Tuning MPI Collectives by Verifying Performance Guidelines

Sascha Hunold
hunold@par.tuwien.ac.at
TU Wien, Faculty of Informatics
Favoritenstrasse 16
Vienna, Austria

Alexandra Carpen-Amarie
carpenamarie@par.tuwien.ac.at
TU Wien, Faculty of Informatics
Favoritenstrasse 16
Vienna, Austria

ABSTRACT
MPI collective operations provide a standardized interface for performing data movements within a group of processes. The efficiency of collective communication operations depends on the actual algorithm, its implementation, and the specific communication problem (type of communication, message size, number of processes). Many MPI libraries provide numerous algorithms for specific collective operations. The strategy for selecting an efficient algorithm is often times predefined (hard-coded) in MPI libraries, but some of them, such as OpenMPI, allow users to change the algorithm manually. Finding the best algorithm for each case is a hard problem, and several approaches to tune these algorithmic parameters have been proposed. We use an orthogonal approach to the parameter-tuning of MPI collectives, that is, instead of testing individual algorithmic choices provided by an MPI library, we compare the latency of a specific MPI collective operation to the latency of semantically equivalent functions, which we call the mock-up implementations. The structure of the mock-up implementations is defined by self-consistent performance guidelines. The advantage of this approach is that tuning using mock-up implementations is always possible, whether or not an MPI library allows users to select a specific algorithm at run-time. We implement this concept in a library called PGMPITuneLib, which is layered between the user code and the actual MPI implementation. This library selects the best-performing algorithmic pattern of an MPI collective by intercepting MPI calls and redirecting them to our mock-up implementations. Experimental results show that PGMPITuneLib can significantly reduce the latency of MPI collectives, and also equally important, that it can help identifying the tuning potential of MPI libraries.

KEYWORDS
MPI, collective operations, tuning, performance guidelines

1 INTRODUCTION
The Message Passing Interface (MPI) is still the most prominent and probably the most frequently used programming model for supercomputers, for example, MPI is driving most of the machines on the TOP500 list. The scalability of parallel applications running on these large platforms is therefore directly dependent on the performance of the underlying MPI implementations. The performance of MPI libraries is therefore of utmost importance for the overall efficiency of the software stack.

In the present article, we address the problem of optimizing the performance of MPI libraries, that is, we want to minimize the latency of a given MPI function for a given payload and a specific number of processes. The performance of MPI libraries can be improved in different ways. One possibility is to devise better algorithms for various communication patterns. Another possibility is to better exploit current hardware, e.g., by aligning memory segments to cache lines or by respecting ccNUMA domains when allocating memory chunks. Altogether, typical open-source MPI implementations, such as MPICH, MVAPICH, or OpenMPI, provide several algorithms for each MPI function, and each of these individual implementations may be able to leverage some hardware-specific optimizations.

Now, the problem is that potentially all provided algorithmic and hardware parameters that an MPI library provides must be considered when tuning on a given parallel machine. The goal of such a tuning processes is to select the best possible algorithm for a given message size and number of processes (and possibly other factors like the process to core mapping, etc.). Since libraries allow developers to control and vary hundreds of parameters (e.g., OpenMPI), the search space can be extremely large and tuning will be extremely costly. Moreover, parameter tuning may suffer from the fact that tuning is done for individual MPI functions, often in isolation and without a baseline implementation. Thus, having found the best set of parameters for a specific function (e.g., MPI_Gather) will not guarantee its efficiency.

Self-consistent performance guidelines can help to provide such a performance baseline. An MPI performance guideline states that the currently inspected, specialized MPI functionality, say functionality A, should not be slower than a less-specialized, but semantically equivalent functionality, say B (A ⪯ B). For example, the specialized MPI_Gather function, which only works with equal-sized messages, should not take longer than more generic MPI_Bcast function on the same equal-sized problem.

In previous work [6], we have shown that many MPI libraries available on production systems violate performance guidelines for several blocking MPI collective operations. We have also demonstrated that guideline violations can be avoided by changing the algorithm used in a specific case. However, not all guideline violations could be fixed by changing the algorithm. First, only some MPI libraries contain multiple algorithmic strategies for each MPI function. Second, many proprietary libraries do not expose algorithmic variants in form of adjustable parameters to the programmer. In both cases, performance violations of MPI libraries cannot be repaired at library level, and in these cases, a programmer would have to adapt the application code (e.g., switching from MPI_Allgatherv to MPI_Bcast).

In order to address this problem, we make the following contributions. We propose the library PGMPITuneLib, which can be used to improve the performance of any MPI library. PGMPITuneLib replaces the default implementation of an MPI function with its semantically equivalent mock-up version, if the corresponding performance guideline has been violated. We propose a tuning strategy...
that allowed us to automatically repair all guideline violations on three different test machines including a BlueGene/Q.

2 BACKGROUND AND RELATED WORK

Due to the diversity of parallel hardware, it is not surprising that MPI libraries only provide implementations of the MPI standard in a best-effort manner, i.e., the decision which underlying algorithm to use for a given case is predefined in a library. Nonetheless, as systems are, among themselves, usually very heterogeneous, it is necessary to adapt/tune MPI libraries to hardware. This tuning process is very difficult for two main reasons: first, the number of parameters that MPI libraries (e.g., Open MPI) expose for tuning can be very large. In addition, theoretically one would need to examine all possible variants of mapping processes to cores and all possible message sizes, which would simply be infeasible. Second, the optimization functions are often not convex (for minimizing the run-time), which makes it harder to find the optimal value as the problems may become intractable. Previous work on library tuning faced these problems, and we will summarize three approaches.

