Automatic Verification of Self-consistent MPI Performance Guidelines

Sascha Hunold(✉), Alexandra Carpen-Amarie, Felix Donatus Lübbe, and Jesper Larsson Träff

Research Group for Parallel Computing, TU Wien, Vienna, Austria
{hunold,carpenamarie,luebbe,traff}@par.tuwien.ac.at

Abstract. The Message Passing Interface (MPI) is the most commonly used application programming interface for process communication on current large-scale parallel systems. Due to the scale and complexity of modern parallel architectures, it is becoming increasingly difficult to optimize MPI libraries, as many factors can influence the communication performance. To assist MPI developers and users, we propose an automatic way to check whether MPI libraries respect self-consistent performance guidelines for collective communication operations. We introduce the PGMPI framework to detect violations of performance guidelines through benchmarking. Our experimental results show that PGMPI can pinpoint undesired and often unexpected performance degradations of collective MPI operations. We demonstrate how to overcome performance issues of several libraries by adapting the algorithmic implementations of their respective collective MPI calls.

Keywords: MPI · Collectives · Performance guidelines · Benchmarking

1 Introduction

Communication libraries implementing the Message Passing Interface (MPI) are major building blocks for developing parallel, distributed, and large-scale applications for current supercomputers. The performance of parallel codes is therefore highly dependent on the efficiency of MPI implementations. Much research is currently conducted to cope with the problems of exascale computing in MPI.

Assessing the performance of MPI implementations is vital for developers, vendors, and users of the libraries. However, the performance of MPI libraries can be measured in different ways. A common approach is to run a set of MPI micro-benchmarks, such as SKaMPI [12] or ReproMPI [7]. Micro-benchmarks usually report the measured (mean or median) run-time of a given MPI function for different message sizes, e.g., the run-time of `MPI_Bcast` for broadcasting a 1 Byte message. Developers can gain insights on how the run-time of an MPI function depends on the message size for a fixed number of processes. It is also

A. Carpen-Amarie and F.D. Lübbe—This work was supported by the Austrian Science Fund (FWF): P26124 and P25530.

P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 433–446, 2016.
DOI: 10.1007/978-3-319-43659-3_32

possible to assess the scalability of MPI functions when the number of processes is increased and the message size stays fixed.

Verifying self-consistent MPI performance guidelines is an alternative, orthogonal method for analyzing the performance of MPI libraries [15]. This approach does not require explicit performance models. Instead, performance guidelines form a set of rules that an MPI library is expected to fulfill. A performance guideline usually defines an upper bound on the run-time behavior of a specialized MPI function. For example, one performance guideline states that a call to `MPI_Scatter` of n data elements should "not be slower" than a call to `MPI_Bcast` with n data elements, as the semantics of an `MPI_Scatter` operation could be emulated using `MPI_Bcast` [15]. Only minor efforts have been made to systematically test self-consistent performance guidelines for MPI implementations in practice, one example being the mpicroscope benchmark [14]. To close this gap, we introduce the benchmarking framework PGMPI that can automatically verify performance guidelines of MPI libraries.

We make the following contributions: (1) We propose the benchmarking framework PGMPI to detect performance-guideline violations. (2) We present a systematic, experimental verification of performance guidelines for several MPI libraries. (3) We examine different use cases, for which the detection of guideline violations enabled us to tune and improve the libraries' performance.

In Sect. 2, we state the scientific problem and introduce our notation. We continue with summarizing related work and comparing it to our approach in Sect. 3. We introduce the PGMPI framework in Sect. 4 and present an experimental evaluation of different MPI libraries using it in Sect. 5. We summarize our findings and conclude in Sect. 6.

2 Problem Statement and Notation

Träff et al. [15] introduced self-consistent performance guidelines for MPI libraries as follows: The run-time of two MPI functionalities A and B can be ordered using the relation $\preceq$ as $\texttt{MPI\_}A(n) \preceq \texttt{MPI\_}B(n)$, which means that functionality $\texttt{MPI\_}A(n)$ is possibly faster than functionality $\texttt{MPI\_}B(n)$ for (almost) all communication amounts n. Performance guidelines are defined for a fixed number of processes p, and thus, they do not mention p explicitly. However, the communication volume per process may vary depending on the semantics of a given MPI function and the number of processes. For example, in the case of `MPI_Bcast`, the total size n is equal to the message size being transferred to each process. In contrast, the individual message size for an `MPI_Scatter` is a fraction of the total communication volume n, i.e., each process receives n/p elements.

We examine three types of performance guidelines: (1) *monotony*, (2) *split-robustness*, and (3) *pattern*. The monotony guideline

$$\texttt{MPI\_}A(n) \preceq \texttt{MPI\_}A(n+k)$$

ensures that communicating a larger volume should not decrease the communication time.

