Parallel Computing 129 (2026) 103205

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Tuned your MPI library? Now check the performance guidelines

Sascha Hunold [a,*], Jesper Larsson Träff [a], Ruben Laso [b]

[a] Faculty of Informatics, TU Wien, Treitlstraße 3, Vienna, 1040, Austria
[b] Faculty of Computer Science, University of Vienna, Währinger Straße 29, Vienna, 1090, Austria

ARTICLE INFO

Keywords:
MPI
Collective communication
Performance guidelines
Benchmarks

ABSTRACT

The MPI standard provides the foundational building blocks for most parallel applications running on large-scale HPC systems. Collective communication operations in MPI are critical components for the scalability of these applications. Most MPI libraries offer several algorithms for each specific collective operation, and each library selects the algorithm to be used based on the number of processes, the message size, and possibly other factors. Each collective algorithm may perform better in certain scenarios, and thus, selecting the most suitable algorithm for each use case is essential. However, even the best algorithm in a given MPI library may deliver suboptimal performance.

Self-consistent MPI performance guidelines capture semantic relationships between different collective operations and exploit these to express performance expectations that collectives should reasonably satisfy to be considered performance-consistent. For collective communication, such *performance guidelines* typically state that a specialized collective call should not be slower than less specialized counterparts.

In this article, we demonstrate how the consistency of MPI libraries with respect to performance guidelines can be analyzed. For this purpose, we present a tool that checks guideline compliance. For regular collective operations such as `MPI_Bcast`, the tool contains multiple emulated versions of the collective by composing less specialized operations. Then, for a specific number of processes and message sizes, the tool experimentally assesses whether the algorithm selected by the MPI library is slower than its emulated counterparts. If that is the case, a performance-guideline violation is detected. In a broader empirical study, we assess the current state of performance consistency in MPI libraries on modern supercomputers.

1. Introduction

The Message Passing Interface (MPI) is a fundamental building block for developing scalable, parallel applications for today's supercomputers [1]. The MPI standard [2] defines a large and versatile collection of operations for communication between processes in distributed systems. For example, the standard defines point-to-point operations, which can come in different flavors, such as blocking or non-blocking variants.

An important set of MPI operations is the set of collective communication operations. These MPI collectives are patterns for commonly used data transfers within groups of MPI processes, such as a broadcast (`MPI_Bcast`). The MPI standard clearly specifies the input–output semantics of each collective communication operation. However, their implementations are not prescribed and are determined entirely by the respective MPI libraries. MPI collectives can typically be realized by (many) different algorithms, each of which may be advantageous in different communication scenarios [3].

From an MPI library's perspective, it is important to select the best performing algorithm for a given collective operation and set of parameters, which includes the message size and the number of processes. Therefore, an MPI library should be tuned for a given platform in order to select the most suitable algorithm for each collective operation and scenario. However, even if an MPI library is well tuned and selects an appropriate algorithm for a specific collective operation and scenario, the selected algorithm may still perform worse than expected in comparison to other, related operations. In the case of such performance inconsistencies between semantically related operations, the collective call may violate one or more suitably formulated *performance guidelines*. Such guidelines can be formulated based on general performance considerations for the MPI collectives. A straightforward performance expectation would, for instance, state that a broadcast of 1 kB of data should not be slower than a broadcast of 1 MB of data, assuming that the same number of processes is used: For any collective operation, performance can reasonably be expected to be monotonic in the data size.

* Corresponding author.
E-mail address: sascha.hunold@tuwien.ac.at (S. Hunold).

https://doi.org/10.1016/j.parco.2026.103205
Received 8 June 2025; Received in revised form 20 May 2026; Accepted 1 June 2026
Available online 6 June 2026

In this article, we present our approach, a new tool, to automatically analyze the performance-consistency of blocking MPI collectives as implemented in a specific MPI library based on a set of explicitly formulated performance guidelines. To that end, the tool, which implements a performance-consistency checker, performs a series of micro-benchmark runs and evaluates the recorded measurements in a statistically sound manner. In this approach, the MPI collectives are not only tested against re-implementations using different collective operations (mock-ups), but they are also tested against collective algorithms from external libraries. In a case study, we show that a large number of MPI collectives can be improved significantly, since they fail performance-consistency checks.

In this article, we make the following contributions:

- We summarize a set of performance guidelines that define expected relationships between MPI collective operations.
- We present `PGchecker`, a tool for the automated verification of performance consistency in MPI collective implementations.
- We report experimental results from multiple large-scale HPC systems, demonstrating that the set of collective algorithms available in current MPI libraries is often insufficient. Our findings further highlight the need for systematic performance tuning in MPI libraries.

In the remainder of the article, we first examine performance guidelines in the context of MPI in Section 2, followed by a summary of related work in Section 3. We then describe our methodology for verifying these guidelines and introduce a tool that implements them in Section 4. Experimental results on the performance consistency of MPI libraries across different supercomputers are presented in Section 5, and we conclude in Section 6.

2. Definitions

Self-consistent MPI performance guidelines were formally introduced by Träff et al. [4]. For two MPI operations A and B, where A is a specific, more specialized operation defined in the MPI standard, and B an operation or a sequence of operations that re-implements the semantics of A, the following performance expectation should hold:

$$\mathrm{MPI}_A \preceq \mathrm{MPI}_B\,. \tag{1}$$

The relation $\preceq$ should be read as *not-slower-than*. It exploits the semantic relationship that B re-implements A – in the specific setting – to pose the reasonable expectation that A in the concrete call will perform no worse than calling B. Träff et al. [4] defined several meta-rules for MPI functions that must be fulfilled, e.g., monotonicity in data size – a call with a larger problem size is expected not to be faster than a call with a smaller problem size – and that subdividing a message into several smaller messages should not reduce the communication time, under the condition that the functions are called in the same circumstances, e.g., process mapping. For simplicity, the relation notation $\mathrm{MPI}_A \preceq \mathrm{MPI}_B$ omits the data size n and the number of processes p. However, it is generally assumed that the data size and number of processes and the overall context match on both sides, everything else being the same.

Träff et al. [4] stated performance guidelines for various types of MPI functionality, e.g., for point-to-point communication including non-blocking semantics. Blocking collectives, like `MPI_Bcast`, are expected to perform better than misusing a non-blocking counterpart, like `MPI_Ibcast`, to emulate the blocking behavior with an immediately following `MPI_Wait`, e.g.,

$$\texttt{MPI\_Bcast}(n) \preceq \texttt{MPI\_Ibcast}(n) + \texttt{MPI\_Wait}\,. \tag{2}$$

Träff et al. [4] also gave guidelines for one-sided communication, irregular collectives, and communicators. For instance, any regular MPI collective, like `MPI_Alltoall`, is expected to perform better (no worse) than implementing the same functionality with a corresponding irregular (vector) version of the collective, like `MPI_Alltoallv` or `MPI_Alltoallw`. The $\preceq$ relation is meant to be transitive, e.g.,

$$\texttt{MPI\_Alltoall}(n) \preceq \texttt{MPI\_Alltoallv}(n)\,, \tag{3}$$

$$\texttt{MPI\_Alltoallv}(n) \preceq \texttt{MPI\_Alltoallw}(n)\,, \tag{4}$$

which implies that

$$\texttt{MPI\_Alltoall}(n) \preceq \texttt{MPI\_Alltoallw}(n)\,. \tag{5}$$

In the context of the present article, we solely focus on regular, blocking collective communication operations, and therefore we redefine the *not-slower-than* rule as

$$\mathrm{MPI}_A(n,p) \preceq R(A(n,p))\,, \tag{6}$$

which means that a specialized MPI function A inside a specific MPI library should not be slower than a re-implementation R of A for a similar, or the same, data size n and the same number of processes p. By "similar" data size, we mean that in the original MPI function A, n data items are communicated. The same n data items also need to be properly delivered to their destinations by R. However, R may use additional storage or take extra communication rounds.