Charaawi et al. [2] developed the Open Tool for Parameter Optimization (OTPO), whose task is to find a good set of parameter values for a given number of processes and an MPI function. It basically performs a brute-force search over all specified parameters and their ranges in Open MPI. A related method was proposed by Pjesivac-Grbovic et al. [7], in which a quadtree scheme is used to encode the best collective algorithm for a given pair of (number of processes, message size). The quadtree is the internal data-structure for allowing a fast lookup of the best-suited algorithm. Since the quadtree can be limited in its depth and granularity, this tuning approach avoids a full enumeration of the search space. A different method was proposed by Sikora et al. [10], where a user can specify parameters and their ranges that should be tuned. Then, a plugin of the Periscope Tuning Framework tries to find the best configuration of these parameters by applying a meta-heuristic, in this case a genetic algorithm. In contrast to previous approaches, the tool of Sikora et al. [10] benchmarks and optimizes the run-time of entire MPI applications instead of optimizing individual MPI functions.

The mentioned previous approaches try to optimize the runtime of MPI functions for different message sizes but using a fixed number of processes. It is also possible to search for optimization potential by looking at the scalability behavior of MPI functions, as it was done by Shudler et al. [9]. In general, MPI functions have an expected and an actual performance, and the expected performance depends on the theoretical lower bound of an algorithm, which can be obtained analytically for different network topologies [3]. The approach of Shudler et al. [9] compares the expected scalability curve of an MPI function to the actual, measured scalability curve. A mismatch indicates that an MPI function has tuning potential.

Performance guidelines (previously called “performance requirements”) can be used to verify the consistency of an MPI library. In MPI, several communication patterns can be expressed in a semantically equivalent way. For example, the specialized MPI function MPI_Allreduce can also be implemented by chaining calls to MPI_Reduce and MPI_Bcast together. The user’s expectation is that the composition of the latter two functions should not be faster than executing the specialized one. In a more formal definition [11], a performance guideline is defined between two functionalities A and B, which semantically implement the same operation. If functionality A is the more specialized of the two, we can state that $\text{MPI}_A(n) \leq \text{MPI}_B(n)$, which means that A should complete faster than B for a comparable communication volume $n$. The communication volume $n$ should be understood as the amount of “actual” data items. It is possible that functionality B needs to transfer messages of larger size, e.g., $pn$, to mimic functionality A with n data items and $p$ processes. However, as B mimics A, only a communication volume of size $n$ is relevant. The majority of MPI performance guidelines are defined for a fixed number of processes and for the same communicator. Guidelines for different communicators can also be devised, but they are not considered in this work.

In previous work [6], we have implemented and tested several performance guidelines for blocking, collective MPI operations, such as MPI_Bcast. Our goal was to get an overview of how many libraries violate such guidelines in practice. For that task, we have implemented the toolkit PGMPI\(^1\), which distinguishes three classes of performance guidelines: monotony, split-robustness, and pattern. The monotony guideline ensures that increasing the message size(s) also increases the run-time. The goal of the split-robustness guideline is to ensure that splitting a communication operation into smaller chunks does not improve the overall performance. Last, pattern guidelines are defined between semantically equivalent operations, e.g., $\text{MPI}_\text{Allreduce} \preceq \text{MPI}_\text{Reduce} + \text{MPI}_\text{Bcast}$. We have shown that all tested MPI libraries (MVAPICH, Open MPI, Intel MPI) violate performance guidelines in various cases. In addition, we have demonstrated how violations of performance guidelines can be fixed by selecting a better underlying algorithm for a specific communication operation.

In the present paper, we combine the detection of performance-guideline violations with the tuning of MPI libraries. Our previous work [6] pointed out two problems: first, several, often vendor-provided MPI implementations lack user-controlled parameters for algorithmic tuning. Second, some MPI libraries only provide a small set of algorithmic choices for several MPI functions. In such cases, even though a performance violation has been detected, it cannot be repaired due to the limited number of algorithms provided.

Therefore, we propose to use performance-guideline variants as possible replacement implementations. The idea is the following: a guideline may state $\text{MPI}_\text{Gather} \preceq \text{MPI}_\text{Gatherv}$, which is a natural and almost trivial requirement. If an MPI library violates this guideline and if no other (or faster) variant (algorithm) implementing $\text{MPI}_\text{Gather}$ is available, the scientific programmer either needs to accept inferior performance, or she could refactor the code and replace the call to $\text{MPI}_\text{Gather}$ with a call to $\text{MPI}_\text{Gatherv}$. Yet, as this optimization might only be useful on machine $Z$ with $K$ processes, it does not seem to be a good strategy, in general. We solve this problem by introducing PGMPI-TuneLib, which sits between the MPI user code and the MPI library. By using the PMPI-interface, it intercepts calls to a specific MPI function, say $\text{MPI}_\text{Gather}$, and redirects them to an internally implemented $\text{MPI}_\text{Gather}$ function, which uses $\text{MPI}_\text{Gatherv}$ as its base implementation. Our approach is in the spirit of the approaches of Pjesivac-Grbovic et al. [7] and Faraj et al. [4]. The latter authors proposed the STAR-MPI library.

\(^{1}\)https://github.com/hunsa/pgmpi
which selects an algorithm for a collective operation (online) after benchmarking (timing) several algorithmic variants during the runtime of an application. Our PGMPITuneLib library instead provides several implementations of a specific collective MPI_C, and each variant corresponds to one performance guideline. We then profile MPI functions in isolation (offline) and check for performance-guideline violations. If violations occur, we record these cases and later redirect MPI calls (online) to faster implementations during application runs. Instead of quadtrees, PGMPITuneLib uses a combination of hash functions and binary searches, ensuring an efficient lookup of algorithmic variants for a given number of processes and message size, which in our case can be done in time $O(\log m)$, where $m$ denotes the largest message size that may occur.

3 AUTOTUNING MPI LIBRARIES WITH PGMPITUNELIB

Now, we describe our approach for autotuning blocking MPI collective operations using PGMPITuneLib. First, we show all performance guidelines that our library comprises and give a short explanation for each of them. Second, we discuss implementation details of the library and describe the tuning process.