The split-robustness guideline

$$\texttt{MPI\_A}(n) \preceq k\ \texttt{MPI\_A}(n/k)$$

states that communicating a total volume of n data elements should not be slower than sending $\frac{n}{k}$ elements in k steps.

Pattern guidelines define upper bounds on the performance of MPI communication operations. The idea is that a specialized MPI function should not have a larger running time than a combination of other MPI operations, which emulate the functionality of the specialized function. Let us consider the following *pattern* performance guidelines:

$$\texttt{MPI\_Scatter}(n) \preceq \texttt{MPI\_Bcast}(n)\,, \text{ and}$$
$$\texttt{MPI\_Bcast}(n) \preceq \texttt{MPI\_Scatter}(n) + \texttt{MPI\_Allgather}(n) \quad .$$

The first states that MPI_Scatter should not be slower than MPI_Bcast. The reason is that the semantics of MPI_Scatter can be implemented using MPI_Bcast, by broadcasting the entire vector before processes take their share depending on their rank. The second guideline states that a call to MPI_Bcast should be at least as fast as a combination of MPI_Scatter and MPI_Allgather, which we call the *mock-up* version of MPI_Bcast, as it emulates its semantics [2].

3 Related Work

Collective communication operations are a central part of the MPI standard, as they are essential for many large-scale applications. Chan et al. [2] provide an overview of typical, blocking collectives and their implementations, as well as lower bounds for the communication cost of each function. For different network topologies, the authors devise algorithms that achieve the lower bounds for either the latency or the bandwidth component. As the model of parallel computation in this paper is rather simplistic, we aim to complement this study by carefully benchmarking MPI collectives on actual hardware.

Träff [14] proposed the MPI benchmark mpicroscope, which can verify two self-consistent performance guidelines: "split-robust" and "monotone". In the present work, we extend this functionality by testing various pattern violations, using the experimental framework that was proposed by Hunold and Carpen-Amarie [7] for better reproducibility of the experimental results. While we focus on performance guidelines for collectives, previous works have also formulated performance guidelines for derived datatypes [4] and MPI-IO operations [5].

As hardware and software factors can influence the performance of MPI collectives, tuning MPI parameters is an essential part for achieving high performance, when installing an MPI library. Yet, optimizing and tuning MPI operations are orthogonal steps compared to the verification of self-consistent performance guidelines, i.e., the latter can help us to verify whether run-times of collectives are consistent in terms of expected performance. For example, the guidelines can be used to ensure that Gather is faster than Allgather for the same problem size. For that reason, if a violation occurs, it usually means that one collective can be tuned. To optimize the latency of collectives at run-time, one can employ the STAR-MPI routines [3]. When a call to a specific MPI function is issued, STAR-MPI selects one of the available algorithms and measures its run-time. When STAR-MPI has enough knowledge about the performance of different algorithms, it is able to pick a good algorithm for a specific case.

Selecting the right algorithm to implement a given MPI function is only one step towards tuning MPI libraries. Another problem is finding the right parameter settings that run-time systems of MPI libraries like Open MPI or MVAPICH offer. Chaarawi et al. [1] introduced the OTPO tool that can be used to tune Open MPI run-time parameters. OTPO takes as input the run-time parameters to be tuned as well as their respective ranges, and then starts measuring for all combinations of parameter values. Another approach to tune Open MPI parameters has been proposed by Pellegrini et al. [10], where the parameter values are predicted using machine learning techniques.

The performance guidelines are formulated as a function of the communication volume. It is also possible to examine the scalability of MPI collectives when increasing the number of processes. Shudler et al. [13] proposed a framework to compare performance characteristics of HPC applications with a theoretical performance model. The framework fits the recorded benchmarking data to analytic speedup functions and compares the experimentally determined scalability behavior to this expected performance model. A model mismatch indicates a scalability problem of the parallel code section.

4 PGMPI: Verifying MPI Performance Guidelines

We now introduce the PGMPI framework to verify self-consistent performance guidelines of MPI libraries. In the first step, PGMPI experimentally determines the number of repetitions needed to obtain stable, reproducible run-time measurements (*Step NREP*). In the second step, the framework performs run-time measurements of all functions for which performance guidelines are formulated (*Step MEASURE*). The data analysis and the statistical verification of performance guidelines is carried out in the last step (*Step ANALYZE*).

4.1 Obtaining Reproducible Results

We start by looking at the main (second) step of PGMPI (*Step MEASURE*), in which the run-times of MPI functions and their emulating counterparts are measured. The guideline-checking program takes as input a set of pattern guidelines, each defined by a pair consisting of an MPI function and its emulating mock-up function. Our PGMPI framework will measure the run-time of one of the specified MPI functions f for all given message sizes m_i and the number of processes p that are given in the input file. Within one call to `mpirun`, each individual measurement for (f, m_i) is repeated r_i times, where r_i is defined for each m_i. As we expect that mean (or median) run-times vary between different calls to `mpirun` [7], the PGMPI framework measures the run-time of each MPI function f over R `mpiruns`.

