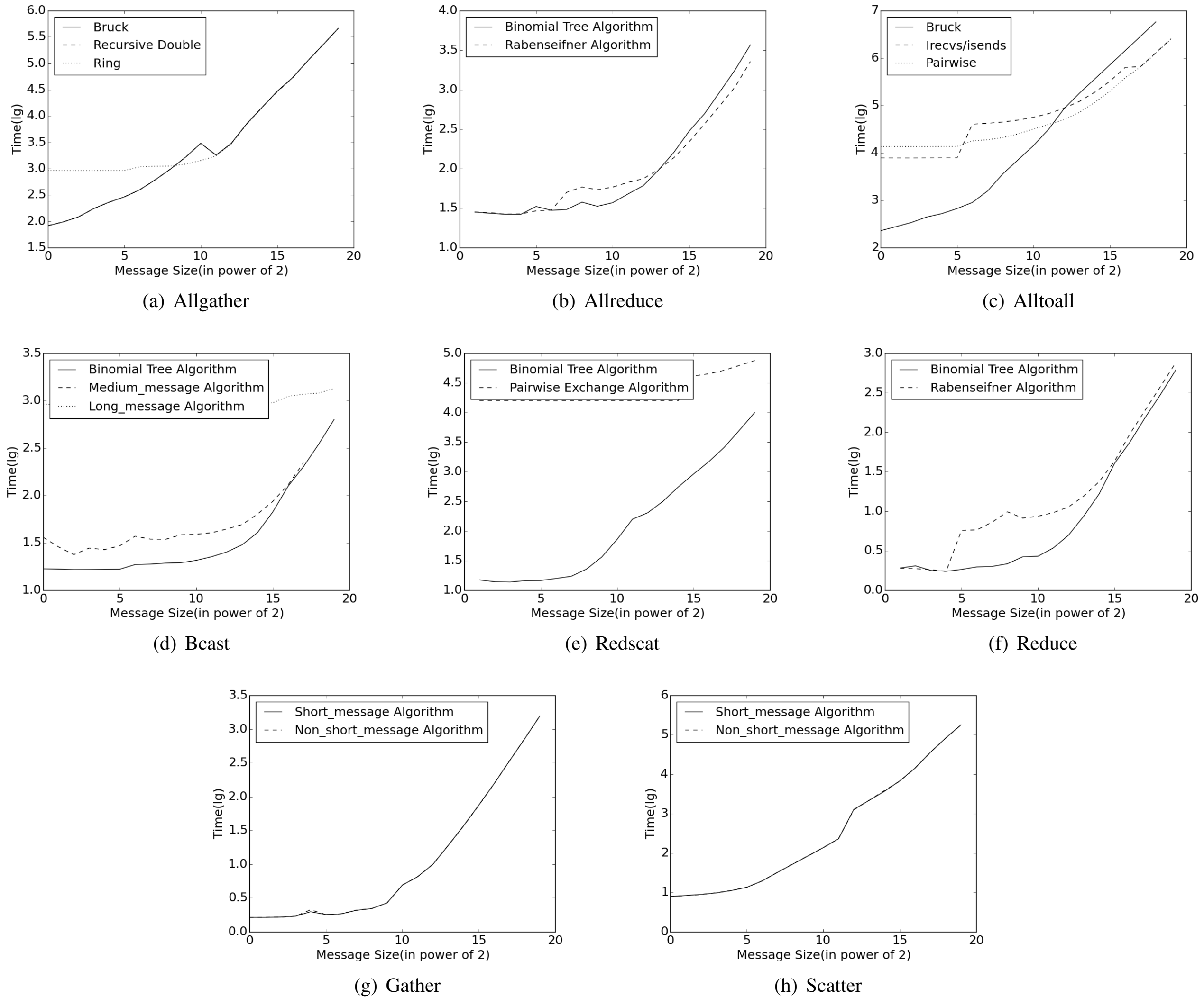
IV. PERFORMANCE MODELING

In this work, we are particularly interested at understanding how the message size and the number of parallel processes affect the choice of communication algorithms. We obtain such knowledge through profiling the `OMB` microbenchmarks.