3.1 Performance Guidelines and Semantics

Currently, PGMPITuneLib contains implementations of the performance guidelines listed in Equations (GL1)–(GL22), some of which were introduced before [6, 11].

\[
\begin{align*}
\text{MPI} \text{Allgather}(n) &\leq \text{MPI} \text{Bcast}(n) \\
\text{MPI} \text{Allgather}(n) &\leq \text{MPI} \text{Alltoall}(n)
\end{align*}
\]

We omit $E$ we would have to say precisely $nE$ where $E$ is the extent of the base datatype used. However, for the sake of a better readability we omit $E$ and simply say $n$ for the memory requirement. Table 1 also uses variable $I$, which denotes the extent of MPI_INT, which is commonly needed when specifying the displacement and the receive (send) count vectors.

In the following, we give a brief summary of the idea and implementation behind each performance guideline.

**MPI_Allgather and its Mock-ups.**

(GL1) trivially composes MPI_Gather with MPI_Bcast to obtain a functionally equivalent version of MPI_Allgather.

(GL2) uses a $p$ times larger send buffer, in which each process puts $p$ copies of its own buffer contents. Then, MPI_Alltoall is called to mimic MPI_Allgather.

(GL3) uses, similar to the MPI_Alltoall mock-up, a $p$ times larger send buffer. This larger buffer is initialized with zeros, and the actual message of each process is copied into the large buffer starting at index $i \cdot n$. Then, an MPI_Allreduce is applied to all buffers and
Table 1: Performance guidelines implemented in PGMPITuneLib. Variable $n$ denotes the number of elements of basetype in the send count of an operation, $p$ denotes the number of processes in the communicator, and $l$ denotes the size of MPI_INT.

| MPI collective | max memory requ. per proc | guideline | mock-up | add. mem. requirement |
|----------------|--------------------------|-----------|---------|-----------------------|
| MPI_Allgather  | $n + pn$                 | GL1       | MPI_Gather + MPI_Bcast | none |
|                |                          | GL2       | MPI_Alltoall            | $pn$ (p times larger send buffer) |
|                |                          | GL3       | MPI_Allreduce            | $pn$ (p times larger send buffer) |
|                |                          | GL4       | MPI_Allgatherv           | $2pl$ (displs, recvcounts) |
| MPI_Allreduce  | $2n$                     | GL5       | MPI_Reduce + MPI_Bcast   | none |
|                |                          | GL6       | MPI_Reduce_scatter_block + MPI_Allgather | $(n + c) + (n + c)/p$ (small c for padding) |
|                |                          | GL7       | MPI_Reduce_scatter + MPI_Allgatherv | $\max\{\lfloor n/p \rfloor + C, C\}$ (chunk size C) + $2pl$ (displs, recvcounts) |
| MPI_Alltoall   | $2pn$                    | GL8       | MPI_Alltoallv           | $2pl$ (displs, recvcounts) |
| MPI_Bcast      | $n$                      | GL9       | MPI_Allgatherv           | $2pl$ (displs, recvcounts) + $n$ (for recv buf) |
|                |                          | GL10      | MPI_Scatter + MPI_Allgatherv | $(n + c) + (n + c)/p$ (small c for padding) |
| MPI_Gather     | $n + pn$                 | GL11      | MPI_Gather              | none on root, $pn$ on other processes |
|                |                          | GL12      | MPI_Gatherv              | $2pl$ (displs, recvcounts) |
|                |                          | GL13      | MPI_Reduce               | $pn$ (for new send buf) |
| MPI_Reduce     | $n + n$ (on root)        | GL14      | MPI_Allreduce            | extra $n$ (on processes other than root) |
|                |                          | GL15      | MPI_Reduce_scatter_block + MPI_Gather | $(n + c) + (n + c)/p$ (c for padding) |
|                |                          | GL16      | MPI_Reduce_scatter + MPI_Gatherv | $\max\{\lfloor n/p \rfloor + C, C\}$ (chunk size C) + $2pl$ (displs, recvcounts) |
| MPI_Reduce_scatter_block | $n + np$ | GL17      | MPI_Reduce + MPI_Scatter | $n$ (for first reduce) |
|                |                          | GL18      | MPI_Reduce_scatter       | $pl$ (recvcounts) |
|                |                          | GL19      | MPI_Allreduce            | $n$ (for new recv buffer) |
| MPI_Scan       | $2n$                     | GL20      | MPI_Excscan + MPI_Reduce_local | none |
| MPI_Scatter    | $n + np$                 | GL21      | MPI_Bcast                | extra $n$ (on processes other than root) |
|                |                          | GL22      | MPI_Scatterv             | $2pl$ (displs, recvcounts) |

A bit-wise or-operation ensures that the result is semantically equivalent to the result of MPI_Allgather.

(GL4) calls MPI_Allgatherv instead, and therefore needs to allocate two additional buffers of size $p$ ($p$ elements of type MPI_INT) for the receive counts and the displacements. Note that we will not further comment on any other mock-up implementation using an irregular operation (e.g., MPI_Gatherv, MPI_Alltoallv, etc.), as they are all straightforward to implement.

**MPI_Allreduce and its Mock-ups.**

(GL5) composes straightforwardly MPI_Reduce and MPI_Bcast. (GL6) first calls MPI_Reduce_scatter_block, which needs equal-sized blocks in the Scatter phase. As the send buffer of the original MPI_Allreduce function does not have to be a multiple of the number of processes, our mock-up version will add (a maximum of $p - 1$) dummy elements of the send type as additional padding. It is now possible to perform an MPI_Allgather on the receive buffer of the previous stage. After this MPI_Allgather has been completed, the mock-up version only copies the first $n$ elements (ignoring the padded elements) back to the original receive buffer.