4.2 Determining the Number of Repetitions

A major problem in MPI benchmarking is the question of how long (how many times) to measure. We need to find the right trade-off between time and measurement stability. One way of dealing with this problem is by executing the experiment sufficiently often, e.g., 1000 times. This would alleviate the problem of low measurement stability, but most often, we cannot afford long-running benchmarking experiments. Therefore, we formulate the following problem:

Definition 1. *The NREP problem is to find a suitable number of repetitions r_i for the tuple (f, m_i, p), such that the obtained run-time metric after r_i repetitions of function f with m_i Bytes on p processes is reproducible between different calls to `mpirun`. Reproducible in this case means that the distribution of the measured values (for a specific metric) obtained from R `mpiruns` has a small variance.*

We have experimented with various ways of estimating the number of repetitions needed to obtain reproducible results. One possibility is to monitor the relative standard error of the mean (RSE). SKaMPI, for example, stops the measurements when the RSE falls below a threshold of 0.1 [12]. Although we have tested many different ways to solve the NREP problem, we could not find a generally superior approach. We therefore designed the NREP predictor for *Step NREP* of the PGMPI framework in a flexible manner. The framework currently provides three different methods (metrics) for solving the NREP problem, but new metrics can be added. The NREP prediction may stop

1. when the **relative standard error** (RSE) is smaller than some predefined threshold t_{RSE}; or
2. when the **coefficient of variation of the mean run-time** (COV_{mean}) is smaller than some predefined threshold $t_{COV_{mean}}$. The value of the COV_{mean} is computed over the last $w_{COV_{mean}}$ means (window size); or
3. when the **coefficient of variation of the median run-time** (COV_{median}) is smaller than some predefined threshold $t_{COV_{median}}$ using a window size of $w_{COV_{median}}$.

Users can choose the NREP prediction method on the command line as follows:

```
mpirun -np 4  ./mpibenchmarkPredNreps --calls-list=MPI_Reduce --msizes-list=8
--rep-prediction min=20,max=1000,step=10 --pred-method=rse --var-thres=0.025
```

It is also possible to combine different metrics, i.e., the NREP prediction stops when all selected metrics have been positively evaluated. An example is shown in Fig. 1, in which both the RSE and the COV_{mean} need to be below a specific threshold (marked with horizontal lines). The prediction function for the RSE metric stops after 85 iterations, at which the COV_{mean} value is also below its threshold. As a result, 85 is the number of iterations that will be used when collecting benchmark data in *Step MEASURE*. To cope with the run-time variation between different `mpiruns`, we perform three NREP predictions for each message size and select the maximum number of repetitions obtained.

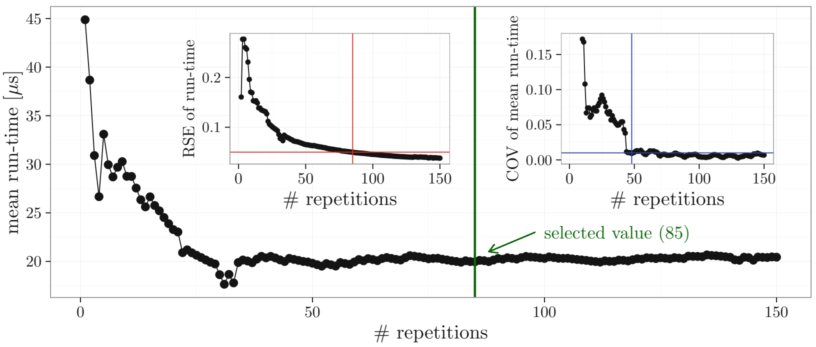


Fig. 1. Example of estimating the required number of repetitions for `MPI_Allgather` (16 B, 16×1 processes, *Jupiter*, $t_{RSE} = 0.025$, $t_{COV_{mean}} = 0.01$, $w_{COV_{mean}} = 20$)

4.3 Statistically Verifying Performance Guidelines

After gathering the measurement results, PGMPI can proceed to *Step ANALYZE*, which consists of the data processing and the verification of performance guidelines. We now explain which statistical methods are applied for guideline verification. For each MPI function, for which guidelines were formulated, the experimental results comprise R (number of `mpirun`s) data sets for a specific number of processes p. Each data set contains r_i run-time measurements for a specific message size m_i. We first reduce the number of measurements per tuple ($\texttt{mpirun}_j$, m_i, p) to a single value, by computing the median run-time over the r_i measurements. In this way, we obtain a distribution of R medians (median run-times) for each message size m_i and processes p. The various performance guidelines will then be verified using these distributions of medians.

