TU WIEN Informatics

Automatic Identification of Violations Against MPI Performance Guidelines

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Maximilian Alfred Hagn
Registration Number 11808237

to the Faculty of Informatics

at the TU Wien

Abstract

The *Message Passing Interface (MPI)* is the most widely used standard for message transfer between processes in parallel applications. Due to the wide distribution, it is used on most supercomputers and therefore many applications, some of them time-critical, rely on these *MPI* implementations. The wide range of use means that there are many providers who develop their own *MPI* implementations. For users, it is particularly important that the applications based on *MPI* run as efficiently as possible, especially if they are *High Performance Computing (HPC)* applications. Many providers offer users the possibility of improving the underlying algorithms by providing parameters for modification. This feature is only of partial benefit, since tuning the parameters is a real challenge and it is not offered by every vendor. Another way to improve the performance of *MPI* applications is to replace collective operations by other functions defined in the *MPI* standard that produce the same results. This approach has the advantage that it can be used vendor-independently, since the substitutes have to be implemented through the standard anyway. Self-consistent performance guidelines are also available for evaluating the quality of a *MPI* implementation. They define which functionalities should execute faster than their substitutes.

The objective of this thesis is to provide an approach to automatically check *MPI* implementations against the self-consistent *MPI* performance guidelines. For this purpose, we have developed a *C++* application, which is built on top of the *PGMPITuneLib* library. This library can be used for tuning as well as for creating performance profiles for different collective *MPI* operations and their equivalent alternatives. With our tool we make use of this library to create performance profiles, and based on the obtained information we detect violations against the self-consistent performance guidelines. With the developed application it becomes possible to test the quality of a *MPI* implementation in an arbitrary environment. To verify the functionality of the tool, we will also perform a detailed analysis on an external supercomputer.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

The *Message Passing Interface (MPI)* is one of the most widespread standards for the exchange of data between processes on parallel computing systems. Its main application is the development of programs performing parallel computations and it is therefore employed on the majority of supercomputers. The performance and scalability of applications running on these high-performance computers is thus directly determined by the underlying *MPI* implementation, which is provided by various vendors. In *MPI*, so-called collective operations are used for the movement of data in a group of processes. The efficiency of these predefined operations depends on the underlying algorithm, which varies from vendor to vendor. In principle, the user should always attempt to find the best performing algorithm for the specific use case. Many vendors offer ways to fine-tune the algorithms manually by changing parameters. Several approaches to tuning the parameters have been proposed, including the approach of Simone Pellegrini et al. [1] who used the *Anova Variance Analysis* to tune the parameters according to the underlying system. Hunold et al. [2] used a differential approach to find the best algorithm for a use case, in which instead of tuning parameters, a latency comparison of a defined *MPI* collective operation and an equivalent function, also called a *mockup-up*, is performed. This approach led to the development of the *PGMPITuneLib* library, which intercepts *MPI* calls and forwards them to specially implemented *mockup-ups*. The library can also be used to analyze the runtimes of various *mockup-up* implementations. In order to evaluate violations against the performance guidelines in an automated and more detailed way, this thesis will focus on the processing and analysis of the collected runtimes.

1.1 Motivation

We will take a look at a specific *MPI* collective operation, called *MPI_Bcast*. With this operation, data is sent from one process to all the other processes belonging to the same group. A visual representation of the *MPI_Bcast* collective can be seen in Figure 1.1.

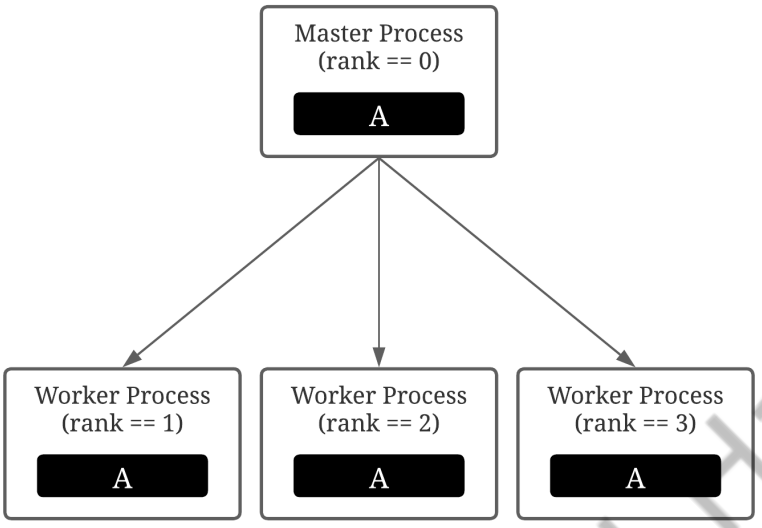

Figure 1.1: The collective operation *MPI_Bcast*.

In this particular case, a *master* process that broadcasts data to three other processes is used. We refer to the main process as *master*, which is assigned rank zero by *MPI*. We call all other involved processes *worker* and assign them arbitrary ranks. After the operation is complete, all processes own the same set of information. This operation can either be performed with the predefined *MPI_Bcast* operation, but also so-called *mockup-up* implementations consisting of one or more other collectives can be used to achieve the same result. In the case of *MPI_Bcast*, other collectives like *MPI_Allgatherv* or a combination of *MPI_Scatter* and *MPI_Allgather* could also be used. The visual representation of the *MPI_Allgatherv* function can be seen in Figure 1.2. In this function, the *master* process owns the data to be sent to all other processes. The *MPI_Allgatherv* function collects the data from all processes and distributes it to all processes. Thus, it is irrelevant which data the other processes possess, since the data of the *master* process is made available to all processes as well. Therefore it can be stated, that exactly as with *MPI_Bcast* after the operation all processes possess the data of the *master* process.

In Figure 1.3 the visual representation of the *mockup-up* consisting of *MPI_Scatter* and *MPI_Allgather* can be seen. In this case, two operations have to be executed in a row to get the same result as for the *MPI_Bcast* operation. First, the data of the *master* process is distributed evenly to all other processes using the *MPI_Scatter* function. After the operation, each process possesses an equally large packet of data. With *MPI_Bcast*, however, all data of the *master* process should be available to all other processes. For this purpose, the operation *MPI_Allgather* can be used, because this operation collects data packets of the same size from all processes and makes them available to all processes. For this mockup, we can also state that after applying these two collective operations, the

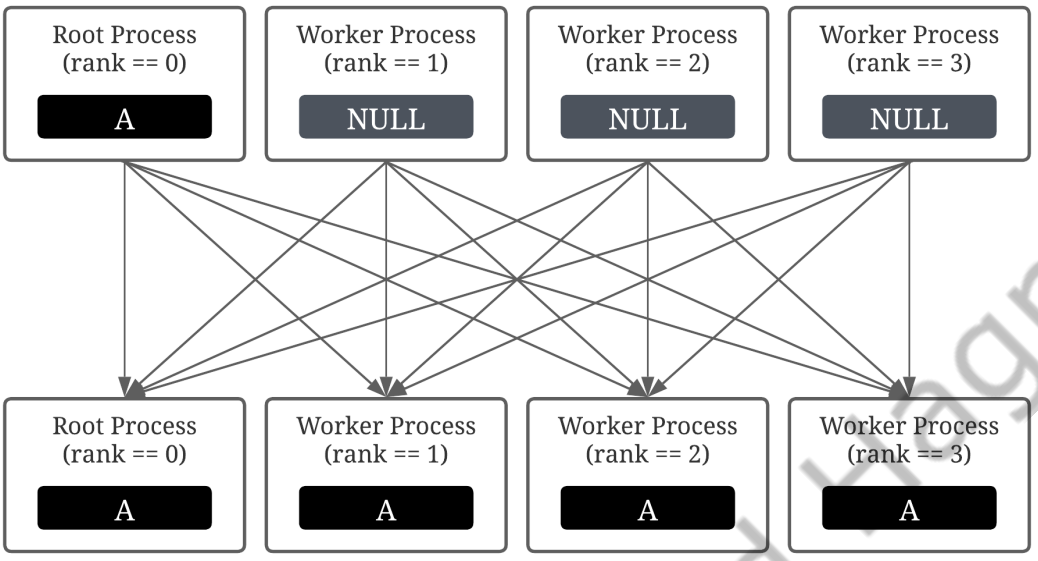

Figure 1.2: The collective operation *MPI_Allgatherv*.

data of the *master* process is available to all processes. In conclusion, we could recognize, that all mentioned implementations lead to the same information set. In the following we will refer to the standard implementation, in this case the *MPI_Bcast* collective, as *default* implementation, and to the alternative collectives that achieve the same result as *mockup-up* implementations.

We could observe that different *default* collective operations can also be represented by their *mockup-up* implementations. The next step is to consider a selected set of the self-consistent performance guidelines proposed by Träff et al. [3]. Briefly, the *MPI* performance guidelines define the characteristics that must be met in terms of runtime in order for an *MPI* implementation to perform at a high level of quality. For instance, according to Träff et al. [3] the runtime should not be reduced by replacing a specific collective operation with a sequence of multiple operations. By applying this rule, we notice that the combination of *MPI_Scatter* and *MPI_Allgather* should not be faster than the *default MPI_Bcast* implementation. Another rule defined by Träff et al. [3] states that the runtime should not be reduced if a specific operation is replaced by a more general operation that produces the same result. Since *MPI_Bcast* is specific and *MPI_Allgatherv* is a more general form, *MPI_Allgatherv* should not be faster than the *default* implementation of *MPI_Bcast*. It can thus be stated that if the *default MPI_Bcast* implementation has the lowest runtime of all implementations, the rules of the guidelines hold. More formally, we can state these rules as follows:

$$\text{MPI\_Bcast(n)} \preceq \text{MPI\_Allgatherv(n)}, \text{ and}$$

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

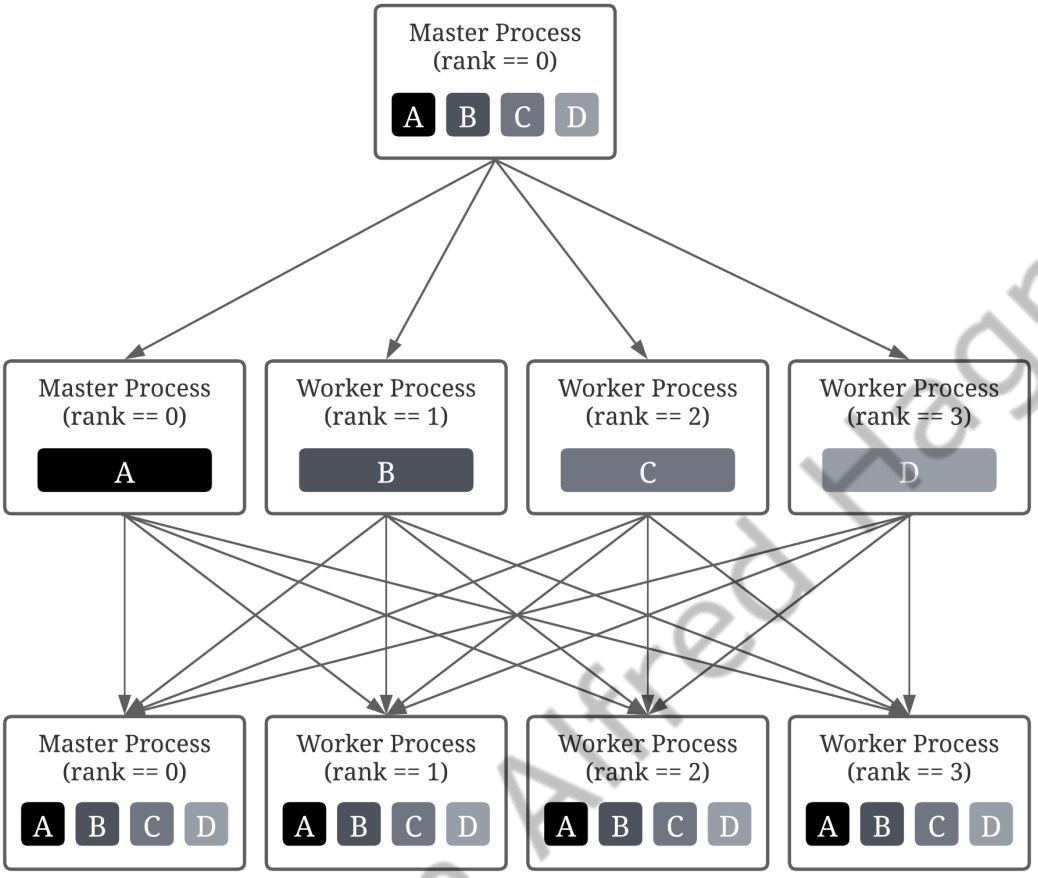

Figure 1.3: One *mockup-up* algorithm for *MPI_Bcast* from a sequence consisting of the collective operations *MPI_Scatter* and *MPI_Allgather*.