(GL7) applies a similar strategy as the previous mock-up function. Since MPI_Reduce_scatter and MPI_Allgatherv work with send buffers of arbitrary size, the mock-up function only needs to allocate and properly handle buffers for the receive counts and for the displacements. We also introduce a variable $C$, $1 \leq C \leq n$, which denotes the minimum size of chunks that are distributed to each process in the scatter phase. Thus, if $C = 1$, each process receives roughly $n/p$ elements in the scatter phase. If $C = n$, only one process receives elements in the scatter phase.

**MPI_Bcast and its Mock-ups.**

(GL9) denotes one specific process to be the root process of the broadcast operation. This process (e.g., rank 0) allocates a send buffer of the size of the original broadcast operation. All other processes contribute zero bytes to the result of the allgather operation. Then, a call to MPI_Allgatherv copies the buffer contents of (the fake) root rank to all other processes in a broadcast-to-all fashion. As MPI_Allgatherv works with different send and receive buffers, an additional receive buffer is needed, holding $n$ elements. Additionally, each process must allocate two buffers of size $p$ for the count and displacement information.

**MPI_Gather and its Mock-ups.**

(GL11) This mock-up simply uses MPI_Allgather behind the MPI_Gather-interface. As now every process needs to receive $n$ elements of the base datatype from every other process, we need to allocate a buffer with space for $p \cdot n$ elements of basetype. Calling MPI_Allreduce will not only give the root the result but also the other processes, which simply ignore the result.

**MPI_Reduce and its Mock-ups.**

(GL14) uses the same strategy as guideline (GL11). Every process, except the root process of the operation, needs to allocate a receive buffer that can accommodate $n$ basetype elements. Calling MPI_Allreduce will not only give the root the result but also the other processes, which simply ignore the result.

(GL15) This mock-up is a rather heavyweight replacement of MPI_Reduce. It first performs an MPI_Reduce_scatter_block on...
the send buffer. For the scatter part, the vector size must be a multiple of the number of processes, as MPI_Reduce_scatter_block requires send buffers of the same size. To achieve that, an extra padding is added to the end of the send buffer. Two new buffers are allocated, one holding the new, padded send buffer and another one for the result of the MPI_Reduce_scatter_block operation, which is exactly $p$ times smaller than the new send buffer. Upon completion of this operation, we can call MPI_Gather on the result buffers of MPI_Reduce_scatter_block, which finally gives us an emulated version of MPI_Reduce.

(GL16) The idea of this mock-up is similar to the one above. The only difference is that we do not need the additional padding, as MPI_Reduce_scatter works on vectors of arbitrary size. However, to accomplish an emulation, we need to allocate two buffers for the displacement and the count information that will be used for MPI_Reduce_scatter and the following MPI_Gatherv. The chunk size $C$ has the same meaning as in guideline (GL7), i.e., chunks of size $C$ are assigned to processes in round-robin fashion.

MPI_Reduce_scatter_block and its Mock-ups.

(GL17) This mock-up function uses a straightforward composition of MPI_Reduce and MPI_Scatter. As the result of the first step (MPI_Reduce) requires a receive buffer of size $n$ (elements), we need to allocate this additional buffer between the two calls.

(GL18) is a trivial emulation using the irregular counterpart, for which an additional buffer holding the receive counts is required.

(GL19) The MPI_Reduce_scatter_block functionality can also be emulated with MPI_Allreduce. We need to allocate an additional receive buffer on each but the root process with space for $n$ elements. Allreduce will then distribute the result to all processes. Now, each process picks its part of the reduction result, which it would have received from a scatter operation. This completes the emulation of MPI_Reduce_scatter_block.

MPI_Scan and its Mock-ups.

(GL20) This mock-up version performs first an exclusive scan on the same data as the inclusive scan would have performed. In order to obtain the same result as the inclusive scan, we need to perform a local reduction operation on all processes but the root. Overall, no additional buffers are needed.

MPI_Scatter and its Mock-ups.

(GL21) This version allocates on all processes but the root an additional receive buffer for the $n$ elements of the root process. The root process then broadcasts all its data to the others. Now, every process (also the root) copies its part of the data ($n/p$ elements) to the receive buffer of the scatter operation.

3.2 Library Design and Implementation

3.2.1 General Design. As the library PGMPITuneLib makes use of the PMPI interface of MPI, it is layered between the MPI user code and the MPI library. If the user code calls an MPI function, in our case a blocking MPI collective, PGMPITuneLib intercepts the call and may select one of the mock-up implementations. Figure 1 shows an example: the MPI user code calls MPI_Allreduce, which is intercepted by PGMPITuneLib. Internally, PGMPITuneLib uses performance profiles containing identifiers of possible replacement algorithms for various message sizes. Therefore, PGMPITuneLib searches for a replacement algorithm for MPI_Allreduce. If such a replacement algorithm can be found, PGMPITuneLib emulates the original call by using its replacement, which is in our example the combination of MPI_Reduce and MPI_Bcast. If no replacement algorithm is found, PGMPITuneLib uses the default implementation, i.e., it calls PMPI_Allreduce.