Let us consider an example guideline $\texttt{MPI\_Scatter}(n) \preceq \texttt{MPI\_Bcast}(n)$, which states that scattering a communication volume of n blocks should not be slower than broadcasting the same vector of n blocks. The general idea of this guideline is depicted in Fig. 1. On the left-hand side of this figure, we see the semantics of the specialized scatter function. On the right-hand side, the re-implementation of scatter using broadcast is shown, where the entire data block is sent to all processes in the communicator. After each process receives the vector via broadcast, it selects the specific block it would have received via scatter. Clearly, and regardless of the additional memory overhead, we can expect the runtime of the re-implementation of scatter by `MPI_Bcast` not to be faster than the library's implementation of `MPI_Scatter`. If that would not be the case, there is obviously something wrong with the library's implementation of `MPI_Scatter`, since it could easily be replaced by its re-implementation by `MPI_Bcast`. In this sense, we expect consistent performance between `MPI_Scatter` and `MPI_Bcast`, and could even require that a good MPI library must fulfill such an expectation.

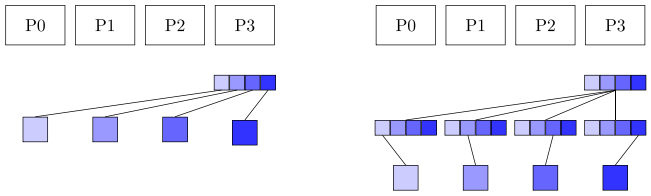


Fig. 1. Illustration of guideline $\texttt{MPI\_Scatter}(n) \preceq \texttt{MPI\_Bcast}(n)$, where the right-hand side re-implements the functionality of `MPI_Scatter`.

3. Related work

Performance guidelines collect and formalize typical expectations that library developers associate with MPI operations. Frequently, the semantics of a specific MPI collective can be reproduced using other, more general MPI collectives. The reasonable expectation is that such a less specialized re-implementation should not outperform the standardized, specialized function, as this would render the specialized version redundant. Such arguments were originally introduced in the context of MPI by Träff et al. [4], who also formulated a large selection of meta- and concrete guidelines for core parts of MPI. This has been extended with additional guidelines for collective operations [5], for MPI I/O [6], for derived datatypes [7,8], and for neighborhood collectives, including communicator creation functionalities [9]. Performance guidelines that

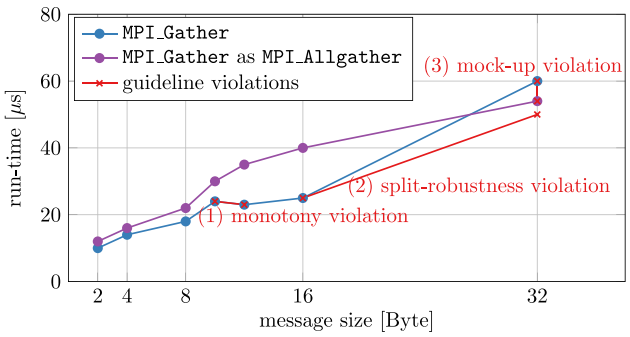


Fig. 2. Three possible guideline violations for `MPI_Gather`: (1) Monotony, i.e., larger messages should not be communicated faster. (2) The red cross at 32 B indicates a split-robustness violation, as the communication in several smaller blocks should not be faster than in a single larger block. (3) A less specialized function should not be faster than the original MPI function, e.g., re-implementing Gather using Allgather should not be faster than the default Gather.

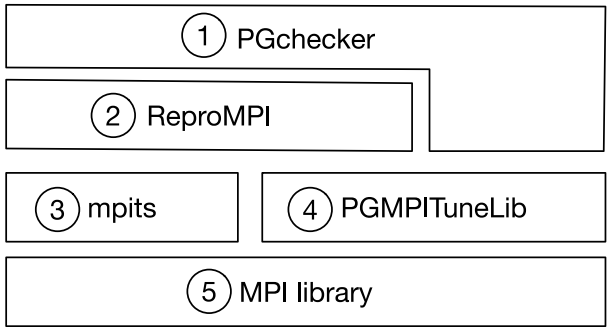


Fig. 3. Layered software architecture of `PGchecker`. `PGchecker` compares whether the collective algorithm selected by the MPI library is slower than alternative implementations provided by PGMPITuneLib. In order to measure the runtime of collectives accurately, ReproMPI can utilize clock synchronization algorithms from mpits.

relate blocking to non-blocking collectives can likewise be formulated and could give insights into the relative quality of implementation. For persistent collectives, this is less obvious, since these collectives allow for extensive optimizations at run-time that make it difficult to relate their performance to either blocking or non-blocking collectives.

A semi-automatic verification of performance guidelines for MPI collectives was initially conducted by Hunold et al. [10], employing a multi-step approach to assess the performance consistency of MPI libraries. In this work, the authors executed a series of runtime experiments on both the original MPI functions and their re-implementations, followed by statistical analysis in a second step.

This original approach verified three types of guideline violations for MPI collectives, all schematically depicted in Fig. 2. The first is a monotony violation, which states that performing a collective on a larger data size should not result in a shorter runtime. The second type concerns split robustness, which requires that dividing the data into smaller blocks and executing multiple collective operations should not outperform a single operation on the full block. An example of such a violation is shown in Fig. 2, where two `MPI_Gather` calls with 16 B each would be faster than a single call with 32 B. The third is the so-called mock-up violation, which states that a re-implementation of a specialized MPI function using more general MPI functionality should not be faster than the original. In the figure, this guideline is violated by `MPI_Gather`, as its mock-up version performs faster for 32 B.

An early benchmarking tool for comparing different MPI implementations of the same (collective) operation was developed by Träff [11]. It included a wide range of semantic reductions from specialized collective operations to more general ones, enabling simple, visual comparison of their performance. This approach facilitated the detection of violations of expected performance relationships between collectives.

The work proposed in the present article is an extension of the previous work by Hunold et al. [10]. In contrast to the earlier study, we now employ a one-shot tool for automated and efficient performance-guideline analysis. Additionally, we aim to assess performance guidelines at a large scale, i.e., utilizing several thousands of processes.

Hunold and Carpen-Amarie [12] also presented a tuning approach based on these performance guidelines, where the so-called mock-ups are used as library replacements if MPI libraries do not provide efficient algorithms for specific MPI functions. They observed that `MPI_Bcast` was often extensively tuned on many modern supercomputers or even had direct hardware support. In such cases, a mock-up implementation – for example, `MPI_Scatter` – could benefit significantly from being implemented using `MPI_Bcast`, despite requiring a larger communication volume. If such a situation were detected, the tuning library intercepted the original MPI call and replaced it with the faster mock-up implementation.

Parker et al. [13] examined the performance of the Cray MPI library on a Cray XC40 system. In their study, they also investigated performance-guideline violations for various MPI collective operations. The authors reported several performance-guideline violations for `MPI_Allgather`, especially for message sizes between 4 kB and 512 kB.

Shudler et al. [14] complemented the performance-guideline idea by assessing whether the scalability of MPI collectives measured in experiments matches their theoretical runtime complexity. In this work, the number of processes is varied, and the runtime behavior is compared to expected complexity classes, e.g., logarithmic or linear scaling. The message size is kept constant in these experiments. This methodology is orthogonal to performance guidelines, which vary the message size while keeping the number of processes constant. Therefore, both approaches could be combined to obtain a more comprehensive performance assessment of MPI collectives.

4. Performance-guideline compliance checker

We now introduce our guideline verification tool, called `PGchecker`.[1] The tool `PGchecker` is an extension of the work proposed by Hunold et al. [10]. The primary limitation of the previous approach was its restricted applicability and limited feature set. The current tool automates all necessary steps: it runs a set of experiments, collects performance data, analyzes the results in a statistically sound manner, and can be extended to compare MPI library functions against alternative implementations of a given collective.

4.1. *`PGchecker`: Tool composition*

The layers of the software architecture of `PGchecker` are shown in Fig. 3. ① `PGchecker` orchestrates a series of micro-benchmarking runs, where the benchmarks are executed by ② ReproMPI. To determine which mock-up implementations are available, `PGchecker` queries the ③ PGMPITuneLib library. ReproMPI measures the runtime of a given collective. To avoid timing bias introduced by `MPI_Barrier`, especially for short-lived collectives, it can utilize the ④ mpits library, which provides several clock synchronization algorithms. At the bottom layer is the selected ⑤ MPI library, such as Open MPI [15], MPICH [16], or MVAPICH [17].

In the following, we briefly outline the key building blocks of `PGchecker`.

[1] The tool `PGchecker` is a component of the ReproMPI repository, available at https://github.com/hunsa/reprompi.