Monotony Guideline. PGMPI checks for each pair of adjacent message sizes m_i and m_j, $m_i < m_j$ that the run-time of an MPI function with a message size of m_i is not larger than the run-time with a size of m_j. We use the *Wilcoxon rank-sum test* [6] to test whether the distribution of medians at m_i is smaller or equal than the one at m_j. If the test rejects our hypothesis, we have statistical confidence (at the provided confidence level) that the monotony between message sizes m_i and m_j is violated.

Split-Robustness Guideline. We want to verify that sending a message of size m_j by transferring k packets of size $m_i < m_j$ is not faster than sending only one message of size m_j. We are only given the run-time distribution of one MPI function at m_i. Unfortunately, we have no knowledge about the shape of the run-time distribution when we communicate messages of size m_i in k rounds. As a matter of fact, we cannot simply shift the distribution at m_i by some constant factor, and therefore, we decided to rely on (and to compare) the median values of the distributions.

Table 1. Overview of parallel machines used in the experiments

Name	Hardware	MPI libraries/Compiler
Jupiter	36 × Dual Opteron 6134 @ 2.3 GHz	NEC MPI 1.3.1, MVAPICH2-2.1
	IB QDR MT26428	Open MPI 1.10.1/ gcc 4.4.7
VSC-3	2000 × Dual Xeon E5-2650V2 @ 2.6 GHz	Intel MPI Library 5.0 (Update 3)
	IB QDR-80	gcc 4.4.7

Since we measure the run-time of MPI functions only for a limited number of message sizes, we compute the factor $k = \min_{l \in \mathbb{N}}(lm_i \geq m_j)$, which denotes the smallest multiple of m_i such that the resulting product is at least m_j. Notice that we explicitly allow lm_i to be larger than m_j, which enables us to check whether sending two messages of size 1024 B is faster than sending one message of size 2000 B. The PGMPI framework checks whether the time to communicate messages of size m_i in k rounds is smaller than the run-time for m_j. If we find such a violation for a message size m_j, we only report the largest message size m_i (the smallest factor k) for which the violation occurred; otherwise too many violations would be reported in some cases. It often happens that the predicted run-time for lm_i is very similar to the run-time for m_j. To avoid reporting split-robustness violations for which only marginal relative run-time differences have been measured, we use a 5 % tolerance level to verify this guideline. Currently, PGMPI does not empirically test whether communicating k messages of size m_i is indeed faster than communication a message of size m_j in practice. This additional check would require an additional benchmarking round, and might be added to PGMPI later.

Pattern Guidelines. The verification of pattern guidelines is done similarly to checking the monotony guideline, except that we now compare two run-time distributions of two distinct functions: an MPI function and its mock-up version. We apply the *Wilcoxon rank-sum test* on the two distributions to test whether the run-time distribution of the MPI function is not significantly shifted to the right of the distribution obtained with the mock-up version ("to the right" means larger run-time). If this is the case, PGMPI reports a pattern violation. Alternatively, the *Kolmogorov-Smirnov test* [6] can be employed, as it is less sensitive to ties. Overall, both tests led to similar results in the majority of the considered cases.

5 Experimental Evaluation and Results

We evaluate our proposed PGMPI framework[1] experimentally using the hardware and software setup listed in Table 1. First, we present a summary of detected performance-guideline violations for several MPI libraries. Second, we demonstrate in two case studies that the knowledge about specific guideline violations can help tuning and adapting MPI implementations to parallel systems. Please refer to our technical report [8] for more details about the experimental evaluation.

[1] https://github.com/hunsa/pgmpi.

Table 2. Performance-guideline violations of different MPI libraries ($R = 10$); violation types: **m**onotony, **s**plit-robustness, **p**attern; message sizes between 1 B and 100 KiB

(a) *Jupiter*

#processes	type	MVAPICH2-2.1	NEC MPI 1.3.1	Open MPI 1.10.1
16x1	m	7/9	6/9	7/9
16x1	s	1/9	0/9	3/9
16x1	p	12/15	7/15	9/15
32x16	m	5/9	4/9	4/9
32x16	s	3/9	0/9	3/9
32x16	p	8/15	7/15	7/15

(b) *VSC-3*

#processes	type	Intel MPI 5.0
16x16	m	7/9
16x16	s	6/9
16x16	p	13/15
64x16	m	6/9
64x16	s	7/9
64x16	p	11/15

5.1 Assessing the Guideline Compliance of MPI Libraries

We used the PGMPI framework to verify the performance guidelines listed in Appendix A for different MPI libraries. On *Jupiter*, we evaluated NEC MPI 1.3.1, MVAPICH2-2.1, and Open MPI 1.10.1. The NEC MPI 1.3.1 library was delivered by NEC pre-compiled for our system and we therefore do not know all internals. The other two libraries, MVAPICH2-2.1 and Open MPI 1.10.1, were compiled using the default settings. On *VSC-3*, we recorded guideline violations for the proprietary Intel MPI Library 5.0 (Update 3).