3.2.2 Tuning Workflow and Modes of Operation. PGMPITuneLib provides two modes of operation, which are encapsulated in different libraries and which can be linked with an arbitrary MPI application (or benchmark). Figure 2 shows the general architecture, where PGMPITune provides the basic API. On top of that core API, two different libraries exist. One is the library called PGMPITuneCLI (CLI stands for command line interface), which is used for benchmarking the performance of mock-up implementations. To that end, MPI developers link their applications against the CLI version of PGMPITuneLib. It is now possible to select a mock-up version for a specific MPI function as follows:

```
mpicxx *.c -o mympicode -lpromptunecli -lmpi
mpirun -np 2 ./mympicode
--module=allgather:alg=allgather_as_gather_bcast
```

In this example, all calls to MPI_Allgather will be replaced with the mock-up implementation of guideline (GL1). By using this CLI
version of PGMPITuneLib, we can analyze the latency of all implemented collective algorithms for different message sizes. Any MPI benchmark suite can be used to measure the latency of MPI collective operations. Benchmarking allows us to discover the message sizes for which the performance guidelines are violated. When violations occur, PGMPITuneLib stores which message sizes a possible replacement mock-up has been found. After scanning over all collectives and selected message sizes, PGMPITuneLib writes a performance profile for each MPI collective. Performance profiles contain the replacement algorithms for specific message ranges. Listing 1 shows a sample profile for MPI_Scatter that was recorded with 64 × 16 processes on JUQUEEN. Each profile only contains message ranges for which violations have occurred and for which a replacement algorithm should be used. As we have measured in this example for discrete message sizes, the sample profile uses the same message size for the start and the end of a message range. For example, algorithm 2 (scatter_as_bcast) should be applied for the message ranges 1Byte to 1Byte, 8Bytes to 8Bytes, and so on.

After the performance profiles have been written, any MPI application can use these profiles. A developer simply needs to link their application against the PGMPITuneD library. Similar to PGMPITuneCLI, the PGMPITuneD library intercepts MPI calls and redirects them to the mock-up versions implemented in the core library. PGMPITuneD reads in all performance profiles from disk, which happens transparently when intercepting MPI_Init. Then, PGMPITuneD has all the information required to select a (possibly) better mock-up version for a collective MPI operation at run-time.

3.2.3 Implementation Details. It seems obvious that a tuned MPI library should be implemented as efficiently as possible. We have therefore tried to keep the overhead incurred by PGMPITuneLib very low. As mentioned before, some mock-up implementations require the allocation of additional memory, e.g., for padded data or for displacement or send/receive count vectors. First, PGMPITuneLib avoids additional system calls (e.g., malloc) and allocates two memory chunks at the start of the MPI program, one for additional message buffers and one for displacement or send/receive count vectors. The size of both buffers can be controlled by the user with the variables size_msg_buffer_bytes and size_int_buffer_bytes, which can be set in the configuration file of PGMPITuneLib. Another advantage of this additional memory management is that users can accurately control how much extra memory they want to dedicate for possibly faster MPI functions. Now, cases may occur, where a replacement algorithm was found to be faster than the default implementation provided by the MPI library, but such a mock-up would need too much extra memory and will therefore not be selected.

For providing an efficiently tuned library, it is also important to perform fast look-ups to check whether a replacement algorithm is available. Currently, performance profiles are read for a specific number of processes only. Thus, PGMPITuneLib can look up the right performance profile for a certain collective and can check whether the profile is compatible with the current number of processes in time \(O(1)\). Then, PGMPITuneLib only needs to verify whether the profile contains a replacement algorithm for the current message size. As we sort the \(M\) different message ranges at program start, such a lookup of the replacement algorithm is performed in time \(O(\log M)\) using binary search.

4 EXPERIMENTAL EVALUATION

4.1 Hardware Setup

We have evaluated PGMPITuneLib on three different machines, whose characteristics are summarized in Table 2. The systems Jupiter and VSC-3 are rather similar when comparing their hardware setup. The advantage of having similar architectures is that the reproduction of phenomena on other systems increases the confidence in the significance of our findings. A BlueGene/Q called JUQUEEN allows us to study a vendor-provided, tailor-made MPI library on an actual supercomputer.

4.2 Tuning Workflow

The tuning process of PGMPITuneLib first checks whether the performance guidelines defined for blocking collective MPI operations are fulfilled. If violations occur, these cases are recorded and a profile is written. In a subsequent execution, PGMPITuneLib can then change to a different mock-up implementation at run-time.

In order to automatically tune an MPI library, we need to benchmark the latency of the default implementations of blocking collectives and their mock-up versions that are part of PGMPITuneLib. For measuring the latency, one could employ any type of MPI benchmark suite, e.g., OSU Micro-Benchmarks [1] or SkA MPI [8]. It is only required that the benchmark suite is linked against PGMPITuneLib.

For the analysis shown in the present paper, we have used our own benchmark suite called ReproMPI\(^2\), which allows to record raw data (the latency of every single measurement) from each experiment [5]. In contrast to other benchmark suites, it refrains...

Listing 1: Profile of MPI_Scatter on JUQUEEN.

|   |   |   |
|---|---|---|
| 1 | # pgtune profile |   |
| 2 | MPI_Scatter |   |
| 3 | 1024 # nb. of processes |   |
| 4 | 2 # nb. of mock-up impl. |   |
| 5 | 2 scatter_as_bcast |   |
| 6 | 3 scatter_as_scatterv |   |
| 7 | 8 # nb. of ranges |   |
| 8 | 1 1 2 # byte_range_start byte_range_end alg_id |   |
| 9 | 8 8 2 |   |
| 10 | 32 32 2 |   |
| 11 | 64 64 2 |   |
| 12 | 100 100 2 |   |
| 13 | 512 512 2 |   |
| 14 | 1024 1024 2 |   |
| 15 | 10000 10000 3 |   |

Table 2: Parallel machines used in our experiments.

| Name  | Hardware                                      | MPI Libraries                  | Compiler |
|-------|-----------------------------------------------|--------------------------------|----------|
| Jupiter | 36 × Dual Opteron 6384 @ 2.3 GHz             | MVAPICH2.2                     | g++ 4.4.7 |
|       | IB QDR MT26428                                |                                |          |
| VSC-3 | 2000 × Dual Xeon ES-2690W2 @ 2.6 GHz         | Intel MPI 2017 (Update 2) icc 16.0.4 |          |
|        | IB QDR-80                                    |                                |          |
| JUQUEEN | 28-672 × IBM PowerX2 @ 1.6 GHz               | IBM BG MPI                     | IBM XL   |
|        | IBM-BlueGene/Q, 3D Torus interconnect         |                                |          |

\(^2\)https://github.com/hunsa/reprompi
from performing any kind of data aggregation (e.g., computation of means) or data removal (e.g., discarding the first X measurements for "warming up" the system). By using ReproMPI, we can record every single measurement and perform the data analysis in R, Python, or Julia later.