A. Impact of Message Size

Our first task for performance modeling is to understand how each available communication algorithm perform when the message size and the process number vary. Given that we have a large range of parameter values, exhaustively trying all possible parameter value and process number combinations would be prohibitively expensive. Instead, we sample the parameter space by considering 16 to 1024 processes and 1B to 1MB for message sizes at a step of power-of-2. We then use the sample data to fit the performance curve to compare how different communication algorithms perform under different message-size and process-number settings.

Specifically we run each of the eight collective operations with the number of processes in the range of {16, 32, 64, 128, 256, 512, 1024}. For MPI operations that have just one configurable runtime parameter, we set the parameter value in a similar manner. As a example, Figure 2 shows the performance curves for each of the collective when the number of parallel processes is 1024.



(a) Allgather (b) Allreduce (c) Alltoall

(d) Bcast (e) Redscat (f) Reduce

(g) Gather (h) Scatter

Figure 2. The achieved performance of available communication algorithms with different message sizes using 1024 processes. The x-axis shows the message size (varies from 1 B to 1MB), and the y-axis shows the measured execution time in the logarithm form.

1) MPI_Allgather: As shown in Figure 2(a), the switching point between Recursive Double and the Ring algorithm is 337B (the message size). We do not show the performance of Bruck, which cannot work with a power-of-2 processes in *MPI_Allgather*. `MPICH_ALLGATHER_LONG_MSG_SIZE` decides which communication algorithm to use. By setting this parameters as 337, the communication performance for the message sized of 1KB can be improved by 9.4% , compared to the default case. When the message size is larger than 2 KB, the Recursive Double algorithm and the Ring algorithm shows similar performance.

2) MPI_Allreduce: The collective has only one parameter with two algorithms (Binomial Tree and RabenSeifner) in the target system. From Figure 2(b), we see that the switch point occurs when the communicated messsage is of 8 KB. This value is larger than the default setting, i.e., 2 KB. If we set the runtime parameter to be 8 KB for *MPI_Allreduce*, the performance can be improved by around 8.1%, e.g., for messages sized of 2 KB.

3) MPI_Alltoall: From Figure 2(c), when the message size ranges from 1 B to 1 MB, we observe that the *irecvs/isends* algorithm runs slower than either the *Bruck* algorithm or the *Pairwise* algorithm for *MPI_Alltoall*. Therefore, we have to avoid using this suboptimal choice. This can be achieved by setting `MPICH_ALLTOALL_SHORT_MSG_SIZE` to be 4 KB, while its default value is 256 B. In this way, we can improve the performance by around 25% for the messages sizing 256 B. When the `MPICH_ALLTOALL_MEDIUM_MSG_SIZE` (default values is 32 KB) is set as 4 KB, we can improve the performance upto 9.5% when the message size is 32 KB.

4) Other MPI collectives: For *MPI_Bcast* (Figure 2(d)) and *MPI_Reduce_scatter* (Figure 2(e)), the binomial tree algorithm consistently performs the best in the corresponding

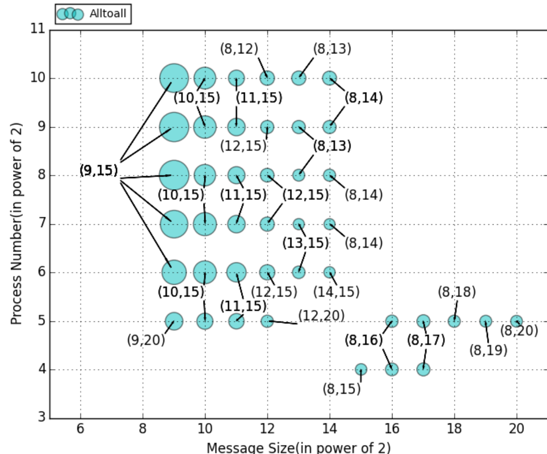


Figure 3. The best configuration under different numbers of processes

communication algorithms. Meanwhile, the long message algorithm delivers a poor performance. In addition, the case of *MPI_Gather* (Figure 2(g)) is similar to that of *MPI_Scatter* (Figure 2(h)), which demonstrates similar performance between short message and long message algorithms.