Hunold et al. [2] developed the *PGMPITuneLib* library, which allows selected collective operations to be automatically replaced by their *mockup-up* implementations. For the automatic tuning of the implementations, one of the steps to be performed is the measurement of the runtimes of each variant. In this work, we will use the data obtained in this step to develop an application called *pgchecker*, which can be used to check the quality of existing *MPI* implementations from different vendors. In our *MPI_Bcast* example, we will now exploit the *PGMPITuneLib* library by running the various *default* and *mockup-up* implementations *nrep* times and measuring them. Our *pgchecker* extension uses these measurements to generate a table as shown in Table 1.1. In addition to the information on the tested collective operation, the message size, the number of nodes, the number of processes and the number of measurements, the table also contains the median runtimes. The last row of the table shows that *pgchecker* found a violation against the guidelines for a message size of 16 KiB. The fastest *mockup-up* implementation including

Table 1.1: The desired output of the *pgchecker* tool.

collective	m_size	N	ppn	nrep	def_median	mockup	mock_median
Bcast	1 KiB	1	12	500	0.0045 ms		
Bcast	4 KiB	1	12	500	0.0099 ms		
Bcast	16 KiB	1	12	500	0.0350 ms	Allgatherv	0.0249 ms

the median runtime of this variant is displayed. In this case, the runtime of the *mockup-up* implementation with the collective operation *MPI_Allgatherv* is shorter than that of the *default* variant. The tool could therefore be used to automatically detect a violation against the *MPI* performance guidelines.

1.2 Goal of the Thesis

The objective of this thesis is to develop a *C++* application that can automatically verify collective operations of *MPI* implementations from different vendors for conformance to the *MPI* performance guidelines. In the following, we will call the developed application *pgchecker*. Our *pgchecker* tool is based on the library *PGMPITuneLib*, developed by Hunold et al. [4], which allows us to automatically replace *default* algorithms by their *mockup-ups* and to determine the execution time of these variants. When executing *pgchecker*, this library is automatically accessed and the runtimes of the different algorithms are measured first. With the obtained data we will then perform various comparisons. For instance, it is possible to display all collected data, or to display a clear table with only the violations and the violating *mockup-ups*, like the table shown in Table 1.1. Furthermore, it is possible to write the results to the command line or to files in predefined folders to allow further analysis. Moreover, it is also possible to choose between different statistical tests used to calculate the violations. Besides these options, there are other possibilities, such as merging the obtained data tables. Lastly, it can be mentioned that all parts of the *pgchecker* tool have been developed with further customization in mind. This means that it is possible for the user to develop more comparisons or add further statistical tests easily.

Another objective of the thesis is to demonstrate that our tool can correctly generate violations from the raw runtime data. For this purpose, we will compare the output generated by our *pgchecker* tool with diagrams generated from this raw data. Subsequently, we will utilize our *pgchecker* tool to analyze different major *MPI* implementations. The analysis is performed on two systems: First, the so-called *Hydra* system is equipped with $N = 36$ compute nodes, each with $n = 32$ cores per node. The second so-called *Irene* system utilizes $N = 1656$ nodes with $n = 48$ cores each. First, we would like to interoduce the workflow of analyzing with *pgchecker* on the *Hydra* system using the *Open MPI* [5] implementation. This example is intended to demonstrate the user how *pgchecker* can be applied to analyze various implementations and which options may be suitable for different use cases. Subsequently, we will analyze the *MPI* implementations *MPICH* [6]

and *MVAPICH* [7] on *Hydra* utilizing different numbers of compute nodes and cores per node. Finally, we will use the *Irene* system to analyze the *Intel ®MPI* [8] and the *Open MPI* implementation. For each implementation, we will look closely at how many violations against the *MPI* performance guidelines occur, which collective operations perform best and which perform worst, and how the number of processes used affects the performance of the implementations.

1.3 Structure of the Work

This thesis starts with a brief introduction to the *MPI* standard. Therefore, we will first clarify the basics of *MPI* and introduce the point-to-point communication. Afterwards, we address the most important collective operations. Furthermore, in Chapter 2, we will take a closer look at the self-consistent *MPI* performance guidelines proposed by Träff et al.[3]. At the end of Chapter 2, we will introduce the *PGMPITuneLib* library developed by Hunold et al. [4], as it is essential for the development of our *pgchecker* tool. In Chapter 3, we will concentrate on the development of the *pgchecker* tool and discuss its functionality as well as the possible configuration options. We will mainly focus on the different types of comparisons as well as on the implemented statistical tests. As an introduction to Chapter 4, we will walk through a complete example explaining the analysis workflow utilizing the developed *pgchecker* tool. After learning about the workflow, we will apply it to analyze some *MPI* implementations from different vendors for violations against the performance guidelines. First, we will examine the implementations on the smaller *Hydra* system for a smaller number of processes, before we use the supercomputer *Irene* to examine the implementations at a larger number of processes. Finally, in Chapter 5, we will summarize the results of our research, consider the limitations of the developed *pgchecker* tool, and explain how users can extended and customized it.

CHAPTER 2

Theoretical Foundations and Related Work

2.1 Introduction to Parallel Computing Utilizing the Message Passing Interface

The *Message Passing Interface (MPI)* [9], first published in 1995, provides an interface for communication between processes in parallel applications running on one or several systems [10]. It is especially important to mention that *MPI* does not offer the opportunity to manage resources, for instance to create or destroy processes, which is intended to maintain portability. More essentially, it defines the possible communication functions between already generated processes. Each process is assigned its own rank by which it can be uniquely identified. The most fundamental way of message exchange is therefore the point-to-point communication, where one process may send data to another process, and this process may receive the data. Furthermore, it is also possible to group the existing processes and assign them to so-called communicators. *MPI* also defines several functions for this purpose, such as union, intersection, or difference of groups. Within these groups or communicators, the collective operations, such as *MPI_Bcast*, *MPI_Gather*, or *MPI_Scatter*, may be used. In our short introduction to *MPI*, we will first look at the basic structure of an *MPI* program. Afterwards, we will discuss an instance of point-to-point communication before looking at groups, communicators and collective operations.

2.1.1 The Basic Structure of a MPI Program

Figure 2.1 shows the basic structure of a program in *MPI*. The first step is always to include the *MPI* header file, which allows the user to access the predefined functions and data types. Furthermore, there are two functions that are needed in every *MPI*

Listing 2.1: The basic structure of a *MPI* program in *C++*.

```
#include <mpi.h>

int main(int argc, char** argv) {

    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int comm_size;
    MPI_Comm_size(MPI_COMM_WORLD, &comm_size);

    // other mpi functions

    MPI_Finalize();

    return 0;
}
```

program, `MPI_Init()` and `MPI_Finalize()`. The first one signals that from this point on calls to *MPI* functions can be made, whereas the latter one indicates that no more *MPI* functions can be called. The function `MPI_Comm_rank()` is used very often since it is used to detect the rank of the current process. The function `MPI_Comm_size()` further indicates how many processes are in the given communicator. This is especially useful if, for instance, a number of t computational tasks should be divided among p processes. The first parameter of the functions, which is filled in with the keyword `MPI_COMM_WORLD`, stands for a communicator in which all created processes can be found. However, another previously created communicator could also be used instead. The second parameter for both functions specifies the memory area in which the returned values should be stored.

Since *MPI* is a standard that can be implemented in several programming languages, we will aim to use the general definitions from the official *MPI* document [9] for the explanations and avoid going into the programming language-specific data types and properties in general. We will, however, refer to incoming parameters with $^{\text{In}}$ and outgoing parameters with $^{\text{Out}}$ to increase the clarity of the description of the functions.

2.1.2 Point-to-Point Communication in MPI

Having introduced the basic structure of a *MPI* program, we can focus on the different forms of communication. The point-to-point communication is the most basic form of communication defined in *MPI*, where a single process sends a message to another process. In order to realize the communication, the sending process must execute the function *MPI_Send*: `MPI_Send(buf`$^{\text{In}}$`, count`$^{\text{In}}$`, datatype`$^{\text{In}}$`, dest`$^{\text{In}}$`, tag`$^{\text{In}}$`, comm`$^{\text{In}}$`)`. The pointer `buf` points at the starting point of the data to be sent, whereas `count` specifies how many elements to send from this starting address. The variable of type

Figure 2.1: The basic point-to-point communication in *MPI*.

`MPI_Datatype` can be used to define one of the data types available in the *MPI* standard. Among others, these could be `MPI_INT`, `MPI_DOUBLE`, or `MPI_BYTE`. The integer value `dest` is used to determine the rank of the receiving process. Optionally a tag can be specified, which is used to distinguish between different messages. Finally, the communicator containing the two processes involved in the message exchange is determined by `comm`.

The receiving process on the other end must execute the function *MPI_Recv*: `MPI_Recv (buf`$^{\text{Out}}$`, count`$^{\text{In}}$`, datatype`$^{\text{In}}$`, source`$^{\text{In}}$`, tag`$^{\text{In}}$`, comm`$^{\text{In}}$`, status`$^{\text{Out}}$`)`. The integer value `source` is the counterpart of `dest` and describes the process from which the message is to be received. If a tag is given for the function *MPI_Send*, it must also be specified for receiving. In addition to the already known options, a memory area for the current status of the operation must also be created for receiving. Furthermore, it should be noted that the communication is only successful if the memory area of the receiving process is larger or equal to the size of the data that is sent.

A visual representation of the point-to-point communication can be seen in Figure 2.1. We would like to build this example into an executable *C++* application. For this purpose, we use the basic framework presented in Listing 2.1 and extend it with the previously introduced functions. Listing 2.2 shows the finished program code. First, the rank of the current process is checked and based on this rank a case distinction is made. If the current process has rank zero, the information to be transmitted is set to the integer value 18. Before and after the transmission a message is displayed on the command line with the aim of better understanding which action is currently being executed. However, if the process has rank one, the variable containing the data is set to zero and the receiving process is called. Again, messages are displayed on the console. After the transmission of the number 18 is completed, the process with rank one should output this number.

One possible output of the execution can be seen in Listing 2.3. There are other outputs probable since we do not know which process will be active first. In this case, the process with rank zero becomes active first and starts sending the data. It can be observed that the process terminates before the data has been received. In general, *MPI* only requires that the process waits until the data memory can be written again without risk. As a side note, there are other versions of the *MPI_Send* and *MPI_Recv* functions that allow blocking. Furthermore, there are already combined functions that allow simultaneous sending and receiving, as well as the possibility to check the current status

Listing 2.2: The *MPI_Send* and *MPI_Recv* example program in *C++*.

```
#include <mpi.h>
#include <iostream>

int main(int argc, char** argv) {

    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {

        int data_in = 18;

        std::cout << "Process 0 starts sending." << std::endl;
        MPI_Send(&data_in, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        std::cout << "Process 0 returns from sending." << std::endl;

    } else if (rank == 1) {

        int data_out = 0;
        std::cout << "Process 1 data: " << data_out << std::endl;
        std::cout << "Process 1 starts receiving." << std::endl;

        MPI_Recv(&data_out, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            ↪ MPI_STATUS_IGNORE);
        std::cout << "Process 1 returns from receiving." << std::endl;
        std::cout << "Process 1 received data: " << data_out << std::endl;

    }

    MPI_Finalize();

    return 0;

}
```

Listing 2.3: One possible output for the *MPI_Send* and *MPI_Recv* example program.

```
Process 0 starts sending.
Process 0 returns from sending.
Process 1 data: 0
Process 1 starts receiving.
Process 1 returns from receiving.
Process 1 received data: 18
```

of the transmission. In our example, the process with rank one becomes active next. The process first outputs the current value of the variable and then receives the message. After receiving the message, the number 18 is written to the data memory and the program executed as desired.