Table 2 presents an overview of the detected guideline violations for several MPI libraries on *Jupiter* and *VSC-3*. For the monotony and the split-robustness guidelines, the table shows the number of MPI functions for which violations occurred, e.g., for MVAPICH2-2.1 using 16×1 processes, PGMPI found seven monotony violations among the nine tested MPI collectives. For the pattern guidelines, we verified the 15 guidelines provided in Appendix A. If a guideline violation is found for any message size of a particular MPI function, we say that this particular guideline is unsatisfied, i.e., a violation is only counted once across all message sizes. We can observe in Table 2 that the monotony and the split-robustness guidelines are violated by approximately 50 % of the collectives. The table also reveals that more than 40 % of the examined pattern guidelines were violated. The guideline violations occurred across different numbers of processes, message sizes, libraries, and machines. We therefore contend that there is a large potential for optimization of the individual MPI libraries on these machines.

Table 3 compares the detected pattern violations for the three libraries on *Jupiter*. Except for two guidelines, we found violations for all other pattern guidelines in at least one MPI library. The experimental results clearly suggest that Reduce-like functions should be improved and tuned in all MPI libraries, which are: `MPI_Allreduce`, `MPI_Reduce`, `MPI_Reduce_scatter`, and `MPI_Reduce_scatter_block`.

A detailed view on the detected guideline violations for MVAPICH2-2.1 on *Jupiter* is given in Table 4. For this MPI library, we observe a couple of monotony violations. For short message sizes (<32 B), the absolute difference in run-times is very small, and thus, fixing these cases has low priority. Monotony violations occur for larger messages when the message size is not a power of two (e.g., between 10000 B and 16384 B). These cases could be investigated in more

Table 3. Pattern guideline violations of different MPI libraries with 32×16 processes on *Jupiter*, $R = 10$; message sizes between 1 B and 100 KiB

Guideline	MVAPICH2-2.1	NEC MPI 1.3.1	Open MPI 1.10.1
`MPI_Allgather` $\preceq$ Allreduce	●		
`MPI_Allgather` $\preceq$ Alltoall			
`MPI_Allgather` $\preceq$ Gather+Bcast		●	
`MPI_Allreduce` $\preceq$ Reduce+Bcast	●	●	●
`MPI_Allreduce` $\preceq$ Reduce_scatter_block+Allgather			●
`MPI_Bcast` $\preceq$ Scatter+Allgather		●	●
`MPI_Gather` $\preceq$ Allgather			
`MPI_Gather` $\preceq$ Reduce	●		
`MPI_Reduce_scatter_block` $\preceq$ Reduce+Scatter	●	●	●
`MPI_Reduce_scatter` $\preceq$ Allreduce	●	●	●
`MPI_Reduce_scatter` $\preceq$ Reduce+Scatterv		●	
`MPI_Reduce` $\preceq$ Allreduce	●		●
`MPI_Reduce` $\preceq$ Reduce_scatter_block+Gather	●		
`MPI_Scan` $\preceq$ Exscan+Reduce_local		●	●
`MPI_Scatter` $\preceq$ Bcast	●		

detail, as padding up the message to the next power of two could be an option. For split-robustness guidelines we can see potential for improvement only for larger message sizes. When analyzing the pattern guidelines, two cases stand out: `MPI_Allreduce` is slower than the emulating function using Reduce and Bcast for message sizes up to 2 KiB and `MPI_Reduce_scatter` exposes a performance degradation compared to Allreduce for almost all message sizes.

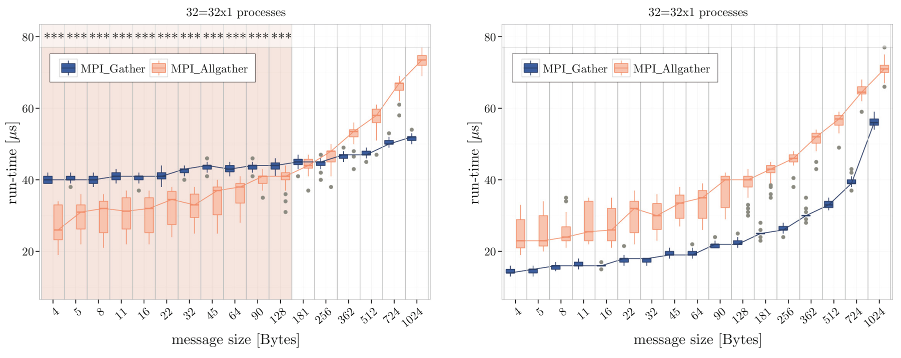
As it is impossible for library developers to provide suitable parameters for each individual installation, checking the compliance to performance guidelines can be seen as indicators for programmers and administrators, how to tune MPI libraries. Often, specific MPI libraries already provide efficient algorithms, and violations would not occur if the right algorithm were enabled for a specific case. We therefore show in the next section how violations can guide us to find more suitable algorithms and implementations for collective calls on a specific machine.