The auto-tuning process with PGMPITuneLib is divided into three steps. The first and a critical step is to estimate the number of repetitions (nrep) of measurements that have to be conducted for a specific MPI function with a given message and communicator size (number of processes). Second, we benchmark the MPI collectives and their mock-up counterparts using the CLI version of PGMPITuneLib (cf. Figure 2). From the performance data gathered, we can then detect violations of the performance guidelines (GL1)–(GL22). Among all mock-up functions for which guideline violations have occurred, the mock-up version that performs best for a given message range is selected and written into a performance profile (cf. Listing 1). In a last step, we re-link the ReproMPI benchmark suite against the tuned version of PGMPITuneLib. Now, PGMPITuneLib can replace individual MPI collectives with their faster mock-up counterparts.

ReproMPI supports different synchronization strategies and different ways (clocks) to measure the run-time (latency) of collective calls. For the presented experiments, we have used the timing procedure shown in Algorithm 1: Before every individual measurement, processes are synchronized with a barrier. Here, we use a dissemination barrier, which—due to its structure—ensures that processes leave this barrier relatively synchronized (which would not be the case for tree-based barriers for example).

The "NREP problem" consists of finding a suitable (and possibly minimal) number of repetitions, such that the derived statistical measures (mean, median) are reproducible [6].

We use the following method to address the NREP problem: for each MPI functionMPI_A, the general idea is to determine the time (t_{nrep}^1) until the latency measurements with a 1 Byte message have stabilized (e.g., a small variance). This time is further used as the reference time for other message sizes (cf. Figure 3). Our assumption is that relative system noise decreases when the message size increases, as the run-time of each collective grows with the message size. Thus, we measure the latency of MPI_A with a different message size msizes (msize > 1 Byte) for at least time t_{nrep}^1.

More precisely, in step (1), we repeat measuring the latency of functionMPI_A with 1 Byte messages until the current relative standard error (RSE) over the measured latencies is below some user-defined threshold. We repeat this process over several calls to mpirun and take the longest time that was required to make the RSE drop below the threshold, and we denote this time as t_{nrep}^1. As this process can take hundreds of repetitions, we only want to do that for a message size of 1 Byte. For all the other messages sizes, we measure for at least time t_{nrep}^1. As many benchmarking tools require the number of repetitions as an input, we convert the time t_{nrep}^1 into a number of repetitions nrep_msize of MPI_A for any other message size msize. To that end, in step (2), we run two batches of measurements called batches b1 and b2, and this user-defined number of repetitions for both batches should be relatively small (<10) or even zero in case of b2. We compute the RSE value of these b1 measurements. If that value is smaller than some predefined threshold (note this is a different threshold than used for 1 Byte messages), measuring is stopped. Otherwise, another batch with b2 measurements is started. For larger message sizes, taking one batch with b1 elements leads to a very small variance, and thus, we can return quickly. However, for smaller message sizes, we need a few more measurements to get a reasonable value of the latency. In step (3), we compute the minimum latency of these b1 + b2 measurements (b2 may be 0), t_msize = min_{1\leq i\leq b1+b2} t_i, and use this value as the "expected" latency when measuring. Then, we compute the estimated number of repetitions needed for MPI_A and message size msize as nrep_msize = max \left\lceil \frac{t_{nrep}^1}{t_msize} \right\rceil , K. The value K ≥ 1 ensures that at least K latency measurements for every collective are performed, especially when t_msize becomes very large.

In our experiments, we use the following values to estimate the nrep value for each collective or mock-up: we repeat measuring the latency with msize = 1 Byte until the RSE value is smaller than 0.01 (1%). We perform b1 = 5 and possibly b2 = 5 more measurements for each collective (and mock-up) and with larger message sizes and then compute nrep_msize. We measure the latency of each collective and its mock-ups for nrep_msize iterations and repeat that for mmpiruns = 5 different calls to mpirun. The selection of which mock-up function to use is done statically on the command line (PGMPITuneCLI). We check for guideline violations of each collective and write a performance profile to disk, if violations have occurred. In our particular case, we only replace a collective with its mock-up if the mock-up is at least 10% faster than the default implementation. We can then run another set of experiments with the tuned version of the MPI library. The selection of the best implementation (default or mock-up) is done dynamically at runtime by PGMPITuneD.

Listing 2 shows the output of ReproMPI when being run and linked against PGMPITuneLib. The output is directly readable as

![Figure 3: Estimating the number of repetitions (NREP) based on the time obtained for 1 Byte messages.](image-url)
CSV data into data processing frameworks like R. The header contains information about the specific benchmarking run, e.g., how many processes, which clock, or which barrier implementation have been used. However, the footer is written by PGMPITuneLib and shows whether certain calls to MPI collectives have been replaced. In the example, for a message size of 100 Bytes, the default implementation of MPI_Allgather has been replaced by the mock-up implementation MPI_Gather + MPI_Bcast. In some other cases, for example with 8 Bytes, the Default implementation has been used. The footer also contains information about how much memory has been reserved for the temporary buffers in PGMPITuneLib. Here, PGMPITuneLib could use additional 100 MBytes for allocating message buffers and 10 kBytes for displacement and count vectors.