By quantitatively benchmarking each MPI collective, we can obtain the performance curves. As can be seen from Figure 2, different algorithms have a significant performance impact on MPI operations, e.g., *MPI_Alltoall*, *MPI_Allgather*, and *MPI_Allreduce*. The short message algorithm of *MPI_Bcast*, *MPI_Reduce_scatter* and *MPI_Reduce* can yeild a better performance than the other algorithms. Meanwhile, *MPI_Gather* and *MPI_Scatter* present a very slight performance change among their communication algorithms. When using a different process configuration, the MPI collectives will exhibit different performance. For example, when using 16 processes, the performance of *Reduce_scatter* changes significantly among communication algorithms, while the performance of *Allreduce* keeps stable when using less than 128 processes.

B. Best configurations of different functions

By synthesizing the performance curves shown in Figure 2, given an application with specific process numbers and message sizes, we can conclude the optimal settings for runtime parameters corresponding to specific application features. At the same time, we execute the *osu_alltoall* programs in the default and best configurations so as to get the speedup of performance, as shown in Figure 3.

For MPI_Alltoall, different settings and their performance depend on the application features, i.e., the message/problem size and the number of processes. In Figure 3, the Alltoall disk represents the *osu_alltoall* application. The horizontal axis represents the message size, and the vertical axis denotes the number of processes (both the message

Table IV
THE ACHIEVED KERNEL SPEEDUP WITH THE BEST CONFIGURATIONS

no.of proc.	Msg Size(B)	osu_ alltoall	FT	MPIFFT	MPI R.A.
64	8K	1.315	1.022	-	-
	16K	1.063	-	1.091	-
128	16K	1.021	1.044	1.322	-
256	8K	1.183	1.18	1.368	1.09
	16K	1.225	-	1.347	-
512	4K	1.452	1.24	-	-
	8K	1.453	-	1.238	1.1
	16K	1.446	1.237	1.255	-
1024	4K	1.756	-	1.407	-
	8K	1.701	1.317	1.337	1.13
	16K	1.642	-	1.353	-

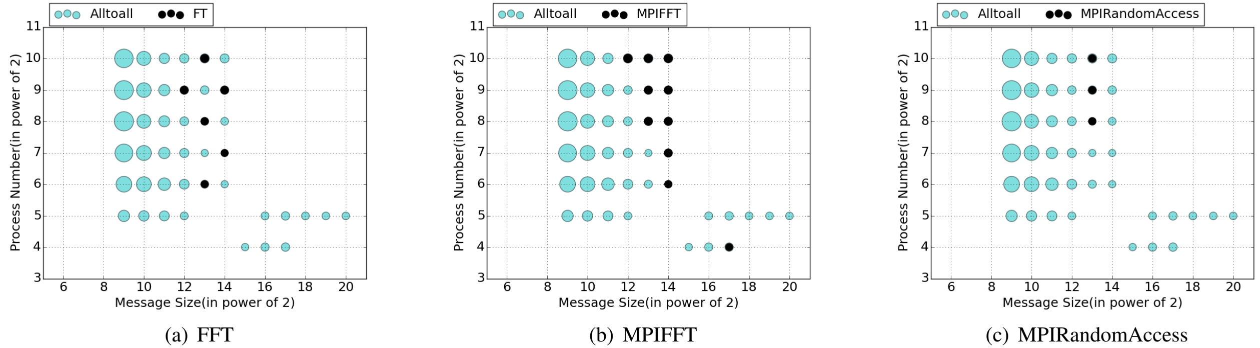
sizes and process numbers are in the form of power-of-2). The two entries within the parentheses, labelled as (A, B), represent the best configurations. The configuration contains two runtime parameters of the *MPI_Alltoall* operation, and they are `MPICH_ALLTOALL_SHORT_MSG_SIZE` and `MPICH_ALLTOALL_MEDIUM_MSG_SIZE` (both are in the form of power-of-2). The circle area represents the performance speedup when using the best configurations. Note that our approach works not only for *alltoall*, but for other MPI collectives such as *MPI_Allreduce* and *MPI_Allgather*. This work examines the performance of alltoall communications because they are among the most wide-spread collectives in parallel applications.