2.1.3 Communicators and Collective Operations

As mentioned above, there are operations in *MPI* that can be performed within a group or a communicator. In the previous section, for the communicator specification we mainly used the keyword `MPI_COMM_WORLD`, that describes a communicator in which all started processes can be found. For completeness, it should be mentioned that there are several functions that can be used to create new communicators or to generate new ones from existing communicators. An example of such a function would be *MPI_Comm_split*: `MPI_Comm_split(comm`$^{\text{In}}$`, color`$^{\text{In}}$`, key`$^{\text{In}}$`, newcomm`$^{\text{Out}}$`)`. The first parameter describes the communicator from which the new one should be generated. The variable `color` defines the communicator into which the current process is to be classified. Furthermore, a `key` can be specified on which the sequence in the new communicator is based. Finally, the last parameter refers to the communicator to be created. The new communicator can be utilized safely after each process has called the function and has been added to the new communicator. However, the original communicator can still be used. In addition to this function, many others are offered by *MPI*, which we will not cover in detail.

In communicators, so-called collective operations can be executed that involve multiple processes. For instance, *MPI_Bcast* is employed to send data from a *master* process to all *worker* processes in the aggregation. The *MPI_Bcast* operation could also be performed using point-to-point communication, where the *master* process sends $m = |worker|$ messages and each *worker* process receives one message. However, since this variant is not optimized and the runtime is correspondingly worse, special algorithms and approaches are used in the background to make the collective operation more efficient. In addition to the *MPI_Bcast* that we have already discussed, there are many other operations. For instance, some frequently utilized operations are: *MPI_Allgather*, *MPI_Allreduce*, *MPI_Alltoall*, *MPI_Barrier*, *MPI_Scan*, *MPI_Gather*, *MPI_Reduce*, *MPI_Reduce_scatter*, *MPI_Reduce_scatter_block*, *MPI_Exscan*, and *MPI_Scatter*. These operations are of particular importance in this thesis on the grounds that they are implemented in the *PGMPITuneLib* library, which will allow us to utilize *pgchecker* to test these operations for violations against the guidelines. In Chapter 1, we have already explained in detail how the *MPI_Bcast* operation can be applied. Additionally, in the following we intend to take a closer look at *MPI_Barrier*, *MPI_Gather*, *MPI_Reduce*, and *MPI_Scatter* since the analysis conducted in Chapter 4 will also be carried out on the basis of these collective operations.

The *MPI_Scatter* operation is quite similar to the *MPI_Bcast* operation. For the *MPI_Scatter* operation, the *master* process holds an arrangement of data. A portion of this data is sent to one process each. The *MPI_Scatter* function is defined by

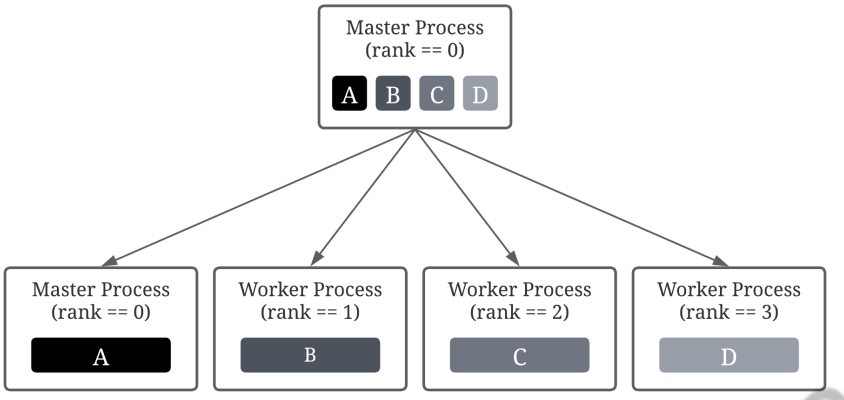

Figure 2.2: The collective operation *MPI_Scatter*.

MPI as follows: `MPI_Scatter(sendbuf`^{In}`, sendcount`^{In}`, sendtype`^{In}`, recvbuf`^{Out}`, recvcount`^{In}`, recvtype`^{In}`, root`^{In}`, comm`^{In}`)`. The first three parameters defines the memory area of the message to be sent, including the number of elements and the data type. Moreover, it is specified where the received data is to be stored and how many elements are to be received and what data type they have. Finally, the sending process is determined and the communicator in which the operation is to be performed. A visual representation can be seen in Figure 2.2. At the beginning, the *master* process holds the four data fields `A`, `B`, `C`, and `D`. These data fields are then distributed to all processes in this network by the *MPI_Scatter* function. After each process has performed the operation, the process with rank zero holds the data `A`, the process with rank one holds the data `B`, process three holds the data `C`, and the process with rank four holds the data `D`. Furthermore, it can be noted that the *master* process still possesses the data fields `A`, `B`, `C`, and `D` if these have not been overwritten by the receive operation. Using the function *MPI_Scatter*, it is important to note that the data held by the *master* process is divided into packets of equal size and each of the *worker* processes thus receives data with size $size = total\_message\_size/|processes|$. If the packets should not be split into equal sized parts, *MPI* offers the possibility to use a vectorized form. In the instance of *MPI_Scatter* this function would be *MPI_Scatterv*. With this function instead of the already known `sendcount` the standard provides a vector `sendcounts`, which allows to describe the number of elements for each process. Additionally, we can also apply the option `displs` using this function, with which a vector of displacements can be specified.

For the collective operation *MPI_Scatter*, we would like to show a simple example in *C++* with the aim of better understanding the functionality of an executable *MPI* program utilizing collective operations. In Listing 2.4, the code of the application can be observed. The array `in` contains the same data values that can be seen in Figure 2.2. After the array is filled by the *master* process, each process executes the function *MPI_Scatter*, where each process should get exactly one element of the array `in` and should store it

Listing 2.4: The *MPI_Scatter* example program in *C++*.

```
#include <mpi.h>
#include <iostream>

int main(int argc, char **argv) {

  MPI_Init(&argc, &argv);

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  char in[4];

  if (rank == 0) {

    in[0] = 'A';
    in[1] = 'B';
    in[2] = 'C';
    in[3] = 'D';

  };

  char out;

  MPI_Scatter(&in, 1, MPI_CHAR, &out, 1, MPI_CHAR, 0, MPI_COMM_WORLD);

  std::cout << "Process " << rank << " received " << out << std::endl;

  MPI_Finalize();

  return 0;

}
```

Listing 2.5: One possible output for the *MPI_Scatter* example program.

```
Process 1 received B
Process 3 received D
Process 0 received A
Process 2 received C
```

in `out`. After the operation has been performed, each process should have exactly one character stored in the variable `out`. A possible output can be seen in Listing 2.5.

MPI_Gather is an operation to collect data from multiple processes, making them available to exactly one process. The *MPI_Gather* function is defined by the *MPI* standard as follows: `MPI_Gather(sendbuf`$^{\text{In}}$`, sendcount`$^{\text{In}}$`, sendtype`$^{\text{In}}$`, recvbuf`$^{\text{Out}}$`, recvcount`$^{\text{In}}$`, recvtype`$^{\text{In}}$`, root`$^{\text{In}}$`, comm`$^{\text{In}}$`)`. It is apparent that this function has

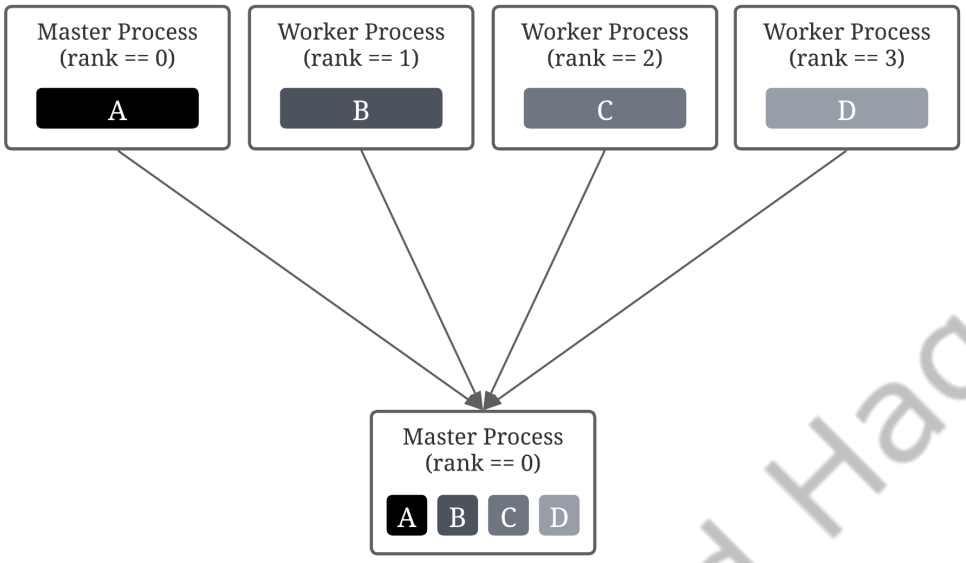

Figure 2.3: The collective operation *MPI_Gather*.

the same parameters as the already known function *MPI_Scatter*. A visual representation can be seen in Figure 2.3. In other words, each process posses a set of data with possibly different information, whereas after the operation one process has the data of all others. For the *MPI_Gather* function, there is again a vectorized variant *MPI_Gatherv*, where it can be specified which process should receive how many elements. Furthermore, it should be mentioned that there are also variants for some *MPI* functions which are labeled by the prefix *All*. In the instance of *MPI_Gather* and *MPI_Gatherv*, this would be *MPI_Allgather* and *MPI_Allgatherv*. The variants have the same purpose, which is to gather data from all processes, but the gathered data will be available to all processes. *MPI_Allgatherv* has already been discussed in Chapter 1, and the graphical representation is shown in Figure 1.2. Since we have discussed *MPI_Scatter* and *MPI_Gather*, we would like to note that there is also a collective operation called *MPI_Alltoall* which is responsible for making data from all processes available on all processes.

The *MPI_Reduce* operation is quite similar to the *MPI_Gather* operation. The function is defined as follows: `MPI_Reduce(sendbuf`$^{\text{In}}$`, recvbuf`$^{\text{Out}}$`, count`$^{\text{In}}$`, datatype`$^{\text{In}}$`, op`$^{\text{In}}$`, root`$^{\text{In}}$`, comm`$^{\text{In}}$`)`. As with *MPI_Gather*, this function is also intended to collect information from several processes. In addition to the already known parameters, it is noticeable that this time an operation can be specified for the reduction by `op`. For instance, by using the *MPI_MAX* operation only the largest element in the root process becomes available, whereas by specifying *MPI_SUM* the sum of all elements becomes available. A visual representation can be seen in Figure 2.4. In the visual representation, it can be observed that the operator *MPI_MAX* should be applied. The processes first

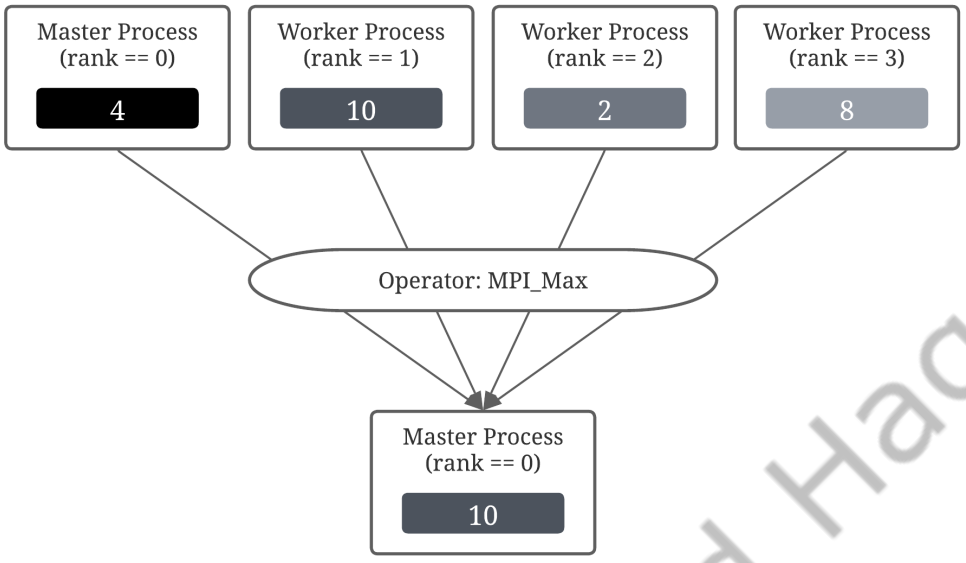

Figure 2.4: The collective operation *MPI_Reduce*.

have different integer values. After the operation, one process has the largest element. This function is also available in the *MPI_Allreduce* variant in which the collected information is made available to all processes.