Listing 2: ReproMPI output when benchmarking a tuned MPI library; some lines were omitted for better readability.

```sh
# @pgmpi config size_int_buffer_bytes 10000
# @pgmpi config size_msg_buffer_bytes 100000000
# @pgmpi alg MPI_Allgather 1 allgather_as_gather_bcast
# @pgmpi alg MPI_Allgather 512 allgather_as_gather_bcast
# @pgmpi alg MPI_Allgather 4096 allgather_as_gather_bcast
# @pgmpi alg MPI_Allgather 100000 default
# @pgmpi alg MPI_Allgather 8192 default
# @pgmpi alg MPI_Allgather 32768 default
# @pgmpi alg MPI_Allgather 32 default
# @pgmpi alg MPI_Allgather 100 allgather_as_gather_bcast
# @pgmpi alg MPI_Allgather 8 default
# @pgmpi alg MPI_Allgather 16000 default
```

The footer also contains information about how much memory has been reserved for the temporary buffers in PGMPITuneLib. Here, PGMPITuneLib could use additional 100 MBytes for allocating message buffers and 10 kBytes for displacement and count vectors.

4.3 Experimental Results

Figure 4 summarizes the tuning results that were obtained for 32 × 1 processes and Open MPI 2.1.0 on *Jupiter*. Each plot contains the performance of the Default algorithm, the Tuned version, and the individual mock-up implementations. As latencies for small and large messages differ by orders of magnitude, we plot the relative performance of each implementation, where the latency of the Default implementation is used as reference. As we measure over multiple calls to `mpirun` (*mpiruns*), we use the median over the *mpiruns* = 5 median latencies measured. The error bars denote the minimum and the maximum of these *mpiruns* medians to reflect the variance of the data. For a better comprehension, let us look at the plot on the right-hand side of Figure 4, which compares the Tuned and the Default version of MPI_Gather. The figure also includes the performance data of three different mock-up implementations of MPI_Allgather (Allgather, Gather, Reduce). We can observe that the Tuned version uses Gather as replacement up to a message size of 1024 Bytes, for which PGMPITuneLib switches back to the Default version. Except for 8192 Bytes, the Default version has been found to perform best for larger message sizes. As the scale of the y-axis is limited, not all individual points are shown, e.g., the red points for the Allgather mock-up.

The data shown in Figure 4 suggest that there is a large tuning potential to improve *MPI_Reduce* in Open MPI, and this case will be considered in Section 4.4.

With MVAPICH2-2.2, different cases were detected for which PGMPITuneLib can improve the performance (see Figure 5). As the relative latency sometimes can be misleading, we also indicate the absolute performance difference for a few cases. For example, the latency of *MPI_Reduce* or MPI_Gather with 32 KiBytes of data can be reduced up to 180 µs or 400 µs, respectively (an improvement of roughly 50%).

Figure 6 shows the performance improvement achievable on *JUQUEEN* and 64 × 16 processes. It is interesting to note, but not surprising, that many performance violations have occurred when being tested against mock-up versions that rely on MPI_Bcast, e.g., for guideline (GL1). It seems often beneficial to employ MPI_Bcast, for which the BlueGene/Q provides hardware support. As a consequence of this, the performance of MPI_Allgather and MPI_Scatter could significantly be improved.

Notice that we only present a selection of the performance plots, showing the most significant results. More performance graphs can be found in Appendix A.

4.4 Parameter vs. Guideline-based Tuning

Now, we inspect two performance guideline violations, one of which we had already examined.

4.4.1 Case *MPI_Reduce* ≤ MPI_Allreduce. We had shown that *MPI_Reduce* violates the Allreduce guideline with Open MPI 1.10.1, for message sizes ranging from 128 kBytes to 725 kBytes and 32 × 16 processes on *Jupiter*. We were able to overcome this violation by implementing our own *MPI_Reduce* function. Now, we would like to go one step further and compare differently tuned versions of *MPI_Reduce*: (1) the best mock-up algorithm found by PGMPITuneLib and (2) the best algorithm found after an exhaustive search using the MCA parameters of Open MPI. To that end, we have varied the relevant MCA parameters (e.g., segment size, fan-out) for all Reduce algorithms provided by Open MPI. The result of this brute-force tuning and the results with PGMPITuneLib are compared in Figure 7. We can observe that the *MPI_Allreduce* mock-up is faster than the Default *MPI_Reduce* implementation over the entire range of message sizes. However, the latency can further be improved (although only moderately) by using the in-order_binary algorithm of Open MPI. This case exemplifies that scanning for guideline violations and performing a serious parameter tuning (e.g., MCA parameters in Open MPI) should complement each other. In this case, a fully parameter-tuned version of Open MPI would not have violated the *MPI_Reduce* performance guidelines in the first place. The downside is that such an exhaustive search is time-consuming.

4.4.2 Tuning Potential. Our extensive experimental analysis also revealed other interesting cases. One of them is shown in Figure 8. The plot shows latencies measured for *MPI_Allreduce*, its mock-up variants implemented in PGMPITuneLib, and several
Figure 4: Performance comparison between Default and Tuned version of Open MPI 2.1.0 (32 × 1 processes, Jupiter).

Figure 5: Performance comparison between Default and Tuned version of MVAPICH2-2.2 (32 × 1 processes, Jupiter).

Figure 6: Performance comparison between Default and Tuned version of IBM BG MPI (64 × 16 processes, JUQUEEN).

algorithmic versions found in Open MPI 2.1.0 (only the fastest ones). Here, the algorithmic version called MCA_nonoverlapping performs almost identical to our Reduce+Bcast mock-up variant. Indeed, when inspecting the internals of Open MPI, this algorithmic variant uses exactly these two collectives. Additionally, we discover that the mock-up version combining MPI_Reduce_scatter and MPI_Allgatherv outperforms all other algorithms, even all versions provided by Open MPI after the exhaustive search was done. Thus, PGMPTuneLib helps developers to detect cases for which a better algorithmic variant exists. We took the role of an Open MPI developer and implemented the Allreduce variant based on Reduce_scatter and Allgatherv within Open MPI 2.1.0. This version is denoted as MCA_NEW_Reduce_scatter+Allgatherv in Figure 8. The plot shows that this new algorithm in Open MPI exactly matches the expected latency achieved by the mock-up combining Reduce_scatter and Allgatherv and outperforms all other variants.