ReproMPI. The ReproMPI benchmark [18] was specifically designed to obtain reproducible results when measuring the running time of blocking collective operations. It distinguishes itself from other MPI micro-benchmarks in two features. First, it can utilize clock synchronization algorithms to obtain accurate logical, global clocks. These global clocks are essential for synchronizing processes in time instead of synchronizing processes using an `MPI_Barrier`. Second, it can measure the runtime of a collective for a predefined number of repetitions or for a maximum amount of time. This way, we can ensure that a very slow collective implementation will not delay the guideline checking process.

PGMPITuneLib. The PGMPITuneLib library provides a collection of mock-up implementations for various blocking collectives [12], which are listed in Appendix. It leverages the PMPI interface to intercept MPI calls and redirect them to alternative implementations. In our setting, for example, the ReproMPI benchmark invokes `MPI_Bcast` to measure its runtime, which may then be redirected to a mock-up version via PGMPITuneLib.

The library is bundled with a large number of re-implementations of specialized MPI collectives. For example, it provides *mock-up* implementation of the following guidelines:

$$\texttt{MPI\_Bcast}(n) \preceq \texttt{MPI\_Allgatherv}(n), \tag{7}$$

$$\texttt{MPI\_Bcast}(n) \preceq \texttt{MPI\_Bcast}_{\text{lane}}(n), \tag{8}$$

$$\texttt{MPI\_Bcast}(n) \preceq \texttt{MPI\_Bcast}_{\text{hier}}(n). \tag{9}$$

In the mock-up implementation of Guideline (7), the functionality of a broadcast operation is emulated using `MPI_Allgatherv`. Here, only the designated root process provides input data, while all other processes contribute zero-length messages. This allows the root's data to be distributed to all processes, effectively mimicking a broadcast.

PGMPITuneLib offers an extensible interface for integrating additional collective implementations. One such extension is the lane collectives [19], which leverage multiple network interfaces through virtual communication lanes. In the experiments presented in this article, we include the lane collectives as additional guideline implementations. For a given MPI library, the runtime of each blocking collective is compared against both the mock-up implementations and the lane collectives.

Two examples are Guidelines (8) and (9), where the algorithm selected by the MPI library is evaluated against the algorithms proposed by Träff and Hunold [19]. As noted earlier, PGMPITuneLib can be easily extended and configured to support other algorithms, for instance, it can be built with support for the circulant collectives [20].

mpits. The mpits library provides several clock synchronization algorithms [21] that mitigate the measurement bias introduced by `MPI_Barrier`, as demonstrated by Schuchart et al. [22]. Eliminating this bias is particularly important for small message sizes, since such collectives are highly sensitive to distorted process arrival patterns induced by `MPI_Barrier` [23].

4.2. Workflow for guideline verification

The `PGchecker` workflow proceeds as follows:

1. The user specifies which blocking collectives should be verified against performance guidelines. For each selected collective, a set of relevant message sizes must be defined.
2. The user may configure benchmarking parameters for each collective. This includes specifying the duration of each test, e.g., 3 s, and the synchronization mechanism used between iterations, like global clock synchronization or `MPI_Barrier`.
3. `PGchecker` queries PGMPITuneLib to determine the number of available mock-up implementations or algorithmic variants for each collective.
4. It then executes a series of ReproMPI runs to measure the runtime of both the default MPI collectives and their counterparts provided by PGMPITuneLib.
5. Once results are collected, `PGchecker` applies a nonparametric statistical test to determine whether any alternative implementation performs significantly better than the default MPI call. If the test rejects the null hypothesis in favor of an alternative, the case is reported as a guideline violation.

4.3. Nonparametric tests

When developing a tool for automated verification of performance guidelines, a fundamental question arises: when is a guideline considered violated? This question is non-trivial, particularly when experiments are conducted on user-shared supercomputers, where variability in performance is inherent. Therefore, a critical prerequisite for fair testing is that all runtime measurements are performed within the same node allocation as determined by the batch scheduler. The fundamental assumption is that the performance characteristics of the system remain stable during the entire measurement period, which may not always be true in practice. Therefore, one should not draw conclusions based on a single experiment. We suggest performing multiple repetitions of each experiment to increase the statistical power of the analysis. For instance, in our experiments, we perform at least three repetitions of the entire experimental campaign. Each individual measurement of a campaign is repeated multiple times until a certain time limit is reached. For small message sizes, the number of repetitions can be in the order of thousands, while for large message sizes, it can be as low as two to five repetitions.

To avoid the need for selecting a fixed performance threshold, `PGchecker` relies on statistical hypothesis testing. We integrate several statistical tests to assess whether one distribution systematically yields larger or smaller runtimes than another. Specifically, we include the Mann-Whitney test (Wilcoxon rank-sum), which is nonparametric and suitable for data that may not follow a normal distribution [24]. Additionally, we support the t-test, which assumes normality but is often robust in practice and commonly used when comparing mean values.

Users can select the statistical test to apply. By default, `PGchecker` employs the Mann-Whitney test. This nonparametric test is typically robust against outliers and does not assume a specific distribution of the data. In particular, the Mann-Whitney test is well-suited for our application, as it can also be applied for small sample sizes [25]. Of course, statistical tests may yield false positives or false negatives, especially when sample sizes are small. The users of `PGchecker` should be aware of these limitations and interpret the results accordingly. For that reason, the *slowdown* is reported in two cases: (1) when the statistical test rejects the null hypothesis, and (2) when the slowdown (ratio of medians) exceeds a user-defined threshold (e.g., 1.0), even if the statistical test does not reject the null hypothesis. That means, `PGchecker` reports potential performance issues even in cases where statistical power is limited. The users then need to decide whether further investigation is necessary.

4.4. `PGchecker`: Use case

Before presenting our experimental study, we illustrate a typical use case of `PGchecker` in Fig. 4. In this example, the performance of `MPI_Bcast` from Open MPI is compared against mock-up implementations and external algorithms, specifically the lane collective from [19]. The guideline verification was executed with 32×64 processes on *VSC-5*, using message sizes ranging from 8 B to 8 000 000 B.

The figure presents a representative output of `PGchecker`, where the runtime of the default collective implementation is contrasted with potentially faster alternatives. In the depicted scenario, the default implementation violates the performance guideline starting at message

```
MPI Collective: MPI_Bcast
collective  count    N   ppn  n     default_median  default_mean    slowdown  stat_violation  fastest_mockup          mockup_median  mockup_mean
MPI_Bcast   8        32  128  5000  0.029240        0.038328
MPI_Bcast   80       32  128  5000  0.030609        0.057201
MPI_Bcast   800      32  128  5000  0.038260        0.053909
MPI_Bcast   8000     32  128  5000  0.057780        0.082498
MPI_Bcast   80000    32  128  5000  0.331367        0.367257
MPI_Bcast   800000   32  128  199   13.940238       14.895282       4.896013  True            bcast_as_bcast_hier     2.847263       2.981668
MPI_Bcast   8000000  32  128  24    113.927973      125.989121      6.526137  True            bcast_as_bcast_lane     17.457185      18.059822
```

Fig. 4. Example output of `PGchecker` for `MPI_Bcast` with 32×64 processes on *VSC-5*. The `count` column indicates the number of elements of type `MPI_BYTE` used in each call. For instance, a count of 8 corresponds to 8 B.

sizes of 800 000 B. In these cases, algorithms from the lane collective library outperformed the default, and a corresponding slowdown was reported. Here, *slowdown* is defined as the ratio of the median runtime of the default algorithm to that of a mock-up implementation.

The column *stat_violation* indicates whether a statistical test has rejected the null hypothesis. For the largest message sizes, i.e., 800 000 B and 8 000 000 B, the statistical test detects a significant difference between the runtime distributions. For example, at 800 000 B, the hierarchical Broadcast implementation (`bcast_as_bcast_hier`) from the lane collective library is nearly 5× faster than the default algorithm selected by the MPI library.

4.5. *PGchecker is not a tuning tool*

Although `PGchecker` is built upon PGMPITuneLib, which provides alternative implementations of MPI collectives, it is important to clarify that `PGchecker` itself is not a tuning tool. Instead, it serves as a performance-guideline compliance checker. Its primary purpose is to identify potential performance issues in MPI libraries by comparing the default collective implementations against alternative algorithms.

There are two main use cases for `PGchecker`: (1) identifying performance-guideline violations in MPI libraries for system administrators and library developers, and (2) providing insights into the performance characteristics of MPI collectives when tuning a given MPI library.