The last operation, we want to analyze, is *MPI_Barrier*, which is defined as follows: `MPI_Barrier(comm`$^{\text{In}}$`)`. This operation is of particular importance as it enables processes to be synchronized in *MPI*. The only parameter of this function is the communicator in which the synchronization should be performed. Figure 2.5 shows how the synchronization process works. If a process reaches the *MPI_Barrier* function, it waits until all the other processes in the specified communicator have also reached the function. If the function has been called by all processes in the communicator, they can proceed with their work. We will discuss the *MPI_Barrier* operation in more detail in Chapter 3 on the grounds that we need the runtime of this operation to issue warnings if a violation should be considered cautious.

We have considered the four basic collective operations of *MPI*, and on closer observation it becomes apparent that many operations can also be used in conjunction with each other to represent other operations. For instance, in Chapter 1, we have already noted that *MPI_Bcast* can also be represented by *MPI_Scatter* and thus we can obtain *mockup-ups* for each operation whose runtime can be compared with the *default* implementations. Therefore, we would like to further explain the *MPI* performance guidelines that must be followed when replacing *default* algorithms of collective operations by their *mockup-ups*.

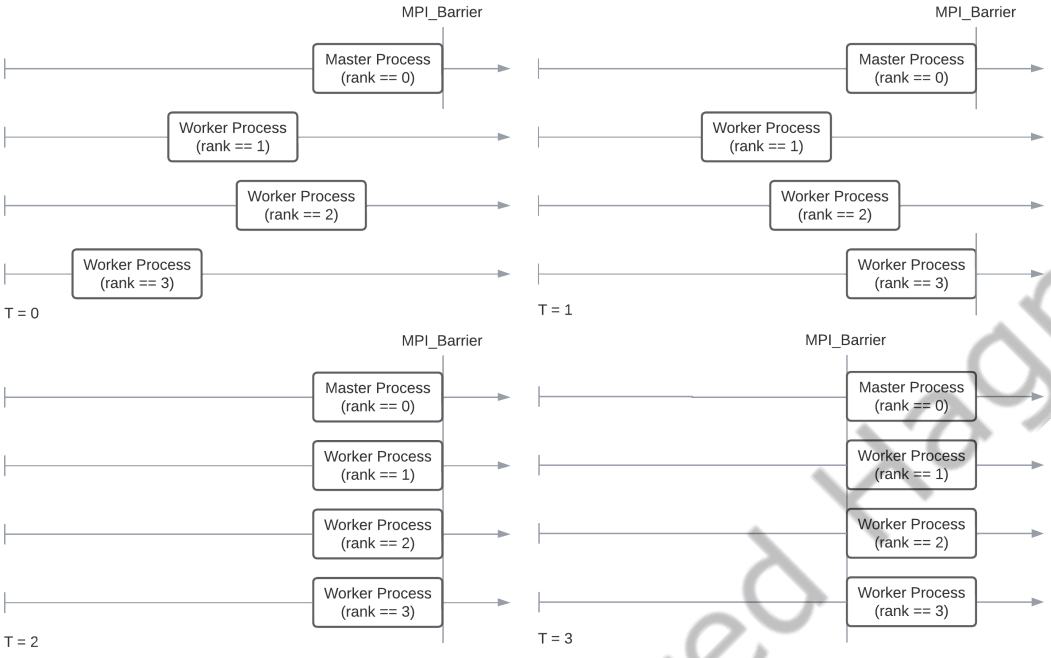

Figure 2.5: The collective operation *MPI_Barrier*.

2.2 An Overview of the Self-Consistent MPI Performance Guidelines

In our introduction to *MPI*, we have already discussed the substitutability of several collective operations. In this section, the focus will be on the self-consistent performance guidelines proposed by Träff et al. [3] and the extension by pattern guidelines of Hunold et al. [2, 4]. In the related work regarding this topic, the main objective is to determine which collective operations must be performed faster than the other ones. For instance, the *MPI_Bcast* operation must possess a shorter runtime than the alternative point-to-point variant, but also than the *mockup-up* consisting of *MPI_Scatter* and *MPI_Allgather*.

Träff et al. [3] defined the following notation for performance guidelines: If there are two *MPI* functions A and B, they can be expressed in relation

$$MPI\_A(n) \preceq MPI\_B(n)$$

if and only if the functionality $MPI\_A(n)$ has no greater execution time than $MPI\_B(n)$. This means that the *MPI* functionality A is possibly faster than B for (almost) all communication amounts n. Furthermore, $\approx$ is used to express that the two functionalities

$$MPI\_A(n) \approx MPI\_B(n)$$

perform similarly for almost all communication amounts n.

As well as the notation, Träff et al. [3] defined metarules specifying the regulations to which *MPI* implementations must be conform: First, the runtime must not be shortened by splitting a message into several smaller messages. Second, the runtime must not be reduced by replacing one *MPI* function with other functions that contain additional meanings. Third, the runtime must not be reduced by replacing a specific operation, meaning an operation that has been developed for exactly this use case, by a general function that yields to the same result. Fourth, the runtime must not be reduced by replacing a *MPI* function with a sequence of other operations. We have already applied this rule in Chapter 1 and Chapter 2 and it is of particular importance for our analysis of violations. Rules five and six refer to the ordering of processes. These state that the communication time must not be reduced by rearranging the processes in a communicator and that the creation of a virtual topology must not reduce the communication time.

In the development of *PGMPITuneLib*, Hunold et al. [2] distinguished between three major classes of performance guidelines: monotony, split-robustness, and pattern. The monotony rule states that the runtime should not be reduced by increasing the amount of data transferred. More formally, this can be written as

$$MPI\_A(n) \preceq MPI\_A(n+k) \quad .$$

The split-robustness guideline rule specifies that the runtime should not be reduced when a message is split into several smaller messages:

$$MPI\_A(n) \preceq k * MPI\_A(n/k) \quad ,$$

where k is the number of fragmented messages and n is the total amount of data. Therefore, this rule is derived directly from the first metarule of Träff et al. [3].

The pattern rule explicitly refers to the substitution of functionalities against other functionalities. Thus, it is determined by definition that both *MPI* functions within the relationship must yield the same result, such as *MPI_Bcast* and *MPI_Allgatherv*. In Chapter 1, we have already formulated two pattern rules:

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

The first rule states that the runtime of *MPI_Allgatherv* should be greater than the runtime of *MPI_Bcast*. The second rule states that the runtime of the *default* implementation of *MPI_Bcast* should not be reduced by replacing it with the *mockup-up* implementation consisting of the sequence of *MPI_Scatter* and *MPI_Allgather*. We can thus state that pattern guidelines define an ordering of the particular *default* implementations and *mockup-up* implementations of *MPI* collective operations. So far, we have only considered the pattern rules for the *MPI_Bcast* operation. During the development

$$\text{MPI\_Allgather(n)} \preceq \text{MPI\_Gather(n)} + \text{MPI\_Bcast(n)}, \tag{2.1}$$
$$\text{MPI\_Allgather(n)} \preceq \text{MPI\_Alltoall(n)}, \tag{2.2}$$
$$\text{MPI\_Allgather(n)} \preceq \text{MPI\_Allreduce(n)}, \tag{2.3}$$
$$\text{MPI\_Allgather(n)} \preceq \text{MPI\_Allgatherv(n)}, \tag{2.4}$$

$$\text{MPI\_Allreduce(n)} \preceq \text{MPI\_Reduce(n)} + \text{MPI\_Bcast(n)}, \tag{2.5}$$
$$\text{MPI\_Allreduce(n)} \preceq \text{MPI\_Reduce\_scatter\_block(n)} + \text{MPI\_Allgather(n)}, \tag{2.6}$$
$$\text{MPI\_Allreduce(n)} \preceq \text{MPI\_Reduce\_scatter(n)} + \text{MPI\_Allgatherv(n)}, \tag{2.7}$$
$$\text{MPI\_Alltoall(n)} \preceq \text{MPI\_Reduce\_scatter(n)} + \text{MPI\_Alltoallv(n)}, \tag{2.8}$$

$$\text{MPI\_Bcast(n)} \preceq \text{MPI\_Allgatherv(n)}, \tag{2.9}$$
$$\text{MPI\_Bcast(n)} \preceq \text{MPI\_Scatter(n)} + \text{MPI\_Allgather(n)}, \tag{2.10}$$

$$\text{MPI\_Gather(n)} \preceq \text{MPI\_Allgather(n)}, \tag{2.11}$$
$$\text{MPI\_Gather(n)} \preceq \text{MPI\_Gatherv(n)}, \tag{2.12}$$
$$\text{MPI\_Gather(n)} \preceq \text{MPI\_Reduce(n)}, \tag{2.13}$$

$$\text{MPI\_Reduce(n)} \preceq \text{MPI\_Allreduce(n)}, \tag{2.14}$$
$$\text{MPI\_Reduce(n)} \preceq \text{MPI\_Reduce\_scatter\_block(n)} + \text{MPI\_Gather(n)}, \tag{2.15}$$
$$\text{MPI\_Reduce(n)} \preceq \text{MPI\_Reduce\_scatter(n)} + \text{MPI\_Gatherv(n)}, \tag{2.16}$$

$$\text{MPI\_Reduce\_scatter\_block(n)} \preceq \text{MPI\_Reduce(n)} + \text{MPI\_Scatter(n)}, \tag{2.17}$$
$$\text{MPI\_Reduce\_scatter\_block(n)} \preceq \text{MPI\_Reduce\_scatter(n)}, \tag{2.18}$$
$$\text{MPI\_Reduce\_scatter\_block(n)} \preceq \text{MPI\_Allreduce(n)}, \tag{2.19}$$

$$\text{MPI\_Scan(n)} \preceq \text{MPI\_Exscan(n)} + \text{MPI\_Reduce\_local(n)}, \tag{2.20}$$

$$\text{MPI\_Scatter(n)} \preceq \text{MPI\_Bcast(n)}. \tag{2.21}$$
$$\text{MPI\_Scatter(n)} \preceq \text{MPI\_Scatterv(n)}. \tag{2.22}$$

of *PGMPITuneLib*, Hunold et al. [2] specified several additional rules that can also be tuned by the library, these are listed in Equation 2.1 to Equation 2.22.

To conclude, it can be expressed that the pattern rules in particular are of significant importance for our *pgchecker* tool. We want to use our tool to check exactly these rules for violations and we will therefore compare the *default* implementation, which is the one on the left side of the relation, with the right side, which is the *mockup-up* variant. After getting an overview of the performance guidelines and understanding which rules we want to verify with *pgchecker*, we can go into more detail about the *PGMPITuneLib* library which already provides some functionality for exchanging and measuring various implementations.

2.3 Introduction to the Software Framework Utilized by *pgchecker*

The *PGMPITuneLib* library is an interface developed by Hunold et al. [4] as an alternative approach to tuning an *MPI* library using parameters. In principle, different algorithms are used in *MPI* implementations from different vendors and the strategy for selecting the best algorithm is usually predefined. Some providers, such as *Open MPI* [5], offer users the possibility to fine-tune the algorithms by allowing parameters to be customized. However, the selection of these parameters is a challenging task, which is further complicated by differences in the environments and by the need to tune a collective operation in isolation. Hunold et al. [2] applied a different approach, one where no parameters are chosen, but the whole collective operation is replaced by a *mockup-up* variant. The *mockup-up* implementations are defined by Equation 2.1 to Equation 2.22 of the self-consistent *MPI* performance guidelines. This approach is particularly beneficial because it can always be applied, even if the *MPI* implementation provider does not offer the possibility of fine tuning [2]. The collective operations must have been implemented through the standard *MPI* anyway. It is worth mentioning that Träff and Hunold [11] have implemented other alternative algorithms in *PGMPITuneLib* as well. For instance, algorithms exists that take the architecture of different parallel machines into account.