V. MODEL EVALUATION

In this section, we run the NPB and HPCC to demonstrate the usefulness of the obtained runtime configurations from our model. According to the best configurations of *alltoall* operation from Figure 3, we set the runtime parameters before running the `FT` of NPB, the `MPIFFT` and `MPIRandomAccess` of HPCC, whose message sizes in different scales are shown in Table III. It is worth noted that all the kernels are not including all kinds of message sizes as the *osu_alltoall* does. Figure 4 shows the execution of the three kernels with best configurations, as well as the distinction with each other. Table IV shows the detailed speedup of kernels.

We can conclude from the Figure 4 and Table IV:

- `FT`. When the number of processes is no more than 64, the performance improvement is not obvious. When using 64 processes, we note only 2% increase for CLASS A. But when the number of processes is more than 64, our approach can significantly improve the performance. When the message size is 8 KB and the number of processes is 1024, the parameters are configured to be (8,13), and the speedup reaches its largest, which is around 31.7%.
- `MPIFFT`. When the number of processes is 32, no performance improvement can be observed, while with other process configurations, the performance improvement is modest. When the message size is 4KB and



(a) FFT (b) MPIFFT (c) MPIRandomAccess

Figure 4. Diagrams show how the knowledge of microbenchmark profiling can be applied to optimize *unseen* programs for the Alltoall collective operation. Given the number of processes (y-axis) and the message size (y-axis), we can quickly find the optimal algorithm from previous profiling information. Here the larger the circle, the better the performance will be. The black disks show how the resulting performance of a testing benchmark and its runtime parameter, FT (a), MPIFFT (b), and MPIRandomAccess (c), lies on the profiling space.

the number of processes is 1024, the parameters are configured to be (8, 12), with an aim to achieve the largest performance improvement (i.e., 40.7%).

- `MPIRandomAccess`. When the message size is 8KB with over 128 processes the performance of the *alltoall* operation can be improved significantly. When the message size is 8KB with 1024 processes, we can achieve a performance improvement of 13% with the configuration of (8, 13).

To summarize, we can determine the runtime parameters by an empirical approach in the case of different scales for a target application. Such an approach can bring an improvement in performance. When using less than 64 processes, we consider that the computation, rather than the communication, takes up the majority of the end-to-end execution time. Thus, the performance change of communication time has little effect on the overall time. When the number of processes becomes larger, the communication time takes a larger part. Thus, the changes of the *alltoall* communications will significantly improve the applications performance.

VI. Related Work

There is an extensive body of work on performance optimization of MPI programs. One of the most popular method is focused on improving the collective communication and implementations of algorithms on distributed systems, with the goals of minimizing latency for short messages and minimizing bandwidth use for long messages. Researchers have developed many algorithms applied to the MPI collective operations [5, 12, 20, 21, 31, 36], but the efforts in optimization of the communication algorithms is still going on. In [11], Faraj and Yuan input the topology information to generate topology specific communication routines, and then select the best implementation among different topology specific and topology unaware algorithms with an empirical approach. Godwin et. al. [13] similarly optimized the performance of MPI communication by scheduling messages according to topology information. Hasanov et.al. [17, 18] proposed new approaches taking into account the hierarchical topology-oblivious transformation of existing communication algorithms. Li et.al. [22] introduced a cache-oblivious algorithms allocating the send and receive buffers on shared heap and use Morton order to guide the memory copies. Unlike the above approaches to improve the communication algorithms, our method in this paper focuses on the selection of a more efficient communication algorithms according to message sizes and process scales.