In the first use case, we see its utility for library developers when being integrated into their development workflow as part of a continuous integration (CI) system. In such a setting, `PGchecker` can be used to automatically verify that new commits or changes to the MPI library do not introduce performance regressions or violate established performance guidelines.

In the second use case, `PGchecker` can assist performance engineers in tuning MPI libraries for specific systems or applications. By identifying guideline violations, performance engineers can focus their efforts on optimizing specific collective operations that exhibit suboptimal performance. An example of determining optimization opportunities for the `MPI_Allreduce` collective is presented in Section 5.4.

5. Experimental evaluation

We now present our experimental results obtained with `PGchecker` on three supercomputers.

5.1. Experimental setup

We begin by introducing the hardware and test setup used to verify performance guidelines on the three systems.

5.1.1. Hardware setup

The hardware configurations of the parallel machines are summarized in Table 1. The systems include *LUMI* in Finland, *Leonardo* in Italy, and *VSC-5* in Austria. Although the machines differ in interconnects and processor architectures, we expect similar patterns of guideline violations between *Leonardo* and *VSC-5*, as both employ the same version of Open MPI.

5.1.2. Tested collectives and data sizes

In this work, we focus on verifying performance guidelines for blocking collective operations with fixed data sizes. We compare the default implementations provided by MPI libraries against various mock-up alternatives and external implementations, in particular hierarchical and lane collectives [19].

Our evaluation includes all blocking collectives listed in Appendix, with the exception of `MPI_Alltoall`. We observed that experimental runs involving Alltoall significantly distorted the runtimes of subsequent collectives. This artifact was consistently reproducible across multiple machines, which led us to exclude Alltoall from the measurements to preserve data integrity.

We do not include vector variants of blocking collectives, such as `MPI_Gatherv` or `MPI_Allgatherv`, in our experiments. This decision is motivated by the combinatorial explosion of possible parameter configurations, which complicates the systematic evaluation of potential guideline violations.

Although a similar argument could be made regarding the choice of message sizes, we selected a representative subset to evaluate MPI libraries' conformance to performance guidelines.

All collectives were benchmarked with message sizes of 8, 80, 800 and 8000 B. This progression starts with a payload large enough to hold at least one `double`, then increases the size by an order of magnitude while avoiding power-of-two patterns. For gathering collectives such as `MPI_Gather`, `MPI_Allgather`, and `MPI_Alltoall`, the defined size refers to the data each process contributes. For instance, in an `MPI_Allgather` call with 8000 B per process and a communicator of 64 × 128 processes, each process receives 64 × 128 × 8000 B, which corresponds to 62 MB.

For fixed-size collectives like `MPI_Bcast`, we additionally tested with 80 000, 800 000 and 8 000 000 B.

We emphasize that the presented configuration represents a snapshot. `PGchecker` can be configured to evaluate any message sizes the experimenter deems relevant.

5.1.3. Measurement setup

Precisely measuring the runtime of collective MPI operations is inherently challenging [18]. On multi-user supercomputers, runtimes are heavily influenced by the node allocation and the current system load. When node allocations span multiple racks – or even different network islands – the performance of collective operations can vary significantly.

A further challenge, discussed previously, is how to synchronize processes between measurements. In our setup, processes are synchronized either via a global clock or using `MPI_Barrier`. Each process starts a local timer before the collective call and records the elapsed time upon completion. The runtime of the collective is defined as the maximum time observed among all participating processes. If processes are well synchronized in time, this method yields a highly accurate measurement. However, `MPI_Barrier` does not ensure synchronized return times, leading to non-uniform arrival patterns that can distort measurements [22]. To mitigate this effect, we synchronize clocks before each collective call and message size, but only for small messages (e.g., 8192 B or less). For larger messages, the execution time of the collective is typically long enough that minor variations in process arrival times do not significantly affect the results.

Table 1
Overview of hardware setup and MPI libraries used in the experimental evaluation.

Machine	Country	Compute nodes	Interconnect	MPI libraries
LUMI	Finland	2048 nodes (HPE Cray EX) 2 × 64-core AMD EPYC 7763	HPE Cray Slingshot-11 (Dragonfly)	Cray MPICH 8.1.29
Leonardo	Italy	1536 nodes (BullSequana X2140) 2 × 56-core Intel Sapphire Rapids	InfiniBand HDR (Dragonfly+)	Open MPI 4.1.6
VSC-5	Austria	704 nodes (MEGWARE SlideSX-LC) 2 × 64-core AMD EPYC 7713	InfiniBand HDR (Fat-Tree)	Open MPI 4.1.6

ReproMPI supports fixed time-interval measurements, enabling predictable and uniform benchmarking. Each collective is executed repeatedly, up to a maximum of 5000 repetitions or 3 s, whichever occurs first. The resulting samples are then analyzed using statistical methods, as discussed earlier.

Since performance measurements can depend heavily on timing and node placement, a single verification run may not be representative or reproducible. To improve reliability, we conduct each verification experiment at least three times per machine, each in a separate batch job. In most cases, this results in different node allocations. Moreover, the performance of a single collective can be strongly affected by external workload, potentially leading to misleading conclusions. Executing multiple batch jobs provides a broader view of the MPI library's behavior under different system conditions.

Considering all these factors, we emphasize that our results represent only a snapshot of potential performance behavior. The purpose of this work is to provide users with a tool that enables them to verify MPI performance guidelines in their own environments and under their own conditions.

5.1.4. Data analysis and presentation

We recorded samples for N different node allocations. For each node allocation N_i, we compute the median runtime of the MPI collective, denoted as η_L, and the median runtimes of each of the j mock-up implementations and algorithmic variants, denoted as η_{M_j}. The best-performing competing implementation is defined as

$$\eta_M = \min_j \eta_{M_j} . \tag{10}$$

We then define the overhead of the MPI function as

$$o = \max \left\{ 0.0, \frac{\eta_L}{\eta_M} - 1 \right\} . \tag{11}$$

This metric o differs slightly from the *slowdown* reported in the `PGchecker` output shown in Fig. 4, which represents the direct ratio of the two medians. The value o captures only the relative overhead of the MPI library function compared to the best available competitor.

In all plots, we express o as a percentage. For example, if $o = 0.3$, then the MPI library function was 30 % slower than the best competing implementation.

To provide a concise overview of the MPI library's performance, we classify the value of o into four categories, which are color-coded in the plots:

- low, $o \leq 0.1$,
- medium, $0.1 < o \leq 0.5$,
- high, $0.5 < o \leq 2.0$,
- very high, $o > 2.0$.

5.2. Experimental results: Overview of guideline violations

We now present a bird's-eye view of the results for each machine. Note that `MPI_Alltoall` is excluded due to the measurement artifacts discussed in Section 5.1.2.

5.2.1. LUMI

We begin with the guideline analysis of Cray MPICH 8.1.29 on the *LUMI* supercomputer. `PGchecker` was executed using three distinct node allocations, each involving 64×128 processes. Fig. 5(a) presents both the best and worst observed overheads across the three batch jobs. Consider, for example, the case of `MPI_Scan` with 8000 B: in the best-case plot (left), the overhead is 9 %, indicating low deviation. In the worst-case plot, however, the overhead increases to 11 %, which remains modest.

The performance guideline verification on *LUMI* yields a generally positive outcome. Most MPI functions exhibit at most a *medium* overhead (i.e., ≤ 50 %). Only `MPI_Reduce` and `MPI_Reduce_scatter_block` consistently failed to comply with the performance guidelines. For instance, the reference implementation of `MPI_Reduce_scatter_block` with 80 B incurred more than 500 % overhead, even in the best case. These violations suggest potential for tuning, e.g., by selecting a different algorithm for `MPI_Reduce_scatter_block`.

5.2.2. Leonardo

Fig. 5(b) presents the best and worst observed overheads for different collectives using 16×112 processes on *Leonardo*. The results differ markedly from those on *LUMI*. Even in the best-case scenario (left), many red cells appear, indicating overheads exceeding 500 %. Two collectives in particular – `MPI_Reduce` and `MPI_Scan` – stand out. These are provided by the Open MPI library. While these collectives are not among the most frequently used [26], such as `MPI_Allreduce` or `MPI_Bcast`, their poor performance may still result in significant slowdowns in applications that depend on them at scale.