5.2 Case Study 1: `MPI_Gather` $\preceq$ `MPI_Allgather`, MVAPICH

We consider the violation of this performance guideline that was detected using 32×1 processes and MVAPICH2-2.1 on *Jupiter* and is shown in Fig. 2a. When the *Wilcoxon rank-sum test* reports a violation for a particular message size, we mark this case in the figure with a red background and add asterisks to show the statistical significance. Here, executing `MPI_Gather` using 32×1 processes (one process per compute node) is slower than performing a Gather using `MPI_Allgather`. Calling `MPI_Gather` in the default installation of

Table 4. Performance-guideline violations of MVAPICH2-2.1 using 32×16 processes on *Jupiter* ($R = 10$); violation types: **m**onotony, **s**plit-robustness, **p**attern

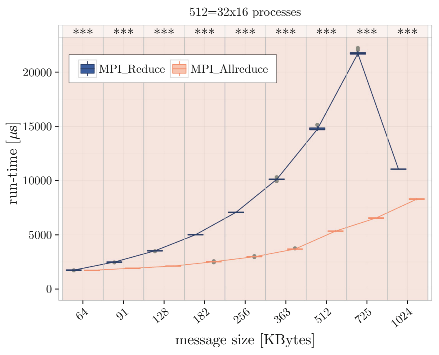
type	function	1	2	4	8	16	32	64	100	128	256	512	1024	1500	2048	4096	5000	8192	10000	16384	32768	102400
m	`MPI_Allgather`			•			•															
m	`MPI_Allreduce`							•												•		
m	`MPI_Gather`																			•		•
m	`MPI_Reduce`							•												•		
m	`MPI_Scatter`		•		•		•															
s	`MPI_Gather`														•	•	•	•	•		•	
s	`MPI_Reduce`																		•			
s	`MPI_Reduce_scatter_block`												•	•								
p	`MPI_Allgather` ⪯ Allreduce	•	•																			
p	`MPI_Allreduce` ⪯ Reduce+Bcast	•	•	•	•	•	•	•	•	•	•	•	•	•	•							
p	`MPI_Gather` ⪯ Reduce	•	•	•																		
p	`MPI_Reduce_scatter_block` ⪯ Reduce+Scatter	•	•										•	•								
p	`MPI_Reduce_scatter` ⪯ Allreduce	•	•					•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
p	`MPI_Reduce` ⪯ Allreduce																		•	•	•	•
p	`MPI_Reduce` ⪯ Reduce_scatter_block+Gather																			•	•	•
p	`MPI_Scatter` ⪯ Bcast	•	•	•																		



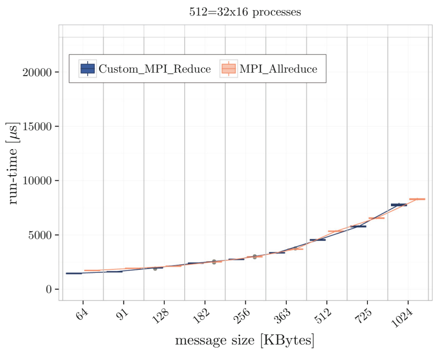
(a) With Violations (b) No Violations

Fig. 2. Verification of `MPI_Gather` ⪯ `MPI_Allgather` (a) before and (b) after changing the Gather implementation (MVAPICH2-2.1, *Jupiter*, $R = 30$, $r_i = 1000$)

MVAPICH2-2.1 will use the internal function `MPIR_Gather_intra` for the first 14 invocations and then switch to `MPIR_Gather_MV2_Direct` for subsequent calls. The direct implementation of Gather performs $(p - 1)$ `MPI_Irecvs` on the root process and an `MPI_Send` on the other processes. We can set the environment variable `MV2_USE_DIRECT_GATHER=0` to force MVAPICH to use `MPIR_Gather_intra` only. The intra-version on our machine uses a binomial tree algorithm to implement Gather, and forcing this algorithm fixes the violation (cf. Fig. 2b). Let us check that the algorithmic change for small message sizes is indeed favorable. We use the Hockney model for `MPI_Reduce` given by Pjesivac et al. [11], but omit the computational term. As the direct algorithm issues $(p - 1)$ receive operations, we obtain a run-time for small message sizes (we neglect the bandwidth term) of about 52.7 μs, for a network latency of roughly 1.7 μs. If we use a binomial tree



(a) With Violations



(b) No Violations

Fig. 3. Verification of `MPI_Reduce` $\preceq$ `MPI_Allreduce` with (a) original and (b) new Reduce implementation (Open MPI 1.10.1, *Jupiter*, $R = 5$, $r_i = 100$)

algorithm instead, the latency cost grows only logarithmically in the number of processes, i.e., $\log 32 \cdot 1.7\,\mu s = 8.5\mu s$. Even though our estimation does not perfectly match the experimental data, it explains why the binomial tree algorithm performs better.