Besides the improvement of communication algorithms, tuning the parameters of the MPI communication library can be used to a particular system. Mohamad C. et al. [6] proposed OTPO, a tool that can optimize OpenMPI runtime parameters, which gives users and system researchers the possibility to make their environment meet the requirement of performance. In [28, 29], the main idea is to conduct an off-line training phase to derive the “best” configurations of OpenMPI for each target architecture based on machine learning algorithms. Jha et.al. [19] executed a parameter set to determine the expected improvement of a collective operation using Hockney’s model and the LogGP model. Similar to our approach, the basic idea is to identify the set of collective operations used by an application and capture their communication information for tuning parameters.

Our study aims at finding the optimal configuration in MPICH for a target supercomputer system, based on the observation of message size and algorithms representation in applications. Furthermore, our method is more general and straightforward to select the right communication algorithms, without breaking up any existing algorithms.

Machine learning based predictive modeling has been employed for a wide range of optimization tasks [42]. These include code optimization [7, 9, 16, 25, 26, 34, 37, 39, 40, 41, 43, 44, 45, 47], task scheduling [10, 14, 15, 24, 32, 33], model selection [35], etc. Our work can be integrated in a learning framework to predict the optimal MPI communica-

tion setting, and we leave this as our future work.

VII. Conclusion

This paper has presented a novel approach to tune MPI collective algorithms. Our approach automatically chooses, at runtime, which of the available algorithms to use for a given communication operation. The runtime decision is based on the message size that is available at program execution time. To develop the runtime model, we first profile a set of microbenchmarks to understand how the collective algorithm affect the communication performance for each target operation. We then apply the extracted knowledge to new, unseen MPI programs. Our approach reduces the human involvement in tuning optimization heuristics, allowing the strategy to be quickly updated in case of hardware or MPI library changes. We evaluate our approach by applying it to optimize the communication algorithms for eight communication primitives provide by MPICH. Our evaluation platform is a 384-node computing cluster of the Tianhe-2 supercomputer. We test our approach on the NPB and HPCC parallel benchmark suites. Experimental results show that our approach achieves, on average, 22.7%(up to 40.7%) improvement over the default setting.

Acknowledgment

This work was supported in part by the National Key Research and Development Program of China (2017YFB0202200), the Advanced Research Project of China (31511010203), Open Fund (No. 201503-02) from State Key Laboratory of High Performance Computing, the Research Program of NUDT (ZK18-03-10), and the National Natural Science Foundation of China under grant agreements 61602501 and 61872294.