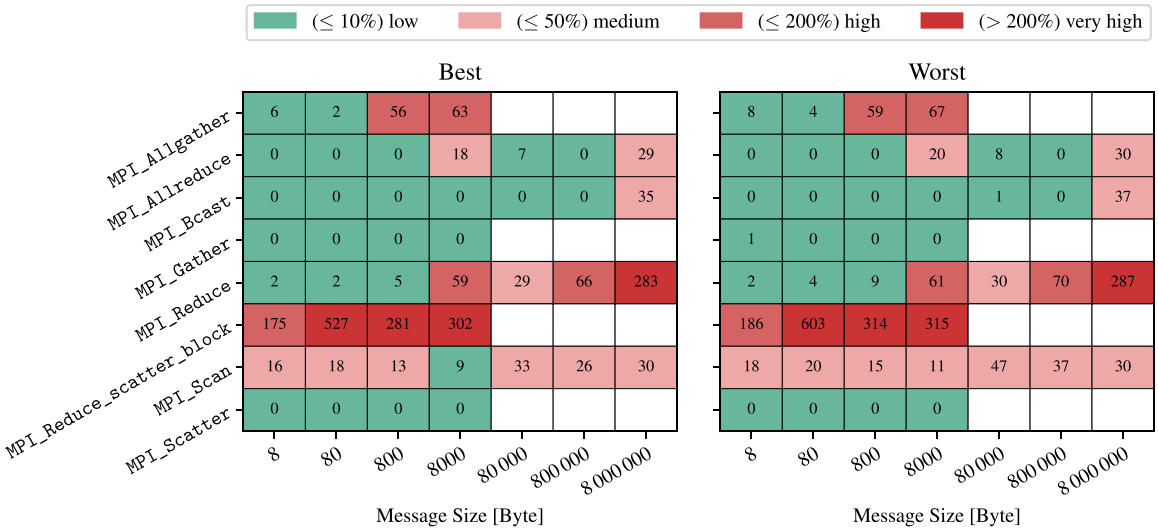
It is also important to note the high variability in runtime observed on *Leonardo*. Since each library and competing collective is evaluated in only three batch jobs, the likelihood that a competing implementation experiences reduced system contention is non-negligible. As a result, performance comparisons may be biased in favor of these alternatives. Although results could vary on a less loaded system, our experiments reflect real-world conditions. The findings suggest that certain library collectives could benefit from additional tuning or alternative algorithmic implementations.

5.2.3. VSC-5

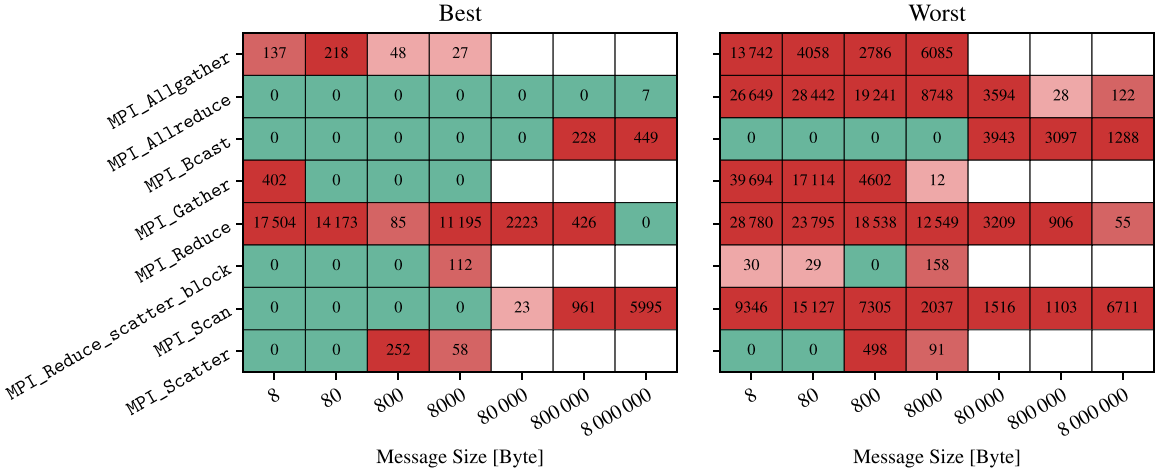
The final overview of guideline violations is shown for *VSC-5*, using 32×128 processes, in Fig. 5(c).

We observe a similar trend to that of *Leonardo*, which is expected since both systems use the same MPI library version, Open MPI 4.1.6. For instance, `MPI_Bcast` with large message sizes performs poorly compared to the competing implementations. The most pronounced violation, however, is found in `MPI_Scan`, which exhibits a consistent 10× slowdown across all cases.

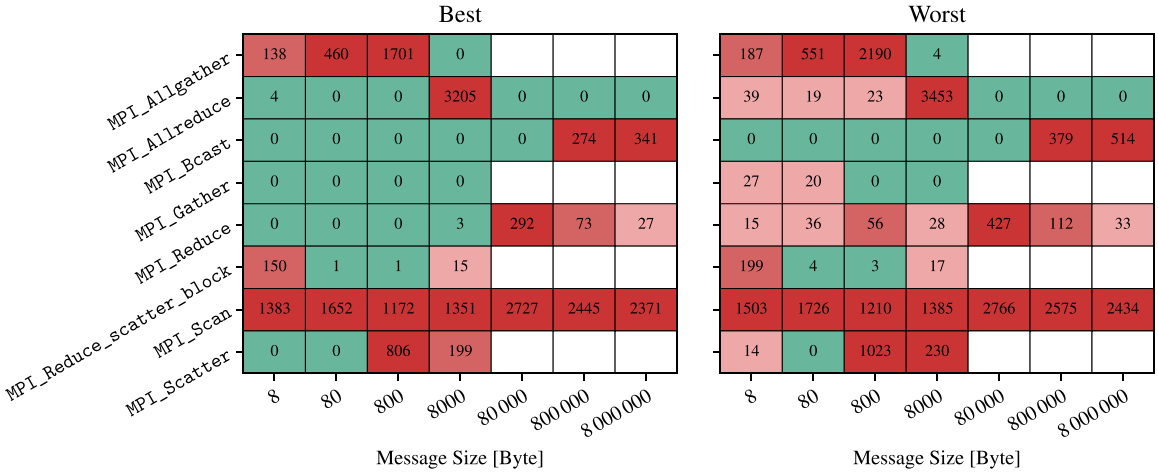
On a positive note, frequently used collectives such as `MPI_Allreduce` and `MPI_Gather` often conform well to the performance guidelines, indicating good default choices for commonly encountered communication patterns.



(a) *LUMI*; 8192 (64 × 128) processes.



(b) *Leonardo*; 1792 (16 × 112) processes.



(c) *VSC-5*; 4096 (32 × 128) processes.

Fig. 5. Guideline overview for the best and the worst execution. The overhead o of the MPI library for each collective and message size is reported as the percentage increase relative to the fastest guideline implementation. The color coding indicates the severity of the overhead, from light red to dark red. Green cells indicate no overhead.

Table 2
Percentage of experiments with *high* or *very high* performance-guideline violations ($o > 0.5$). Lower is better.

MPI call	*LUMI*	*Leonardo*	*VSC-5*
MPI_Allgather	50 %	83 %	75 %
MPI_Allreduce	0 %	43 %	14 %
MPI_Bcast	0 %	38 %	29 %
MPI_Gather	0 %	58 %	0 %
MPI_Reduce	43 %	90 %	33 %
MPI_Reduce_scatter_block	100 %	25 %	25 %
MPI_Scan	0 %	76 %	100 %
MPI_Scatter	0 %	50 %	50 %

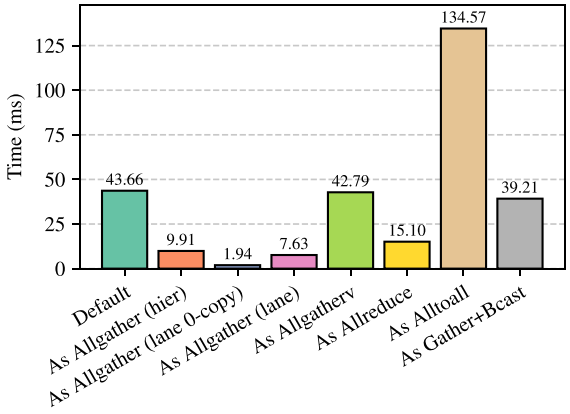


Fig. 6. Median runtimes of `MPI_Allgather` implementations obtained on *VSC-5* with 32×128 processes and a message size of 8000 B.