First, we would like to explain the architecture of *PGMPITuneLib*. *PGMPITuneLib* is located between the *MPI* user code and the respective *MPI* library. Therefore, the possibility is given to intercept *MPI* collective operation calls and to replace them with *mockup-up* implementations if present. Sticking to our *MPI_Bcast* instance, in such a scenario *PGMPITuneLib* would intercept the call to *MPI_Bcast* and replace it with one or more passes through all the already defined *mockup-ups*. However, if no *mockup-up* is found by the library for a collective operation, it can still redirect to the *default* implementation. This process is visually represented in Figure 2.6 [4]. It can be recognized that *PGMPITuneLib* reads the best *mockup-up* implementation from the tuning profile. After it has received the information, the individual operations of the *mockup-up* are performed by using the respective *MPI* library. *PGMPITuneLib* offers two different modes, both of which have the task of intercepting *MPI* calls and forwarding

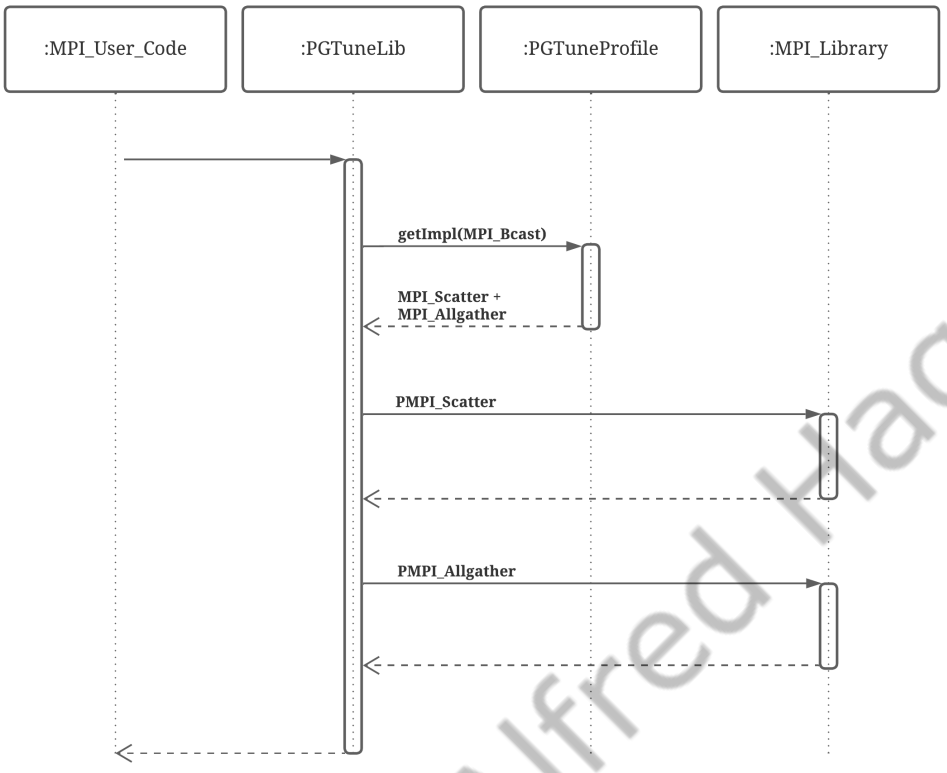

Figure 2.6: The functionality of the *PGMPITuneLib* library.

Listing 2.6: The *PGMPITuneLib* statement for gathering data for the *default MPI_Bcast* implementation.

```
./bin/pgchecker
    --msizes-list=1,2,4
    --nrep=10
    --calls-list=MPI_Bcast
    --output-file=./data/data_MPI_Bcast_default.dat
    --module=bcast=alg:default
```

them to *mockup-up* implementations. We will mainly build out our *pgchecker* tool on the *PGMPITuneCLI*, which is the command line utility of *PGMPITuneLib*. This mode allows us to create performance profiles of the individual *mockup-ups* and therefore we are able to gather information about the runtimes of the *mockup-ups* for our *pgchecker* tool. The second mode called *PGMPITuneD* is used after the analysis to load the previously created performance profiles and thus tune the *MPI* program written by the user. For simplicity, in this thesis, we will reference all modes as *PGMPITuneLib*.

Listing 2.7: The *PGMPITuneLib* statement for gathering data for the *MPI_Allgatherv mockup-up* implementation.

```
./bin/pgchecker
    --msizes-list=1,2,4
    --nrep=10
    --calls-list=MPI_Bcast
    --output-file=./data/data_MPI_Bcast_bcast_as_allgatherv.dat
    --module=bcast=alg:bcast_as_allgatherv
```

For our *pgchecker* tool, this means that we can link the *PGMPITuneLib* to our application, which will give us the possibility to collect information about the runtimes of the different *mockup-ups*. The last piece we are missing to get into the development of *pgchecker* is how we will exploit *PGMPITuneLib* to obtain the measurements for different algorithms. Once the library is linked to our program, we can issue the command shown in Listing 2.6 to take measurements of the various implementations. It can be observed that we have the opportunity to execute the collective operation in several iterations but with different message sizes, in this instance we use 1 B, 2 B, and 4 B. Furthermore, we specify with `--nrep=10` that we execute the selected variant ten times and thus also receive ten measurement values. With `--calls-list=MPI_Bcast`, we select the desired collective operation, in our case *MPI_Bcast*, and define an output file that we will later parse with our *pgchecker* tool. Instead of *MPI_Bcast*, it is also possible to specify any other collective operation that Hunold et al. [4] and Träff et al. [11] implemented in the *PGMPITuneLib* library. Finally, it is important to consider the option `--module=bcast=alg:default`. In this case, we select the *default* implementation and use *PGMPITuneLib* only to execute and measure the runtime of the *default* implementation of *MPI_Bcast*. In contrast, Listing 2.7 shows that we are selecting `alg:bcast_as_allgatherv`, which is the *mockup-up* consisting of *MPI_Allgatherv* that yields the same result as the *default* algorithm of *MPI_Bcast*. With this option, we are able to forward all *default* implementations to the *mockup-ups* defined in Equation 2.1 to Equation 2.22 and check them for violations against the performance guidelines.

Finally, it should be mentioned that the benchmarking approaches used by *PGMPITuneLib* were also developed by Hunold et al. [12]. Our *pgchecker* tool is developed under the environment of this *ReproMPI* benchmark suit.

CHAPTER 3

Software Design and Statistical Analysis of Violations

In the previous chapter, we introduced all the important concepts that are necessary for the development of our tool. We introduced the *MPI* standard, the self-consistent performance guidelines, and the underlying software framework *PGMPITuneLib*. In this chapter, we will focus on the development of the tool *pgchecker*. For this purpose, we will first have a look at the kinds of preconditions that have to be met on the machine to use *pgchecker*, and subsequently, we will explain how the program can be executed. Secondly, we will explain the architecture of *pgchecker* and describe how *pgchecker* can be extended by the user. Afterwards, we will cover all comparisons already implemented in *pgchecker* and show how the output looks like in the actual usage for each comparison. Finally, we will explain which statistical tests are used to evaluate significant violations against the *MPI* performance guidelines.

3.1 Preconditions and Utilization of *pgchecker*

pgchecker is an application that is built on top of the *PGMPITuneLib* library and thus requires it to be operational under all circumstances. In order to utilize the program, the user needs an arbitrary *MPI* library that is to be checked for violations against the self-consistent performance guidelines. Furthermore, a *CMake* release greater than *Version 3.22* is required, so that the provided *CMake* [13] files can be used for building. Finally, the *GNU Scientific Libraries (GLS)* [14] are needed, which provide mathematical functions for the programming languages *C* and *C++*. If all preconditions are fulfilled, the project can be built and the user will eventually be presented with the program *pgchecker* as a binary file.

Listing 3.1: The synopsis of *pgchecker*.

```
sh-3.2# pgchecker --help

USAGE: ./bin/pgchecker -f input_file [options]

OPTIONS:
  ?, -h, --help                     Display this information.
  -c, --comparer {0|1|2|3|4|5}      Specify the comparer type (
                                      0=Simple,
                                      1=Absolute Median,
                                      2=Relative Median,
                                      3=Violation-Test,
                                      4=Detailed Violation-Test,
                                      5=Grouped Violation-Test )
  -t, --test {0|1|2}                Specify the test type (
                                      0=T-Test,
                                      1=Wilcoxon-Rank-Sum-Test,
                                      2=Wilcoxon-Mann-Whitney )
  -o, --output <path>               Specify an existing output folder.
  -m, --merge                       Additionally results of all collectives
                                    are merged into one table.
  -s, --csv                         Print results to .csv file.
                                    Output directory must be specified.
  -v, --verbose                     Print warnings and results to console.
```

Listing 3.2: The content of the input file used for testing the collective operation *MPI_Bcast*.

```
MPI_Bcast --msizes-list=10 --nrep=10 --proc-sync=roundtime
    ↪ --rt-bench-time-ms=2000 --rt-barrier-count=0
```

When starting to utilize a program, the first step is to get familiar with it. For this purpose, there is the possibility to call *pgchecker* with the option `--help` to have a look at the structure of the options of the program. The output of this command is shown in Listing 3.1. For the time being, we are mostly interested in the general use of the command and the mandatory parameters. *pgchecker* requires only one mandatory parameter, and that is an input file which can be specified with the option `-f`. In this file, *pgchecker* searches for the collective operations to test.

For instance, if we want to test the *MPI_Bcast* operation, we will specify a file that looks similar to the one shown in Listing 3.2. The first entry indicates that we want to check the collective operation *MPI_Bcast*. Instead of *MPI_Bcast*, all other collective operations implemented by Hunold et al. [4] in *PGMPITuneLib* can be used: *MPI_Allgather*, *MPI_Allreduce*, *MPI_Alltoall*, *MPI_Barrier*, *MPI_Scan*, *MPI_Gather*, *MPI_Reduce*, *MPI_Reduce_scatter*, *MPI_Reduce_scatter_block*, *MPI_Exscan*, and *MPI_Scatter*. The *mockup-ups* matching the *default* implementations can be derived from the pattern

performance guidelines we have already described in Equation 2.1 to Equation 2.22. Additionally, we define the message size with which the tests should be performed. It should be noted that we can also add other sizes so that the option looks like the following: `--msizes-list=1,10,100`. By specifying a list containing multiple sizes, the operation is tested using different message sizes, in this case 1 B, 10 B, and 100 B. The option `--nrep=10` describes how often measurements should be collected. In this case, ten runs would be performed. Furthermore, the option `--rt-bench-time-ms=2000` can be used to set an upper bound for the runtime of *pgchecker*. If the runtime of the `--nrep` passes is greater than the runtime defined with option `--rt-bench-time-ms`, not all passes will be executed.

In addition to the mandatory parameters, we have also given the user the opportunity to customize *pgchecker* in many ways. The first customization is done with the option `--comparer`, which can also be seen in Listing 3.1. Using this option, the comparison applied to the data can be set. In this work we will mainly use the *grouped violation comparer*, which outputs the fastest *mockup-up* and its median runtime for each collective operation and message size if a violation occurred. This comparison is particularly useful because the most important information is displayed in a well-structured and clear manner and the user can immediately see which collective operations violate one of the guidelines. We will discuss the comparisons in more detail in Section 3.3. Another option that can be applied is `--test`. With this option, the user defines the statistical test used to detect a significant deviation. The tests are described in more detail in Section 3.4. Another option is `--output`, which can be used to specify an existing output folder where all the obtained data should be stored. The option `--merge` is used to merge the data tables of all tested collective operations, which is particularly interesting if further analysis is to be performed on this data. The `--csv` option can be used to save the generated tables in the form of a *comma-separated values (csv)*-file in the specified folder. This feature is also particularly useful for creating graphical presentations in other programs from the information obtained. In addition to the customization options, the object-oriented design of *pgchecker* offers further possibilities for the user to customize the tool further and, for instance, to develop new comparisons or statistical tests.

3.2 Understanding the Program Flow and the Object Architecture of *pgchecker*

Having clarified the prerequisites and the execution of our *pgchecker* tool, we would like to take a closer look at the basic architecture, the exploitation of *PGMPITuneLib* and the functionality of *pgchecker*. For this purpose, we first want to give an overview of the flow of *pgchecker*. Therefore, we can consider Figure 3.1, which shows a flow chart of *pgchecker*. When the tool is executed, it is first checked whether the specified options are valid. Depending on the configuration, further options are selected if required. For instance, the option `--verbose` is always activated in case no output folder is defined. The reason for this is that the generated statistics would otherwise not be returned anywhere.