5.3 Case Study 2: `MPI_Reduce` $\preceq$ `MPI_Allreduce`, Open MPI

In the second case study, we consider the guideline violations that occurred for `MPI_Reduce` using Open MPI 1.10.1 on the *Jupiter* system. Here, in contrast to the first case study, violations have only been measured for larger message sizes ($> 2^{16}$ B), but for various numbers of processes: 16×1, 32×1, 16×16, and 32×16. Figure 3 limits the view to message sizes for which violations were detected. Since Open MPI is highly configurable via the `MCA` parameters, we have tried to find parameter settings for `MPI_Reduce`, such that executing the latter would be faster than executing `MPI_Allreduce`. We have tried various segment sizes and fan-outs (where the parameters were applicable). Unfortunately, we failed to tune the parameters in such a way that the violations would disappear on our machine. For that reason, we implemented our own Reduce algorithm, which is based on the `MPI_Allreduce` algorithm found in Open MPI 1.10.1. Here, `MPI_Allreduce` is implemented using a Reduce-scatter followed by an Allgatherv on a ring of processes [9]. We modified this algorithm to become an `MPI_Reduce` by replacing the final Allgatherv by a Gatherv to the root. The Gatherv was realized using a direct Irecv/Send scheme. In the MPI semantics of Reduce, only the root process has a receive buffer. We therefore need to allocate additional buffer space to send and receive data segments in the Reduce-scatter phase. We found that executing `malloc` in each Reduce call has a severe impact on the performance of Reduce. To overcome this problem, we allocate a temporary buffer outside of Reduce but accessible to the Reduce implementation. This modification helped us to significantly speed up the run-time, and made this

Reduce implementation a suitable candidate to be included in the Open MPI library. In sum, our Reduce implementation avoids violations for larger messages sizes, as shown in Figure 3b.

6 Conclusions

The experimental verification of performance guidelines is an orthogonal approach to traditional MPI library tuning. It allows to find performance degradations of MPI functions, which would be hidden otherwise. For example, it is possible to optimize several existing implementations of `MPI_Gather`, but even the fastest of these Gather algorithms might be slower than the call to `MPI_Allgather`.

We have introduced the PGMPI framework to verify self-consistent performance guidelines of MPI functions. Currently, the framework supports blocking MPI collective communication operations, but it can be extended to support MPI point-to-point communication operations and derived datatypes. We have evaluated 17 different guidelines for collective communication operations for several MPI libraries such as MVAPICH and Open MPI. The experimental results reveal that none of the libraries was well adapted to our parallel machines, which might not be surprising. However, by using PGMPI we were able to pinpoint exactly which MPI functions should be tuned and which message sizes should be considered. Thus, PGMPI is a useful tool for MPI developers and system administrators to easily spot tuning potentials.

A Self-consistent Performance Guidelines in PGMPI

The guidelines are formulated for a variable communication volume n, $n \geq 0$ and fixed number of processes p, $p \geq 1$, which is omitted.

Monotony Guideline

$$\texttt{MPI\_}A(n) \preceq \texttt{MPI\_}A(n+k) \quad , \qquad k \geq 0 \tag{GL1}$$

Split-Robustness Guideline

$$\texttt{MPI\_}A(n) \preceq k\,\texttt{MPI\_}A(n/k) \quad , \qquad k \geq 1 \tag{GL2}$$

Pattern Guidelines

$$\texttt{MPI\_Gather}(n) \preceq \texttt{MPI\_Allgather}(n) \tag{GL3}$$
$$\texttt{MPI\_Gather}(n) \preceq \texttt{MPI\_Reduce}(n) \tag{GL4}$$
$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Alltoall}(n) \tag{GL5}$$
$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Allreduce}(n) \tag{GL6}$$
$$\texttt{MPI\_Scatter}(n) \preceq \texttt{MPI\_Bcast}(n) \tag{GL7}$$
$$\texttt{MPI\_Reduce}(n) \preceq \texttt{MPI\_Allreduce}(n) \tag{GL8}$$

$$\texttt{MPI\_Reduce\_scatter}(n) \preceq \texttt{MPI\_Allreduce}(n) \tag{GL9}$$

$$\texttt{MPI\_Bcast}(n) \preceq \texttt{MPI\_Scatter}(n) + \texttt{MPI\_Allgather}(n) \tag{GL10}$$