5.2.4. Summary of empirical guideline violations

In Table 2, we summarize the percentage of experiments with *high* or *very high* performance-guideline violations ($o > 0.5$) for each collective and machine.

Three collectives show consistent performance issues across all machines: `MPI_Allgather`, `MPI_Reduce`, and `MPI_Reduce_scatter_block`. For `MPI_Allgather` and `MPI_Reduce`, the violations are usually triggered because of the better performance of the lane or hierarchical collectives [19]. In the case of `MPI_Reduce_scatter_block`, the MPI library's implementation is often outperformed by the mock-up version based on `MPI_Allreduce`.

Comparing the three machines, *LUMI* exhibits the fewest violations with only three collectives showing issues, most notably `MPI_Reduce_scatter_block` with a 100 % violation rate. In contrast, *Leonardo* and *VSC-5* show widespread violations across most collectives, with *Leonardo* having particularly high rates for `MPI_Allgather` (83 %) and `MPI_Reduce` (90 %). *VSC-5* also shows significant violations, especially for `MPI_Scan` (100 %).

5.3. Comparing algorithmic options

A high-level overview may not provide sufficient insight into the performance behavior of MPI collectives. To address this, `PGchecker` also enables a detailed examination of individual algorithms across different message sizes. Fig. 6 illustrates such an analysis, comparing the median runtimes of various algorithms for `MPI_Allgather` using 32×128 processes and a message size of 8000 B on *VSC-5*.

The results show that the MPI library's implementation of `MPI_Allgather` closely resembles the behavior of a mock-up version based on `MPI_Allgatherv`. Additionally, we observe that the mock-up relying on a highly tuned `MPI_Allreduce` operation achieves a notably lower runtime.

The most efficient algorithms in this setting, however, are those provided by the lane collective library. These findings highlight two important directions: (1) the need to integrate better-performing algorithms into MPI libraries; and (2) the importance of evaluating new collective algorithms against a broad set of alternatives. A comparison limited to the default MPI implementation is insufficient for validating the efficiency of novel collective algorithms.

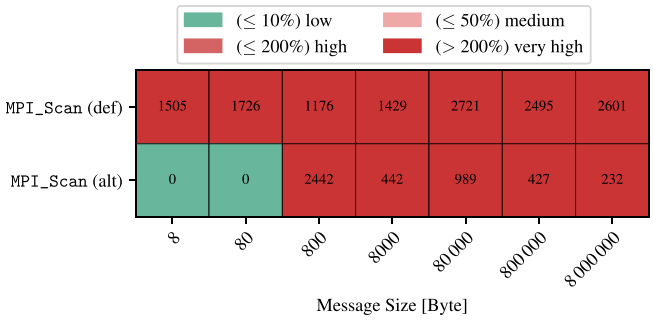


Fig. 7. Performance guideline analysis for `MPI_Scan` on *VSC-5* after adapting the internal algorithm.

5.4. Developer perspective: Pinpointing tuning potential

We aim to improve the performance of `MPI_Scan` on *VSC-5*, which exhibits significant guideline violations, as shown in Fig. 5(c). We emphasize that `PGchecker` solely compares the MPI library's implementation against various guideline-conforming alternatives. The selection of the specific algorithm, however, is performed internally by the MPI library, based on factors such as the number of processes and message size.

Consequently, enhancing the overall performance of an MPI library requires a tuning strategy that complements the guideline-checking methodology. A systematic tuning process – guided by the insights provided by `PGchecker` – can enable the MPI library to make more informed algorithmic choices, thereby improving performance robustness.

An example of such a tuning effort is shown in Fig. 7. By default, Open MPI selects one of two available algorithms for `MPI_Scan` (linear or recursive doubling), depending on the number of processes and message size. In the experiment illustrated in the figure, we modified the configuration to force Open MPI to always use the recursive doubling algorithm. When running `PGchecker` on this alternative configuration, the results are labeled as "`MPI_Scan` (alt)". We observe a notable reduction in overhead relative to the guideline implementations. However, for some message sizes, such as 800 B, the overhead increases, indicating that the recursive doubling algorithm is not optimal in all scenarios.

This analysis highlights two key insights: first, `PGchecker` can be used as a complementary tool to guide and validate MPI tuning efforts. Second, tuning a specific algorithm alone is insufficient. In this case, Open MPI offers only two algorithms for `MPI_Scan`, and both exhibit performance degradations under certain conditions. This indicates that further algorithmic diversity is necessary. Therefore, `PGchecker` should be employed after tuning to verify that the MPI library remains self-consistent and compliant with performance guidelines.

We provide a second example involving the `MPI_Reduce` collective on *Hydra* in Fig. 8, which is a local cluster consisting of 36 nodes (32 cores per node) with dual-rail Intel Omni-Path interconnect. The red line represents the default `MPI_Reduce` performance from Open MPI 4.1.6, for different message sizes and 32×32 processes (note the logarithmic scale). In this case (red), Open MPI selects the algorithm to be used for `MPI_Reduce` based on a predefined internal decision table. We also ran an exhaustive search over all available algorithms and their parameters (e.g., segment sizes for pipelining) in Open MPI,

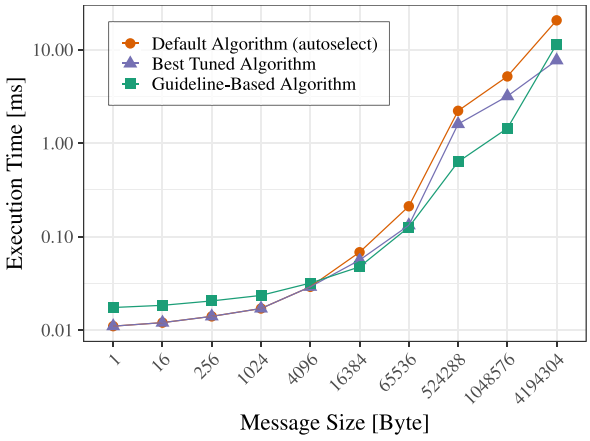


Fig. 8. Comparing the default, tuned, and guideline mock-up implementations of `MPI_Reduce` on *Hydra* with 32×32 processes and Open MPI 4.1.6.

to find the best possible configuration for each message size (blue). We can observe that this tuning effort significantly reduces the runtime of `MPI_Reduce` starting from message sizes of 16 kB. Therefore, the blue line represents the best possible performance achievable with Open MPI on *Hydra*, given the currently available algorithms in this MPI library. Now, we run `PGchecker` on *Hydra* (green) to compare the default `MPI_Reduce` implementation against various guideline-conforming alternatives. The results, shown in green, indicate that even after the tuning effort (blue), there are still significant performance deficits compared to the best guideline-conforming implementation, primarily with message sizes of 512 kB and 1 MB. In such a case, the developers could consider integrating the better-performing guideline-conforming implementation into Open MPI, to further enhance the performance of `MPI_Reduce` on different systems.

5.5. User perspective: PGchecker-guided application tuning

Although `PGchecker` is primarily designed to assist MPI library developers in tuning collective operations, it can also be a valuable tool for application developers seeking to optimize their MPI-based applications. We now present a case demonstrating that guideline mock-up implementations can outperform default MPI library collectives on a specific system. In this scenario, we improved the performance of the Laghos application [27] on *Hydra* using 32×32 processes. Laghos is a high-order finite element application that heavily employs the `MPI_Allreduce` collective with a message size of 8 B. When profiling the application using IPM [28], we found that 51.67 % of the total runtime is spent in `MPI_Allreduce`.

Using `PGchecker`, we identified performance-guideline violations for `MPI_Allreduce` on *Hydra*, i.e., the hierarchical collective implementation outperforms the Open MPI implementation. We exhaustively benchmarked `MPI_Allreduce` with 8 B messages on *Hydra*, comparing the default Open MPI implementation against all available algorithms and their parameter configurations. We found that Open MPI already selects the best-performing algorithm within its implementation for this message size. Despite Open MPI selecting the optimal internal algorithm, the hierarchical mock-up implementation still outperforms it with 8 B.