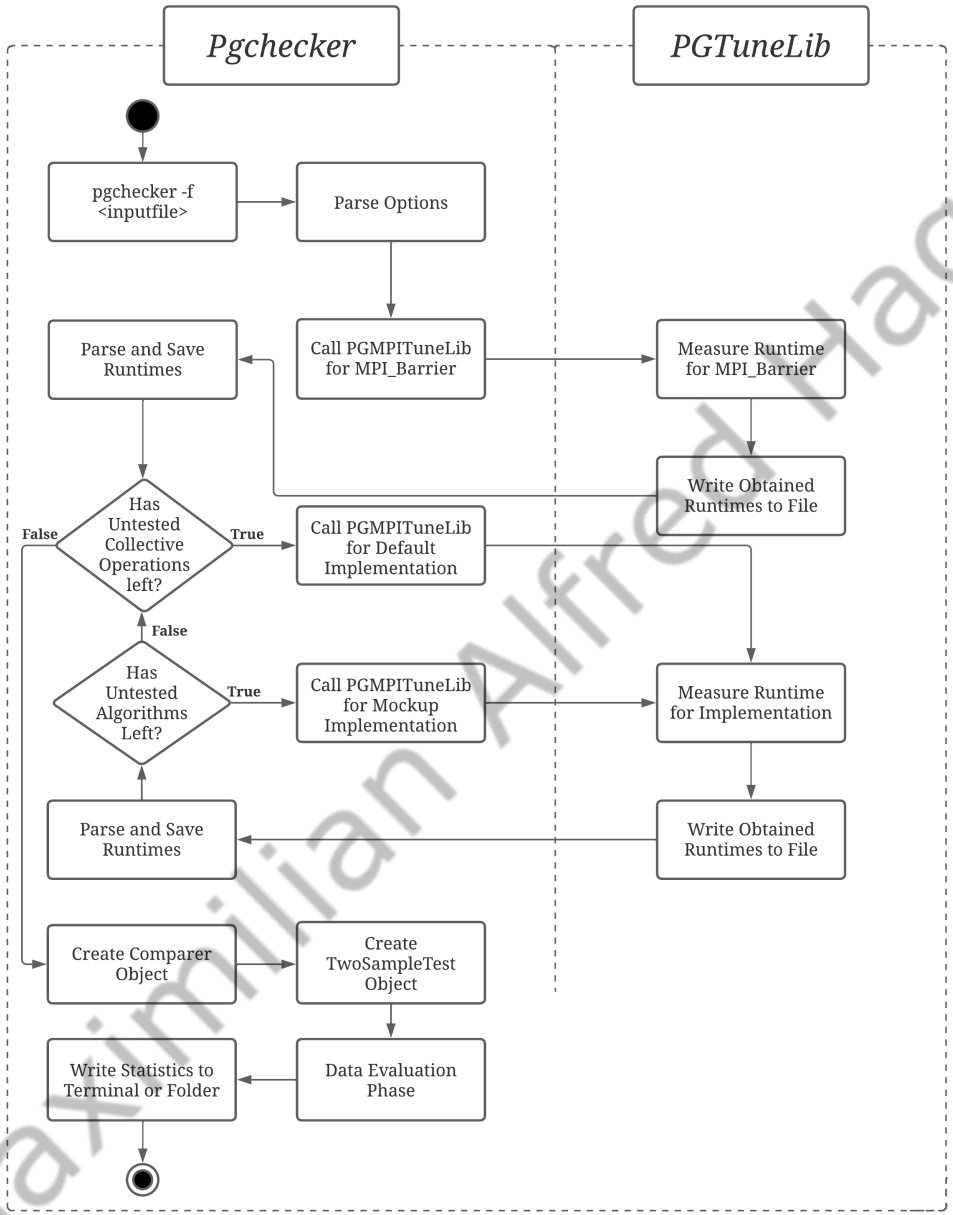

Figure 3.1: The flow chart of *pgchecker*.

It should be mentioned that at this point *pgchecker* executes *PGMPITuneLib* for the first time to perform the measurement of a *MPI_Barrier* operation. We need this runtime to issue a warning in the output if the runtime of *MPI_Barrier* is greater than the runtime of the individual algorithms. This warning is crucial because the *MPI_Barrier* function is used to synchronize the processes between the measurements. Since the processes can leave the *MPI_Barrier* functions at different points in time, the *MPI_Barrier* functions have a significant impact on the measurement. Hunold and Carpen-Amarie [15] have investigated this characteristic and presented alternative approaches to measure the runtime of collective operations.

Subsequently, two nested loops are performed. The outer loop is executed for each collective operation specified in the input file, and the inner loop is carried out for each algorithm defined for that operation. However, the *default* implementation is always inspected first. For instance, if we have defined the *MPI_Bcast* operation in the input file, the *default* implementation should be tested first. This is done by invoking the *PGMPITuneLib* library with the appropriate parameters. *PGMPITuneLib* will execute the selected algorithm and then save the obtained runtimes to a file. Immediately afterwards, this data is parsed by *pgchecker* into an internal data object and stored for later processing. This process is then performed for all *mockup-up* implementations defined in the *PGMPITuneLib* library and is described by the inner `for`-loop. Once all implementations have been measured, the process is performed for all other collective operations defined in the input file. This procedure is finished when all *default* and all *mockup-up* implementations of the collective operations specified in the input file have been measured. At this point, *pgchecker* possesses a lot of data about the runtimes of the different variants. A *comparer* is created for this data depending on the given option and the process of evaluation is started. Mostly statistical values of the data will be calculated and stored in tables. For the *comparer* that should additionally provide a violation value, the defined statistical test is additionally created and used. The resulting data is then stored in tables, depending on the type of *comparer*. The tables are then read by a so-called `PGDataPrinter` object and then written to the specified output, such as to the terminal or to files. The `PGDataPrinter` takes care of all options concerning the output. For instance, it merges the tables, creates clear tables, or generates *csv*-files. *pgchecker* terminates when the complete information has been written. The information is then visible to the user either in the terminal or in the previously defined output folder and can be used for further analysis.

We have already considered some of the most crucial objects, as we introduced the program flow of *pgchecker*. We would like to discuss these objects in more detail. The most essential objects are: `PGCheckOptions`, `PGData`, `PGDataTable`, `PGDataPrinter`, `PGDataComparer`, `TwoSampleTest`, and `ComparerData`. As already mentioned, the first task of *pgchecker* is to parse and check the given options. This is the responsibility of the `PGCheckOptions` object. It is used to parse the options, but it can also be used to set default options. Additionally, it provides the possibility to enable options that are

Listing 3.3: The *pgmpi_conf.csv*-file containing the available algorithms for *MPI_Bcast*.

```
cliname;mpiname;algname;rooted
.
.
bcast;MPI_Bcast;bcast_as_allgatherv;1
bcast;MPI_Bcast;bcast_as_scatter_allgather;1
.
.
```

needed but not specified. If the measured values are written to a file by *PGMPITuneLib*, *pgchecker* becomes active again and parses the obtained data. This data is written to an object of type `PGData` and stored there until further processing starts. For further processing, the data is usually written to `PGDataTable` objects, which can be used for different purposes due to their flexible form. The `PGDataComparer` object is the heart of *pgchecker* and contains functions to evaluate the obtained data. Actually, `PGDataComparer` is a superclass implemented by the different *comparers*. We will discuss these classes in more detail in Section 3.3, but at this point, it should already be stated that we use the factory pattern in order to be able to select the individual *comparer* as flexibly as possible. `TwoSampleTest` and `ComparerData` are auxiliary objects employed by the *comparer*. `TwoSampleTest` is again a superclass of several two-sample tests we have already implemented, but also provides an interface to add more tests. `ComparerData` is another auxiliary object used by the *comparer*, and it contains all evaluated data and stores them for subsequent conversion into `PGDataTable` objects. Finally, we learned about the object `PGDataPrinter`. This object is responsible for all kinds of probable outputs, such as the printing of the possible options, warnings, errors or the correct statistics after evaluation. If, for instance, the formatting of the tables should be changed or additional file formats should be supported, the object `PGDataPrinter` is the right place to start extending.

Finally, we would like to discuss the utilization of *PGMPITuneLib*. We have already seen that *PGMPITuneLib* is called for each collective operation and each of its algorithms. In Chapter 2, we demonstrated that *PGMPITuneLib* can also be used to select the best algorithm from tuning profiles. In our use case, however, we want to check the runtimes of all possible algorithms. For this purpose, we need a file that lists all *default* and *mockup-up* implementations. A section of this file describing the *MPI_Bcast* operation is shown in Listing 3.3. Using the *MPI_Bcast* example, it can be seen that each *mockup-up* implementation is listed. The object `PgTuneLibInterface` provides functions that allow reading this file and listing the alternative implementations of all collective operations. Subsequently, the appropriate arguments for the *PGMPITuneLib* are assembled by *pgchecker*. As already mentioned when looking at the program flow, the *PGMPITuneLib* is then executed for each *default* and each *mockup-up* implementation. After the data has been produced by *PGMPITuneLib*, *pgchecker* will resume to either start new executions or to start with the evaluation phase.

3.3 Comparing Raw Runtime Data

The main purpose of *pgchecker* is to check the runtimes of the different implementation possibilities of the collective operation against the performance guidelines. Thus, the main goal of *pgchecker* is to provide a clear overview of the data. For this purpose, we used so-called *comparer*, which are able to perform different evaluations and therefore generate different statistics. We distinguish between *comparer* that compare raw runtimes, we refer to them as *runtime comparer*, and *comparer* that create tables that list the violations of the guidelines, we refer to them as *violation comparer*. The former can mainly be utilized to generate graphs. The second one is primarily used for quick detection of the violations against the guidelines. In the following, we will run through an example for each *comparer* and thus discuss the output of the respective *comparer*.

Before looking at the individual *comparer*, we would like to explain the architecture of these first. We have already explained that they can be created by the factory pattern. This is made possible by the fact that all specific *comparer* implementations inherit from the superclass `PGDataComparer`. Each *comparer* exposes exactly one function to the outside, and that is the `PGDataTable get_results()`, as shown by the class diagram in Figure 3.2. Thus, an object of type `PGDataTable` is always returned, as we already mentioned, it is a flexible object for storing text-based tables. The advantage of this standardization is that changes and additions to the *comparer* only have to be carried out at one place. If the evaluation of the data is completed, the `PGDataPrinter` object expects a table of type `PGDataTable` in any case, where the content is determined by the *comparer*. This design also improves the management of the output in the `PGDataPrinter` object, because the table is always present, and the printer can decide on which output and in which format it is to be written. If a new *comparer* is to be added, it is possible to simply derive it from the superclass `PGDataComparer` and implement the necessary methods. Currently, five *comparer* are implemented in *pgchecker*, as shown by the class diagram in Figure 3.2.

In Table 3.1, a brief textual description of the objectives of each *comparer* can be seen. However, we also would like to consider each *comparer* in detail and discuss the outputs of them. The tables will be created on the *Hydra* system with $N = 36$ compute nodes and $n = 32$ cores per node. We will utilize the *MPI* implementation *OpenMPI version 4.1.4* [5]. For the workflow example, we have tested the *default* implementation, the *mockup-up* implementation *MPI_Allgatherv*, and the *mockup-up* implementation consisting of *MPI_Scatter* and *MPI_Allgather* for the collective operation *MPI_Bcast*. We use the message sizes 1 B, 2 B, 4 B, 8 B, 32 B, 64 B, 512 B, 1 KiB, 4 KiB, 8 KiB, 16 KiB, 32 KiB, and 64 KiB, but for the purpose of clarity we will only show a subset of these in the tables. Furthermore, it should be mentioned that we will format the values in the tables with units. When using *pgchecker*, the unit millisecond is always used for runtime values and the unit byte for message size values.