5 CONCLUSIONS
Tuning MPI libraries can be extremely rewarding in terms of overall efficiency of parallel machines, as MPI is the de-facto standard for...
data communication on larger distributed memory machines. Parameter tuning is usually a valuable method for achieving the goal of an improved MPI software layer. The downsides of parameter tuning are twofold: (1) it is relatively expensive as libraries such as Open MPI provide hundreds of possibly interacting parameters, and (2) a performance baseline is often missing, i.e., how good is good enough since global minima are usually unknown.

Tuning MPI libraries by using performance guidelines can complement the traditional parameter-based approach. Self-consistent performance guidelines define relations between the performance of a specialized functionality and a less specialized functionality, both of which realize semantically the same operation, e.g., the latency of $\text{MPI\_Allreduce}$ should be smaller than using $\text{MPI\_Gather}$ and a subsequent call to $\text{MPI\_Bcast}$.

In the present paper, we have extended performance guidelines for blocking, collective MPI operations. We have implemented each semantically matching guideline as a mock-up function in a library called PGMPITuneLib. With this library, it is possible to find performance deficits of MPI libraries by scanning for guideline violations. The library creates so-called performance profiles that can be used to replace specific MPI functions by their mock-up version at run-time.

Our experimental results show that PGMPITuneLib can indeed overcome performance problems of MPI libraries on all systems that we have tested on. In addition, our results also show that PGMPITuneLib also reveals cases in MPI libraries (e.g., Open MPI) for which even better algorithms exist. The biggest advantage of PGMPITuneLib, however, is the fact that it can be used with any MPI library, whether or not it exposes parameters for tuning purposes.

**ACKNOWLEDGMENTS**

We thank Bernd Mohr (Julich Supercomputing Centre) for providing us access to JUQUEEN.

**REFERENCES**

[1] OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[2] Mohammad Chaarawi, Jeffrey M. Squyres, Edgar Gabriel, and Saber Fekri. 2008. A Tool for Optimizing Runtime Parameters of Open MPI. In EuroPVM-MPI (LNCS), Vol. 5260. 210–217. DOI: http://dx.doi.org/10.1007/978-3-540-87475-1_30

[3] Ernie Chan, Marcel Heimlich, Arv Purkayastha, and Robert A. van de Geijn. 2007. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience* 19, 13 (2007), 1749–1783. DOI: http://dx.doi.org/10.1002/cpe.1206

[4] Ahmad Faraji, Xin Yuan, and David K. Lowenthal. 2006. STAR-MPI: self-tuned adaptive routines for MPI collective operations. In *International Conference on Supercomputing (ICS)*. ACM, 199–208. DOI: http://dx.doi.org/10.1145/1183401.1183431

[5] Sascha Hunold and Alexandra Carpen-Amarie. 2016. Reproducible MPI Benchmarking is Still Not as Easy as You Think. *IEEE Trans. Parallel Distrib. Syst.* 27, 12 (2016), 3617–3630. DOI: http://dx.doi.org/10.1109/TPDS.2016.259167

[6] Sascha Hunold, Alexandra Carpen-Amarie, Felix Donatus Lübke, and Jesper Larsson Träff. 2016. Automatic Verification of Self-consistent MPI Performance Guidelines. In *EuroPar 2016 (LNCS)*, Vol. 9833. Springer, 433–446. DOI: http://dx.doi.org/10.1007/978-3-319-43609-3_22

[7] Jelena Pjesivac-Grbovic, George Bosilca, Graham E. Fagg, Thara Angskun, and Jack Dongarra. 2007. MPI collective algorithm selection and quadtree encoding. *Parallel Comput.* 33, 9 (2007), 613–623. DOI: http://dx.doi.org/10.1016/j.parco.2007.06.005

[8] Ralf Reussner, Peter Sanders, and Jesper Larsson Träff. 2002. SkA MPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming* 10, 1 (2002), 55–65.

[9] Serghei Shudler, Alexandru Calotoiu, Torsten Hoefler, Alexandre Strube, and Felix Wolf. 2015. Exascaling Your Library: Will Your Implementation Meet Your Expectations?. In *International Conference on Supercomputing (ICS)*. 165–175.

[10] Anna Sikora, Eduardo César, Ilias A. Comprés Ureña, and Michael Gerndt. 2016. Autotuning of MPI Applications Using PTI. In *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*, Atul Kumar, Santonu Sarkar, and Michael Gerndt (Eds.). ACM, 31–38. DOI: http://dx.doi.org/10.1145/2916028.2916028

[11] Jesper Larsson Träff, William D. Gropp, and Rajeev Thakur. 2010. Self-Consistent MPI Performance Guidelines. *IEEE TPDS* 21, 5 (2010), 698–709. DOI: http://dx.doi.org/10.1109/TPDS.2009.120
A APPENDIX
We show a collection of performance graphs for JUQUEEN, VSC-3, and Jupiter. When no violations of performance guidelines are detected, we mark these cases with a gray background.

A.1 JUQUEEN

Figure 9: Performance comparison between Default and Tuned version of IBM BG MPI (64 x 16 processes, JUQUEEN)
Figure 10: Performance comparison between Default and Tuned version of Intel MPI (64 × 16 processes, VSC-3)
A.3 Jupiter

Figure 11: Performance comparison between Default and Tuned version of Open MPI 2.1.0 (32 × 1 processes, Jupiter)
Figure 12: Performance comparison between Default and Tuned version of Open MPI 2.1.0 (32 × 16 processes, Jupiter)
Figure 13: Performance comparison between Default and Tuned version of MVAPICH2-2.2 (32 × 1 processes, Jupiter)
Figure 14: Performance comparison between Default and Tuned version of MVAPICH2-2.2 (32 × 16 processes, Jupiter)