We used the PGMPITuneLib library [12] to intercept `MPI_Allreduce` calls in Laghos and redirect them to the hierarchical collective implementation. Table 3 shows the average execution times, across five runs, for the various phases of Laghos. In these results, we observed a reduction in overall execution time of 3.35 %, with strong statistical significance ($p = 3.63 \times 10^{-7}$). This showcases how `PGchecker` can guide application developers in identifying and exploiting performance improvements in MPI collectives, even when the MPI library is already well-tuned.

Table 3
Average execution times (in seconds) for the various phases of Laghos. Some phases may overlap, and "Major Kernels" represents the primary figure of merit. Lower is better.

Phase	Default	Best mock-up	Improv. [%]
CG (H1)	53.68	51.62	3.83
CG (L2)	1.87	1.71	8.56
Forces	0.53	0.53	0.00
Update Quad Data	7.00	6.99	0.14
Major Kernels	60.76	58.72	3.35

5.6. Extensibility of PGchecker

The `PGchecker` framework was designed to examine the blocking collective operations defined in the MPI standard, as they are widely used in MPI applications [1] and have well-defined performance guidelines.

However, the framework can be extended to cover additional collective types and communication paradigms, including: non-blocking collectives, persistent collectives, and neighborhood collectives. The main challenge in extending `PGchecker` to these areas lies in defining appropriate performance guidelines, as the semantics and usage patterns differ from those of blocking collectives. For non-blocking collectives, which are also used by the persistent versions, defining meaningful performance guidelines is complex. We can, however, propose similar guidelines for non-blocking collectives as for blocking collectives, such as:

$$\texttt{MPI\_Iallreduce}(n) + \texttt{MPI\_Wait} \preceq \texttt{MPI\_Ireduce}(n) + \texttt{MPI\_Wait} + \texttt{MPI\_Ibcast}(n) + \texttt{MPI\_Wait}. \quad (12)$$

For persistent collectives, such as broadcast, we can state that

$$\texttt{MPI\_Start} + \texttt{MPI\_Wait} \preceq \texttt{MPI\_Ibcast} + \texttt{MPI\_Wait}, \quad (13)$$

where `MPI_Bcast_init` must have been called before, but is not part of the performance guideline. Support for neighborhood collectives is more challenging, as it depends on the given neighborhood definition, which is often very application-specific. Also, guidelines for datatypes are more difficult to define, as there are many degrees of freedom [8].

Nonetheless, the concept implemented by `PGchecker` can also be applied to other communication paradigms beyond MPI, such as CCL (Collective Communication Library) frameworks [29,30] like Nvidia's NCCL or AMD's RCCL. Similar to the MPI performance guidelines, we can define guidelines for NCCL collectives, such as:

$$\texttt{ncclAllReduce}(n) \preceq \texttt{ncclReduce}(n) + \texttt{ncclBcast}(n), \quad (14)$$

$$\texttt{ncclAllGather}(n) \preceq \texttt{ncclAlltoAll}(n). \quad (15)$$

Since these calls are asynchronous by default, in the practical implementation, we would need to add synchronization calls, such as

$$\texttt{ncclAllReduce}(n) + \texttt{cudaStreamSynchronize} \preceq \texttt{ncclReduce}(n) + \texttt{ncclBcast}(n) + \texttt{cudaStreamSynchronize}. \quad (16)$$

6. Conclusions

In this work, we presented `PGchecker`, a tool for the automated verification of performance guidelines for MPI collective operations. `PGchecker` compares the runtime of default MPI implementations against a set of mock-up collectives and external algorithmic variants, including hierarchical and lane-based collectives. By systematically benchmarking under controlled experimental conditions and applying statistical tests, the tool identifies potential violations of performance guidelines. The evaluation on three supercomputers – *LUMI*, *Leonardo*, and *VSC-5* – revealed both well-optimized and poorly tuned collective

operations, highlighting the variability across platforms and implementations. Crucially, PGchecker provides detailed analyses that enable users to isolate inefficient algorithms and detect tuning opportunities.

Our experiments further demonstrate that simply relying on the default MPI implementation is insufficient to ensure performance portability and efficiency. Cases like MPI_Scan on *VSC-5* illustrate that even widely used MPI libraries may lack appropriate algorithmic diversity or tuning. We showed how PGchecker can support MPI library developers and users by serving as a verification layer after tuning, ensuring performance consistency and highlighting areas for algorithmic enhancement. Overall, PGchecker complements tuning tools and provides experimenters with a practical mechanism to assess and improve the performance of MPI collectives in a systematic and reproducible manner.

CRediT authorship contribution statement

Sascha Hunold: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Conceptualization. **Jesper Larsson Träff:** Writing – review & editing. **Ruben Laso:** Writing – review & editing, Visualization, Software.

Acknowledgments

We thank Maximilian Hagn (TU Wien) for helping us to implement PGchecker.

This work was partially supported by the Austrian Science Fund (FWF): project P 33884-N. We acknowledge PRACE and EuroHPC Joint Undertaking for awarding us access to LUMI at CSC, Finland.

We acknowledge the DCGP Austria 2024 project for awarding us access to Leonardo at Cineca, Italy.

Appendix. Overview of performance guidelines

The following performance guidelines are implemented in PGMPITuneLib and are used by PGchecker to verify the performance consistency of MPI collectives. Note that we have also tested the MPI collectives against competing implementations, such as the lane or hierarchical algorithms [19].

$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Gather}(n) + \texttt{MPI\_Bcast}(n), \tag{GL1}$$

$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Alltoall}(n) \tag{GL2}$$

$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Allreduce}(n) \tag{GL3}$$

$$\texttt{MPI\_Allgather}(n) \preceq \texttt{MPI\_Allgatherv}(n) \tag{GL4}$$

$$\texttt{MPI\_Allgatherv}(n) \preceq \texttt{MPI\_Alltoallv}(n) \tag{GL5}$$

$$\texttt{MPI\_Allreduce}(n) \preceq \texttt{MPI\_Reduce}(n) + \texttt{MPI\_Bcast}(n), \tag{GL6}$$

$$\texttt{MPI\_Allreduce}(n) \preceq \texttt{MPI\_Reduce\_scatter\_block}(n) + \texttt{MPI\_Allgather}(n) \tag{GL7}$$

$$\texttt{MPI\_Allreduce}(n) \preceq \texttt{MPI\_Reduce\_scatter}(n) + \texttt{MPI\_Allgatherv}(n) \tag{GL8}$$

$$\texttt{MPI\_Alltoall}(n) \preceq \texttt{MPI\_Alltoallv}(n) \tag{GL9}$$

$$\texttt{MPI\_Bcast}(n) \preceq \texttt{MPI\_Allgatherv}(n) \tag{GL10}$$

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

$$\texttt{MPI\_Gather}(n) \preceq \texttt{MPI\_Allgather}(n) \tag{GL12}$$

$$\texttt{MPI\_Gather}(n) \preceq \texttt{MPI\_Gatherv}(n) \tag{GL13}$$

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

$$\texttt{MPI\_Reduce}(n) \preceq \texttt{MPI\_Allreduce}(n) \tag{GL15}$$

$$\texttt{MPI\_Reduce}(n) \preceq \texttt{MPI\_Reduce\_scatter\_block}(n) + \texttt{MPI\_Gather}(n) \tag{GL16}$$

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

$$\texttt{MPI\_Reduce}(n) \preceq \texttt{MPI\_Reduce\_scatter}(n) \tag{GL18}$$

$$\texttt{MPI\_Reduce\_scatter\_block}(n) \preceq \texttt{MPI\_Reduce}(n) + \texttt{MPI\_Scatter}(n) \tag{GL19}$$

$$\texttt{MPI\_Reduce\_scatter\_block}(n) \preceq \texttt{MPI\_Reduce\_scatter}(n) \tag{GL20}$$

$$\texttt{MPI\_Reduce\_scatter\_block}(n) \preceq \texttt{MPI\_Allreduce}(n) \tag{GL21}$$

$$\texttt{MPI\_Scan}(n) \preceq \texttt{MPI\_Exscan}(n) + \texttt{MPI\_Reduce\_local}(n) \tag{GL22}$$

$$\texttt{MPI\_Scatter}(n) \preceq \texttt{MPI\_Bcast}(n) \tag{GL23}$$

$$\texttt{MPI\_Scatter}(n) \preceq \texttt{MPI\_Scatterv}(n) \tag{GL24}$$

Data availability

Data will be made available on request.