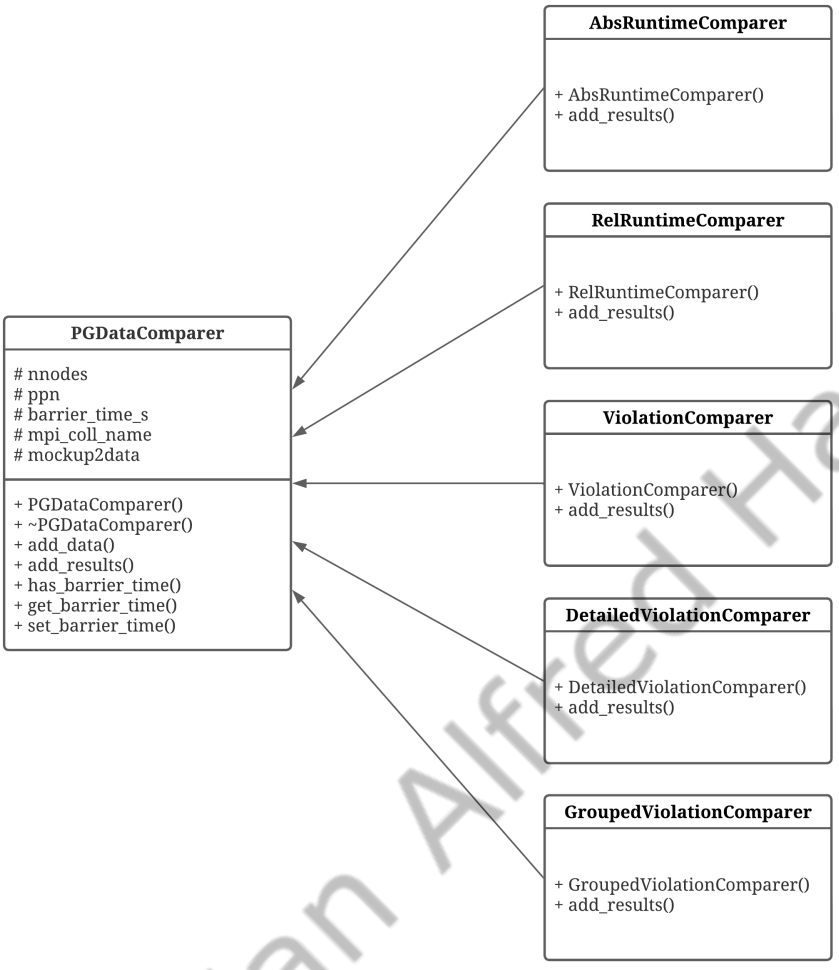

Figure 3.2: Class diagram representing the `PGDataComparer` superclass and the specific implementations `AbsRuntimeComparer`, `RelRuntimeComparer`, `ViolationComparer`, `DetailedViolationComparer` and `GroupedViolationComparer`.

First, we would like to explain the two *runtime comparer* in more detail. The first *comparer* we want to review is the `AbsRuntimeComparer`. The resulting table can be seen in Table 3.2. There are four columns within the table. The first one states the tested collective operation. The message size can be found in column two. The *comparer* does not group by collective operation or message size, instead it returns all tested implementations. The third column contains the underlying algorithm which can be a *default* or a *mockup-up* implementation. The last column holds the median runtime in milliseconds of the individual implementations. The output of the second *runtime comparer* `RelRuntimeComparer` is structured similarly.

Table 3.1: The different objectives of the *comparer* in *pgchecker*.

Comparer	**Objective**
Absolute Runtime	The focus of this *comparer* is to display the median runtimes of all tested algorithms as clearly as possible.
Relative Runtime	In this *comparer*, the median runtime of the *default* implementation is listed relative to the respective *mockup-up* algorithms. The higher this value is above one, the more severe the violation of the performance guidelines is.
Violation	This *comparer* can be used to apply a statistical two sample test to the raw runtime data. In addition to the mean and median values of the runtimes, it displays whether there is a violation and how severe it is for each *mockup-up* implementation.
Detailed Violation	This *comparer* provides many different statistical evaluations, such as the mean values, the median values, a violation indicator, and the slowdown. The goal is to gather a great amount of information in one run.
Grouped Violation	The aim of this *comparer* is to display only one row for each collective operation and message size combination. For each combination, the fastest *mockup-up* is displayed if there were multiple violations. If no violations occur, these columns are left empty.

The table generated by `RelRuntimeComparer` can be seen in Table 3.3. The only difference is that instead of the runtime in milliseconds, the relative runtime is given instead. Use case for these two *runtime comparer* is for instance, the generation of plots showing the slowdown of the *mockup-up* algorithms in relation to the *default* algorithm. A graph created from this data using *R* can be seen in Figure 3.3. The graph indicates the slowdown of the *mockup-ups* relative to the *default* implementation for each measured message size. This graph clearly shows the violations of the guidelines for which we are searching. The green square highlights the area where the *mockup-up* implementations are not significantly faster than the *default* implementation, and thus there is no violation. However, if a *mockup-up* is in the red zone, there is a violation because the runtime is reduced by replacing the *default* implementation with the *mockup-up* implementation. In this specific case, for instance, we can observe that the *mockup-up* implementation consisting of *MPI_Scatter* and *MPI_Allgather* is faster than the *default* algorithm for large message sizes. In the following, we will further look at the *comparer* that output these found violations in the form of a table.

Table 3.2: The generated output using the `AbsRuntimeComparer` in *pgchecker*.

collective_name	message_size	mockup_name	median
MPI_Bcast	8 KiB	Default	0.032 022 ms
MPI_Bcast	16 KiB	Default	0.070 215 ms
MPI_Bcast	32 KiB	Default	0.142 892 ms
MPI_Bcast	64 KiB	Default	0.223 846 ms
MPI_Bcast	8 KiB	Scatter+Allgather	0.053 380 ms
MPI_Bcast	16 KiB	Scatter+Allgather	0.073 972 ms
MPI_Bcast	32 KiB	Scatter+Allgather	0.113 139 ms
MPI_Bcast	64 KiB	Scatter+Allgather	0.194 805 ms
MPI_Bcast	8 KiB	Allgatherv	0.117 725 ms
MPI_Bcast	16 KiB	Allgatherv	0.214 396 ms
MPI_Bcast	32 KiB	Allgatherv	0.375 414 ms
MPI_Bcast	64 KiB	Allgatherv	0.663 881 ms

Table 3.3: The generated output using the `RelRuntimeComparer` in *pgchecker*.

collective_name	message_size	mockup_name	relative_runtime
MPI_Bcast	8 KiB	Default	1
MPI_Bcast	16 KiB	Default	1
MPI_Bcast	32 KiB	Default	1
MPI_Bcast	64 KiB	Default	1
MPI_Bcast	8 KiB	Scatter+Allgather	0.618476
MPI_Bcast	16 KiB	Scatter+Allgather	0.992201
MPI_Bcast	32 KiB	Scatter+Allgather	1.182435
MPI_Bcast	64 KiB	Scatter+Allgather	1.249800
MPI_Bcast	8 KiB	Allgatherv	0.266964
MPI_Bcast	16 KiB	Allgatherv	0.316004
MPI_Bcast	32 KiB	Allgatherv	0.331851
MPI_Bcast	64 KiB	Allgatherv	0.335941

Next, we would like to analyze the *violation comparer* in more detail. We start by using the `ViolationComparer`, the output of which can be seen in Table 3.4. Since the tables of the *violation comparer* contain more information, we have formatted them differently for the presentation in this thesis. However, we will always point out changes in the real application and the presentation. For instance, in the underlying table, the column indicating the collective operation has been removed and the amount of decimal points has been reduced for all numbers. In the first column of the table, we can see the respective algorithm tested, with both *default* and *mockup-up* implementations displayed. Behind some names of the *mockup-up* implementations an asterisk symbol can be recognized. This indicates that the runtimes of this algorithm are smaller than the runtime of a

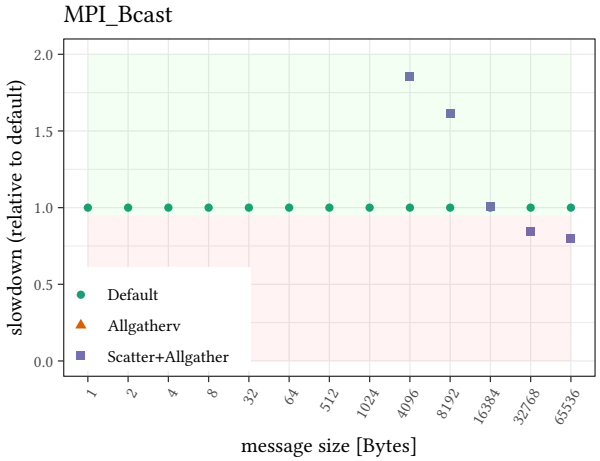

Figure 3.3: A possible graph created from the output of the `RelRuntimeComparer`. The relative runtimes of the *mockup-up* algorithm in relation to the *default* implementation are plotted for the collective operation *MPI_Bcast*. On the x-axis the message sizes are applied. If a runtime value of a *mockup-up* is within the green area, no violation occurred. The red area indicates that there is a violation. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 32$ cores per node.

Table 3.4: The generated output using the `ViolationComparer` in *pgchecker*.

mockup	m_size	N	ppn	nrep	mean	median	z_value	crit	vio
Default	8 KiB	36	32	5000	0.03 ms	0.03 ms			
Default	16 KiB	36	32	5000	0.07 ms	0.07 ms			
Default	32 KiB	36	32	5000	0.13 ms	0.12 ms			
Default	64 KiB	36	32	5000	0.24 ms	0.22 ms			
Scatter+Allgather*	8 KiB	36	32	5000	0.05 ms	0.05 ms	16.84	1.64	0
Scatter+Allgather*	16 KiB	36	32	5000	0.07 ms	0.07 ms	1.05	1.64	0
Scatter+Allgather*	32 KiB	36	32	5000	0.11 ms	0.10 ms	-6.10	1.64	1
Scatter+Allgather	64 KiB	36	32	5000	0.19 ms	0.18 ms	-13.18	1.64	1
Allgatherv*	8 KiB	36	32	5000	0.12 ms	0.11 ms	74.86	1.64	0
Allgatherv*	16 KiB	36	32	5000	0.21 ms	0.21 ms	93.00	1.64	0
Allgatherv*	32 KiB	36	32	5000	0.38 ms	0.37 ms	81.20	1.64	0
Allgatherv*	64 KiB	36	32	4295	0.66 ms	0.66 ms	104.68	1.64	0

single *MPI_Barrier* operation. Since *MPI_Barrier* operations are sometimes needed to synchronize the individual processes, they are especially significant for algorithms marked with asterisks. Algorithms marked with the asterisk symbol should therefore be considered with particular caution. In the second column, we find the message size and in columns three and four the number of nodes utilized and the processes per node. In the next column, we see the number of measurements that *PGMPITuneLib* has taken. We can observe that the number does not always correspond to the specified one and

Table 3.5: The generated output using the `DetailedViolationComparer` in *pgchecker*.

mockup	count	alt_mean	alt_med	z_val	crit	vio	slowdown
Scatter+Allgather	1 KiB	0.03 ms	0.03 ms	64.83	1.64	0	0.44
Scatter+Allgather	4 KiB	0.04 ms	0.04 ms	64.75	1.64	0	0.52
Scatter+Allgather	8 KiB	0.05 ms	0.05 ms	56.34	1.64	0	0.61
Scatter+Allgather	16 KiB	0.07 ms	0.07 ms	2.58	1.64	0	1.00
Scatter+Allgather	32 KiB	0.11 ms	0.10 ms	-9.27	1.64	1	1.20
Scatter+Allgather	64 KiB	0.20 ms	0.18 ms	-7.13	1.64	1	1.24
Allgatherv	1 KiB	0.05 ms	0.05 ms	77.56	1.64	0	0.24
Allgatherv	4 KiB	0.08 ms	0.07 ms	146.55	1.64	0	0.26
Allgatherv	8 KiB	0.11 ms	0.10 ms	126.14	1.64	0	0.27
Allgatherv	16 KiB	0.21 ms	0.20 ms	122.09	1.64	0	0.32
Allgatherv	32 KiB	0.38 ms	0.37 ms	59.44	1.64	0	0.33
Allgatherv	64 KiB	0.67 ms	0.66 ms	162.99	1.64	0	0.33

that some of the runs are not performed due to the specified upper runtime limit. In the next two columns, we see the mean and the median for the respective algorithm and its message size. Afterwards, the z-value calculated by the selected statistical test is given. In the second to last column, the critical z-value is given. If this value is not reached, a violation occurs and the value in the last column is set to 1. In this instance it can be recognized that there is a violation against the performance guidelines for the message sizes of 32 KiB and 64 KiB. The violating *mockup-up* is the combination of *MPI_Scatter* and *MPI_Allgather*.