$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Gather}(n) + \texttt{MPI\_Bcast}(n) \tag{GL11}$$

$$\texttt{MPI\_Allreduce}(n) \preceq \texttt{MPI\_Reduce}(n) + \texttt{MPI\_Bcast}(n) \tag{GL12}$$

$$\texttt{MPI\_Allreduce}(n) \preceq \texttt{MPI\_Reduce\_scatter\_block}(n) + \texttt{MPI\_Allgather}(n) \tag{GL13}$$

$$\texttt{MPI\_Reduce}(n) \preceq \texttt{MPI\_Reduce\_scatter\_block}(n) + \texttt{MPI\_Gather}(n) \tag{GL14}$$

$$\texttt{MPI\_Reduce\_scatter\_block}(n) \preceq \texttt{MPI\_Reduce}(n) + \texttt{MPI\_Scatter}(n) \tag{GL15}$$

$$\texttt{MPI\_Scan}(n) \preceq \texttt{MPI\_Exscan}(n) + \texttt{MPI\_Reduce\_local}(n) \tag{GL16}$$

$$\texttt{MPI\_Reduce\_scatter}(n) \preceq \texttt{MPI\_Reduce}(n) + \texttt{MPI\_Scatterv}(n) \tag{GL17}$$

References

1. Chaarawi, M., Squyres, J.M., Gabriel, E., Feki, S.: A tool for optimizing runtime parameters of Open MPI. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 210–217. Springer, Heidelberg (2008)
2. Chan, E., Heimlich, M., Purkayastha, A., van de Geijn, R.A.: Collective communication: theory, practice, and experience. Concurrency Comput. Pract. Experience **19**(13), 1749–1783 (2007)
3. Faraj, A., Yuan, X., Lowenthal, D.K.: STAR-MPI: self tuned adaptive routines for MPI collective operations. In: International Conference on Supercomputing (ICS), pp. 199–208. ACM (2006)
4. Gropp, W., Hoefler, T., Thakur, R., Träff, J.L.: Performance expectations and guidelines for MPI derived datatypes. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroPVM/MPI 2011. LNCS, vol. 6960, pp. 150–159. Springer, Heidelberg (2011)
5. Gropp, W.D., Kimpe, D., Ross, R., Thakur, R., Träff, J.L.: Self-consistent MPI-IO performance requirements and expectations. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 167–176. Springer, Heidelberg (2008)
6. Hollander, M., Wolfe, D.A., Chicken, E.: Nonparametric Statistical Methods, 3rd edn. Wiley, Hoboken (2014)
7. Hunold, S., Carpen-Amarie, A.: Reproducible MPI benchmarking is still not as easy as you think. IEEE TPDS (2016)
8. Hunold, S., Carpen-Amarie, A., Lübbe, F.D., Träff, J.L.: PGMPI: automatically verifying self-consistent MPI performance guidelines. CoRR abs/1606.00215 (2016)
9. Patarasuk, P., Yuan, X.: Bandwidth optimal all-reduce algorithms for clusters of workstations. JPDC **69**(2), 117–124 (2009)
10. Pellegrini, S., Wang, J., Fahringer, T., Moritsch, H.: Optimizing MPI runtime parameter settings by using machine learning. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) EuroPVM/MPI 2009. LNCS, vol. 5759, pp. 196–206. Springer, Heidelberg (2009)

11. Pjesivac-Grbovic, J., Angskun, T., Bosilca, G., Fagg, G.E., Gabriel, E., Dongarra, J.: Performance analysis of MPI collective operations. Cluster Comput. **10**(2), 127–143 (2007)
12. Reussner, R., Sanders, P., Träff, J.L.: SKaMPI: a comprehensive benchmark for public benchmarking of MPI. Sci. Program. **10**(1), 55–65 (2002)
13. Shudler, S., Calotoiu, A., Hoefler, T., Strube, A., Wolf, F.: Exascaling your library: will your implementation meet your expectations? In: International Conference on Supercomputing (ICS), pp. 165–175 (2015)
14. Träff, J.L.: mpicroscope: towards an MPI benchmark tool for performance guideline verification. In: Träff, J.L., Benkner, S., Dongarra, J.J. (eds.) EuroPVM/MPI 2012. LNCS, vol. 7490, pp. 100–109. Springer, Heidelberg (2012)
15. Träff, J.L., Gropp, W.D., Thakur, R.: Self-consistent MPI performance guidelines. IEEE TPDS **21**(5), 698–709 (2010)