References

[1] N. Sultana, M. Rüfenacht, A. Skjellum, P.V. Bangalore, I. Laguna, K. Mohror, Understanding the use of message passing interface in exascale proxy applications, Concurr. Comput. Pr. Exp. 33 (14) (2021) http://dx.doi.org/10.1002/cpe.5901.

[2] MPI Forum, MPI: A message-passing interface standard. Version 4.1, 2023, www.mpi-forum.org.

[3] E. Chan, M. Heimlich, A. Purkayastha, R.A. van de Geijn, Collective communication: theory, practice, and experience, Concurr. Comput.: Pr. Exp. 19 (13) (2007) 1749–1783, http://dx.doi.org/10.1002/cpe.1206.

[4] J.L. Träff, W.D. Gropp, R. Thakur, Self-consistent MPI performance guidelines, IEEE Trans. Parallel Distrib. Syst. 21 (5) (2010) 698–709, http://dx.doi.org/10.1109/TPDS.2009.120.

[5] J.L. Träff, S. Hunold, I. Vardas, N.M. Funk, Uniform algorithms for reduce-scatter and (most) other collectives for MPI, in: Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER, 2023, pp. 284–294, http://dx.doi.org/10.1109/CLUSTER52292.2023.00031.

[6] W.D. Gropp, D. Kimpe, R. Ross, R. Thakur, J.L. Träff, Self-consistent MPI-IO performance requirements and expectations, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface. 15th European PVM/MPI Users' Group Meeting, in: Lecture Notes in Computer Science, vol. 5205, Springer, 2008, pp. 167–176, http://dx.doi.org/10.1007/978-3-540-87475-1_25.

[7] W.D. Gropp, T. Hoefler, R. Thakur, J.L. Träff, Performance expectations and guidelines for MPI derived datatypes: a first analysis, in: Recent Advances in Message Passing Interface. 18th European MPI Users' Group Meeting, in: Lecture Notes in Computer Science, vol. 6960, Springer, 2011, pp. 150–159, http://dx.doi.org/10.1007/978-3-642-24449-0_18.

[8] A. Carpen-Amarie, S. Hunold, J.L. Träff, On expected and observed communication performance with MPI derived datatypes, Parallel Comput. 69 (2017) 98–117, http://dx.doi.org/10.1016/J.PARCO.2017.08.006.

[9] F.D. Lübbe, Micro-benchmarking MPI neighborhood collective operations, in: 23rd International Conference on Parallel and Distributed Computing (Euro-Par), in: Lecture Notes in Computer Science, vol. 10417, Springer, 2017, pp. 65–78, http://dx.doi.org/10.1007/978-3-319-64203-1_5.

[10] S. Hunold, A. Carpen-Amarie, F.D. Lübbe, J.L. Träff, Automatic verification of self-consistent MPI performance guidelines, in: Euro-Par 2016: Parallel Processing, in: LNCS, vol. 9833, Springer International Publishing, 2016, pp. 433–446, http://dx.doi.org/10.1007/978-3-319-43659-3_32.

[11] J.L. Träff, mpicroscope: Towards an MPI benchmark tool for performance guideline verification, in: Recent Advances in Message Passing Interface. 19th European MPI Users' Group Meeting, in: Lecture Notes in Computer Science, vol. 7490, Springer, 2012, pp. 100–109, http://dx.doi.org/10.1007/978-3-642-33518-1_15.

[12] S. Hunold, A. Carpen-Amarie, Autotuning MPI collectives using performance guidelines, in: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia), ACM, 2018, pp. 64–74, http://dx.doi.org/10.1145/3149457.3149461.

[13] S. Parker, S. Chunduri, K. Harms, K. Kandalla, Performance evaluation of MPI on Cray XC40 Xeon Phi systems, in: Cray UserGroup Proceedings, 2018.

[14] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, F. Wolf, Exascaling your library: Will your implementation meet your expectations? in: Proceedings of the 29th ACM on International Conference on Supercomputing, ACM, 2015, pp. 165–175, http://dx.doi.org/10.1145/2751205.2751216.

[15] R.L. Graham, B. Barrett, G.M. Shipman, T.S. Woodall, G. Bosilca, Open MPI: a high performance, flexible implementation of MPI point-to-point communications, Parallel Process. Lett. 17 (1) (2007) 79–88, http://dx.doi.org/10.1142/S0129626407002880.

[16] W. Gropp, E.L. Lusk, N.E. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Comput. 22 (6) (1996) 789–828, http://dx.doi.org/10.1016/0167-8191(96)00024-5.

[17] D.K. Panda, H. Subramoni, C.-H. Chu, M. Bayatpour, The MVAPICH project: Transforming research into high-performance MPI library for HPC community, J. Comput. Sci. 52 (2021) 101208, http://dx.doi.org/10.1016/j.jocs.2020.101208, Case Studies in Translational Computer Science.

[18] S. Hunold, A. Carpen-Amarie, Reproducible MPI benchmarking is still not as easy as you think, IEEE Trans. Parallel Distrib. Syst. 27 (12) (2016) 3617–3630, http://dx.doi.org/10.1109/TPDS.2016.2539167.

[19] J.L. Träff, S. Hunold, Decomposing MPI collectives for exploiting multi-lane communication, in: Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER, 2020, pp. 270–280, http://dx.doi.org/10.1109/CLUSTER49012.2020.00037.

[20] J.L. Träff, Optimal, non-pipelined reduce-scatter and allreduce algorithms with an application to all-to-all communication, ACM Trans. Parallel Comput. 12 (4) (2025) 1–23, http://dx.doi.org/10.1145/3760789.

[21] S. Hunold, A. Carpen-Amarie, Hierarchical clock synchronization in MPI, in: Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER, 2018, pp. 325–336, http://dx.doi.org/10.1109/CLUSTER.2018.00050.

[22] J. Schuchart, S. Hunold, G. Bosilca, MPI process synchronization in space and time, in: Proceedings of the 30th European MPI Users' Group Meeting, ACM, 2023, pp. 1–11, http://dx.doi.org/10.1145/3615318.3615325.

[23] M. Salimi Beni, B. Cosenza, S. Hunold, MPI collective algorithm selection in the presence of process arrival patterns, in: IEEE International Conference on Cluster Computing, CLUSTER 2024, Kobe, Japan, September 24-27, 2024, IEEE, 2024, pp. 108–119, http://dx.doi.org/10.1109/CLUSTER59578.2024.00017.

[24] D.J. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures, fourth ed., Chapman & Hall/CRC, 2007.

[25] N. Nachar, The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution, Tutorials Quant. Methods Psychol. 4 (1) (2008) 13–20, http://dx.doi.org/10.20982/tqmp.04.1.p013.

[26] S. Chunduri, S. Parker, P. Balaji, K. Harms, K. Kumaran, Characterization of MPI usage on a production supercomputer, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC, IEEE / ACM, 2018, pp. 30:1–30:15, http://dx.doi.org/10.1109/SC.2018.00033.

[27] J.C. Camier, Laghos (LAGrangian High-Order Solver) benchmark summary (CTS2), Tech. Rep. LLNL-TR–770220; 961674, Lawrence Livermore National Laboratory, 2019, http://dx.doi.org/10.2172/1544948.

[28] K. Fürlinger, N.J. Wright, D. Skinner, Performance analysis and workload characterization with IPM, in: M.S. Müller, M.M. Resch, A. Schulz, W.E. Nagel (Eds.), Tools for High Performance Computing 2009, Springer Berlin Heidelberg, 2010, pp. 31–38, http://dx.doi.org/10.1007/978-3-642-11261-4_3.

[29] A. Weingram, Y. Li, H. Qi, D. Ng, L. Dai, X. Lu, xCCL: A survey of industry-led collective communication libraries for deep learning, J. Comput. Sci. Tech. 38 (1) (2023) 166–195, http://dx.doi.org/10.1007/S11390-023-2894-6.

[30] C. Chen, K.S. Khorassani, P. Kousha, Q. Zhou, J. Yao, H. Subramoni, D.K. Panda, MPI-xCCL: A portable MPI library over collective communication libraries for various accelerators, in: Proceedings of the SC23 International Conference on High Performance Computing, Network, Storage, and Analysis Workshops, ACM, 2023, pp. 847–854, http://dx.doi.org/10.1145/3624062.3624153.