References

[1] Integrated performance monitoring. http://ipm-hpc.sourceforge.net/, 2019.
[2] Mpi forum. http://www.mpi-forum.org, 2019.
[3] Mpich-a portable implementation of mpi. https://www.mpich.org/, 2019.
[4] Mvapich ohio state university micro benchmark. http://mvapich.cse.ohio-state.edu/benchmarks/, 2019.
[5] Jehoshua Bruck et al. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE TPDS*, 1997.
[6] Mohamad Chaarawi et al. A tool for optimizing runtime parameters of open mpi. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2008.
[7] Donglin Chen et al. Optimizing sparse matrix–vector multiplications on an armv8-based many-core architecture. *International Journal of Parallel Programming*, 2018.
[8] Sudheer Chunduri et al. Characterization of mpi usage on a production supercomputer. In *Characterization of MPI Usage on a Production Supercomputer*, 2018.
[9] Chris Cummins et al. End-to-end deep learning of optimization heuristics. In *PACT '17*, 2017.
[10] Murali Krishna Emani et al. Smart, adaptive mapping of parallelism in the presence of external workload. In *CGO '13*, 2013.
[11] Ahmad Faraj and Xin Yuan. Automatic generation and tuning of mpi collective communication routines. In *ICS*, 2005.
[12] Ahmad Faraj and Xin Yuan. Message scheduling for all-to-all personalized communication on ethernet switched clusters. In *IPDPS*, 2005.
[13] Jeffrey Godwin et al. Runtime optimization of broadcast communications using dynamic network topology information from mpi. In *HPCC*, 2012.
[14] Dominik Grewe et al. A workload-aware mapping approach for data-parallel programs. In *HiPEAC '11*, 2011.
[15] Dominik Grewe et al. Opencl task partitioning in the presence of gpu contention. In *LCPC '13*, 2013.
[16] Dominik Grewe et al. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *CGO '13*, 2013.
[17] Khalid Hasanov et al. Topology-oblivious optimization of mpi broadcast algorithms on extreme-scale platforms. *Simulation Modelling Practice and Theory*, 2015.
[18] Khalid Hasanov and Alexey Lastovetsky. Hierarchical redesign of classic mpi reduction algorithms. *The Journal of Supercomputing*, 2017.
[19] Shweta Jha and Edgar Gabriel. Impact and limitations of point-to-point performance on collective algorithms. In *CCGrid*, 2016.
[20] Laxmikant V Kalé and ohters. A framework for collective personalized communication. In *IPDPS*, 2003.
[21] Amit Karwande et al. Cc–mpi: a compiled communication capable mpi prototype for ethernet switched clusters. In *ACM Sigplan Notices*, 2003.
[22] Shigang Li et al. Cache-oblivious mpi all-to-all communications based on morton order. *IEEE TPDS*, 2018.
[23] Xiang-Ke Liao et al. High performance interconnect network for tianhe system. *Journal of Computer Science and Technology*, 2015.
[24] Vicent Sanz Marco et al. Improving spark application throughput via memory aware task co-location: a mixture of experts approach. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, 2017.
[25] William F Ogilvie et al. Fast automatic heuristic construction using active learning. In *LCPC '14*, 2014.
[26] William F Ogilvie et al. Minimizing the cost of iterative compilation with active learning. In *CGO '17*, 2017.
[27] Zhengbin Pang et al. The th express high performance interconnect networks. *Frontiers of Computer Science*, 2014.
[28] Simone Pellegrini et al. Optimizing mpi runtime parameter settings by using machine learning. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, 2009.
[29] Simone Pellegrini and ohters. Tuning mpi runtime parameter setting for high performance computing. In *IEEE International Conference on Cluster Computing Workshops*, 2012.
[30] Jelena Pješivac-Grbović et al. Performance analysis of mpi collective operations. *Cluster Computing*, 2007.
[31] R Rabenseifner. A new optimized mpi reduce and allreduce algorithms, 1997.
[32] Jie Ren et al. Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach. In *INFOCOM '17*, 2017.
[33] Jie Ren et al. Proteus: network-aware web browsing on heterogeneous mobile systems. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 379–392, 2018.
[34] Ben Taylor et al. Adaptive optimization for opencl programs on embedded heterogeneous systems. In *LCTES '17*, 2017.
[35] Ben Taylor et al. Adaptive deep learning model selection on embedded systems. In *LCTES '18*, 2018.
[36] Rajeev Thakur et al. Optimization of collective communication operations in mpich. *IJHPCA*, 2005.
[37] Georgios Tournavitis et al. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09*, 2009.
[38] Bibo Tu et al. Multi-core aware optimization for mpi collectives. In *IEEE Cluster*, 2008.
[39] Zheng Wang et al. Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems. *ACM TACO*, 2014.
[40] Zheng Wang et al. Exploitation of gpus for the parallelisation of probably parallel legacy code. In *CC '14*, 2014.
[41] Zheng Wang et al. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM TACO*, 2014.
[42] Zheng Wang and Michael O'Boyle. Machine learning in compiler optimization. *Proc. IEEE*, 2018.
[43] Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: A machine learning based approach. In *PPoPP '09*, 2009.
[44] Zheng Wang and Michael FP O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT '10*, 2010.
[45] Zheng Wang and Michael FP O'boyle. Using machine learning to partition streaming programs. *ACM TACO*, 2013.
[46] Min Xie et al. Tianhe-1a interconnect and message-passing services. *IEEE Micro*, 2012.
[47] Peng Zhang et al. Auto-tuning streamed applications on intel xeon phi. In *IPDPS*, 2018.