The next *comparer* we would like to introduce is the `DetailedViolationComparer`, whose output can be seen in Table 3.5. As the name suggests, this *comparer* contains a lot of information. All columns already presented in Table 3.4 have therefore been omitted for the purpose of clarity, but they are also displayed when using the actual tool. The biggest difference to the `ViolationComparer` is that in each line the mean and median values of the *default* implementation are displayed in comparison to the mean and median values of the *mockup-up* implementations. Since the most important information of the *default* implementation can thus be found in each line, no separate lines are created for the *default* variants. Displaying the median runtime of the *default* next to the median runtime of the *mockup-up* implementation, it is possible to detect, if the deviation is significant. Additionally, this *comparer* shows the slowdown in relation to the *default* implementation. The slowdown is the median of the *default* implementation divided by the median of the respective *mockup-up* implementation for each message size. In the real application, we would see two more columns. The first column shows, as before, with an asterisk symbol, whether the runtime of the *MPI_Barrier* operation is greater than the runtime of the actual collective operation. In the `DetailedViolationComparer`, this symbol is not placed behind the *mockup-up* name as in the others *comparer*, but is displayed in a separate column for a better overview. Furthermore, the second column

Table 3.6: The generated output using the `GroupedViolationComparer` in *pgchecker*.

coll	m_size	N	ppn	nrep	def_med	slowd	mockup	alt_med
Bcast	1 B	36	32	5000	0.0068 ms			
Bcast	2 B	36	32	5000	0.0061 ms			
Bcast	4 B	36	32	5000	0.0062 ms			
Bcast	8 B	36	32	5000	0.0061 ms			
Bcast	32 B	36	32	5000	0.0104 ms			
Bcast	64 B	36	32	5000	0.0096 ms			
Bcast	512 B	36	32	5000	0.0114 ms			
Bcast	1 KiB	36	32	5000	0.0126 ms			
Bcast	4 KiB	36	32	5000	0.0190 ms			
Bcast	8 KiB	36	32	5000	0.0291 ms			
Bcast	16 KiB	36	32	5000	0.0654 ms			
Bcast	32 KiB	36	32	5000	0.1205 ms	1.1736	Scatter +Allgather*	0.1026 ms
Bcast	64 KiB	36	32	5000	0.2214 ms	1.2530	Scatter +Allgather	0.1767 ms

would indicate whether a *MPI_Barrier* operation is used at all when measuring this collective operation.

The most advanced *comparer* is the `GroupedViolationComparer`. For this *comparer*, a separate row is not written for each tested algorithm, but only for each combination of collective operation and message size. In principle, the statistical values of the *default* implementation are displayed for each combination, but if one or more violations against the *MPI* performance guidelines occur, only the fastest *mockup-up* and the corresponding statistical values are displayed. An output of this *comparer* can be seen in Table 3.6. Like the other *comparer*, the first column presents the tested collective operation. In the second column, the message sizes are shown. In columns three and four, the utilized nodes and the processes per node are displayed. Column five indicates the number of measurements performed. The next column represents the median runtime of the *default* implementation. Columns seven to nine are filled dynamically and therefore only contain entries if a violation against the *MPI* performance guidelines has occurred. With this *comparer*, we can easily recognize the found violations. At a glance, it can be observed that there is a violation against the guidelines only for the message sizes 32 KiB and 64 KiB. As already seen by using the other *comparer*, the violating *mockup-up* is the combination of *MPI_Scatter* and *MPI_Allgather*. For the occurred violations, the slowdown, the violating *mockup-up* and the median runtime of the violating implementation are displayed. It is important to note that in the `GroupedViolationComparer` only the violating algorithm with the lowest runtime is displayed. For instance, if there are another violation by the *mockup-up MPI_Allgatherv*, where the median runtime is 0.11 ms, it would not be displayed because it is greater than that of the *MPI_Scatter* and *MPI_Allgather mockup-*

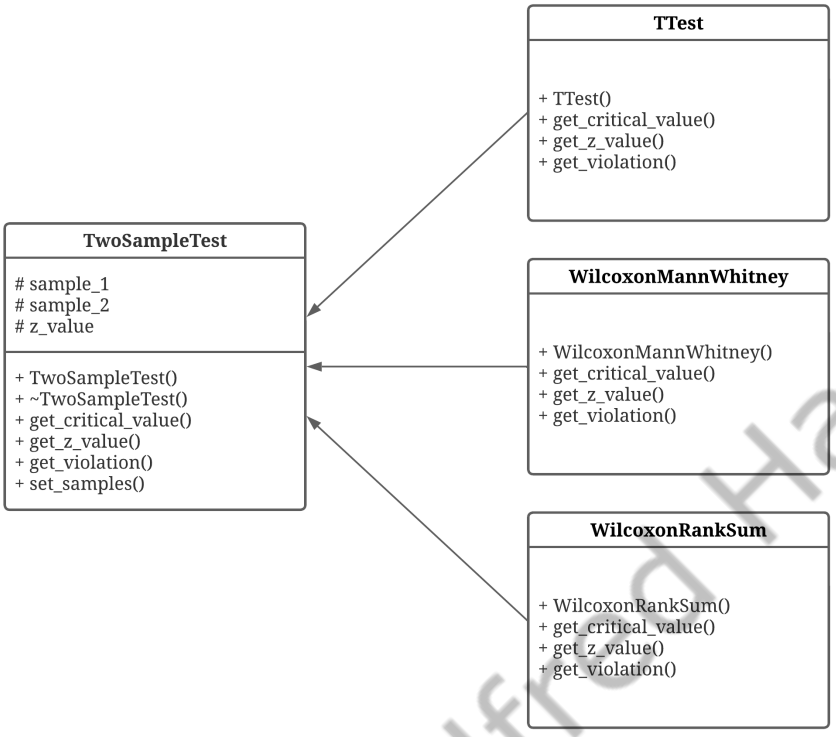

Figure 3.4: Class diagram representing the `TwoSampleTest` superclass and the specific implementations of `TTest`, `WilcoxonMannWhitney`, and `WilcoxonRankSum`.

up implementation. In Chapter 4, we will mainly use the `GroupedViolationComparer` for analysis and compare the outcome to graphs created from the raw runtime data generated by *PGMPITuneLib*.

3.4 Evaluating Statistical Tests Implemented in *pgchecker*

To round off Chapter 3, we would like to discuss the various statistical tests that can be performed by *pgchecker*. Figure 3.4 shows the class diagram of the architecture of the two-sample tests. It can be seen that similar to the *comparer* architecture, there is a superclass `TwoSampleTest` from which all specific two sample tests implementations inherit. Thus, for statistical tests as well, it is possible for the user to design additional tests and simply add them to the predefined tests. All implemented tests inherit two arrays containing the data of sample one and sample two respectively. The functions `get_critical_value()`, `get_z_value()`, and `get_violation()` are unique to each specific implementation. The first function returns the critical value for the respective test. The second function calculates the deviation value corresponding to the defined calculations of that test type. The last function compares the calculated value to the critical value and returns a boolean

value indicating whether the deviation is statistically significant. Currently, `TTest` and two variants of statistical tests from Wilcoxon, `WilcoxonMannWhitney`, and `WilcoxonRankSum` have been implemented in *pgchecker*. It should be mentioned that we assume that the data is normally distributed for sample sizes $n > 20$ and thus apply the critical value of the normal distribution of $crit\_value = 1.644854$ for a one-tailed significance level of 5%. For smaller sample sizes, we use the critical values of the respective tests.

First, we would like to take a closer look at the object `TTest`. For the calculation of the deviation value, we need degrees of freedom, which are calculated by adding the sizes of the two samples minus two:

$$\begin{aligned} sample\_1\_df &:= |sample\_1| - 1 \quad , \\ sample\_2\_df &:= |sample\_2| - 1 \\ \Rightarrow df &:= sample\_1\_df + sample\_2\_df \quad . \end{aligned}$$

The standard deviation is defined as follows:

$$sd := \sqrt{\frac{sample\_1\_df \times \sigma^2(sample\_1) + sample\_2\_df \times \sigma^2(sample\_2)}{df}} \quad .$$

Further, the calculated deviation value of a *t-test* is:

$$t\_value := \sqrt{\frac{|sample\_1| \times |\text{sample}\_2|}{|sample\_1| + |sample\_2|}} \times \frac{mean(sample\_2) - mean(sample\_1)}{sd} \quad .$$

In case of the object `TTest`, this calculated t_value is returned from the function `get_z_value()`. The function `get_violation()` is used to check whether the violation against the guidelines is significant:

$$violation = \begin{cases} 1 & \text{if } t\_value < critical\_value \\ 0 & otherwise \end{cases} \quad .$$

We have seen that the *t-test* works on the mean values of random samples. However, we often use medians when analyzing the *MPI* implementations, for instance when calculating the slowdown. In order to optimize the reporting of violations, we have thus implemented other tests, most notably the *Wilcoxon Rank-Sum Test*, which operates mainly on the medians of the samples. The initial situation is the same for this type of test, meaning that we again receive two samples. The first step in our program is to join the two samples together, storing which entry comes from which sample. The merged array is then sorted by runtime. Each value is now assigned a rank one after the other. It is important to be careful with entries that have exactly the same runtime. The ranks of these are stored and each rank is subsequently assigned the average value:

$$average\_rank := \sum_{i=1}^{n} \frac{rank_i}{n} \quad ,$$

where n is the number of entries that share a rank. After each entry has been assigned a rank, the ranks for each sample are summed up. For this test type, we calculate the standard deviation as follows:

$$sd := \sqrt{\frac{|sample\_1| \times |sample\_2| \times (|sample\_1| + |sample\_2| + 1)}{12}} \quad .$$

In order to calculate the deviation value, we also have to know the expected value:

$$expected\_value := \frac{|sample\_1| \times (|sample\_1| + |sample\_2| + 1)}{2} \quad .$$

Finally, the z-value is given by the following formula:

$$z\_value := \frac{rank\_sum\_mockup - expected\_value}{sd} \quad .$$

In the implementation of this test, the calculated value of the deviation is again returned using the function `get_z_value()`. The calculation of the violation is identical to the calculation carried out for the *t-test*.

Furthermore, it should be noted that we have implemented another variant of the *Wilcoxon Rank-Sum Test*, the *Wilcoxon-Mann-Whitney Test*. This test is a variation of the already presented *Wilcoxon Rank-Sum Test*. In the normalized form, the two tests give exactly the same result. However, due to the calculated U-value, we could also refer to the critical U-values tables. We would like to additionally explain this variant because we will mainly apply this implementation for median based violation testing. The construction of the rank sums is done exactly as previously described. However, this time we assemble a different test size, using a U-value:

$$U := \frac{|sample\_1| \times |sample\_2| + |sample\_1| * (|sample\_1| + 1)}{2} - rank\_sum\_default \quad .$$

The standard deviation and the expected value are also calculated in the exact same way as for *Wilcoxon Rank-Sum Test*. The deviation value is given as follows:

$$z\_value := \frac{U - expected\_value}{sd} \quad .$$

Finally, it should be mentioned that in Chapter 4, we always utilize the *t-test* unless otherwise stated. If *pgchecker* is not given an option regarding the type of test, the *t-test* will also be performed by default.

CHAPTER 4

Workflow and Implementation Analysis on Multiple Systems

In the previous chapters, we have described the theoretical background as well as the developed tool *pgchecker*. In this chapter, we would like to introduce the analysis workflow with *pgchecker*. For this purpose, we will execute the most common collective operations *MPI_Allgather*, *MPI_Allreduce*, *MPI_Bcast*, *MPI_Gather*, *MPI_Reduce*, and *MPI_Scatter* with different message sizes on our *Hydra* system using the *MPI* implementation *Open MPI* [5]. We want to demonstrate how an *MPI* implementation can be analyzed using our *pgchecker* tool. Additionally, we will examine whether *pgchecker* can correctly detect the violations that appear in the raw runtime data. Throughout the workflow example, we will also analyze the *Open MPI* implementation. Afterwards, we will perform an analysis of several *MPI* implementations ourselves. Therefore, we will first use the *Hydra* system again to analyze the implementations *Intel ® MPI* [8], *MPICH* [6], and *MVAPICH* [7]. Lastly, we will test on the *Irene* system to take a closer look at the *Intel ® MPI* and *Open MPI* implementations by utilizing several thousand cores.

4.1 Hardware Configuration and Notations Used for the Performance Analysis

The evaluation of *pgchecker* will be carried out on two different systems. The so-called *Hydra* system is an Intel Skylake dual-socket machine with $N = 36$ compute nodes, each consisting of $n = 32$ cores. Thus, in total, we can utilize 1152 cores. The larger system is called *Irene*. As of November 2022, *Irene* [16] is the worlds 133rd most powerful supercomputer according to the *TOP500* list [17]. Every node contained in this machine is powered by two 24-cores Intel Skylake compute units with a clock speed of 2.7 GHz. The *Irene* system has a total of $N = 1656$ compute nodes, each consisting of $n = 48$ cores.

Listing 4.1: The commands used for the installation of the *pgchecker* tool.

```
sh-3.2# cmake -DOPTION_ENABLE_PGCHECKER=ON .
sh-3.2# make
sh-3.2# make config
```

For our analysis, we will use $N = 256$ compute nodes and $n = 48$ cores per node at most. Thus, we will utilize a maximum of 12288 cores. Each node in this system has 192 GB of memory available. In the following, we will reference the first system as *Hydra* and the second system as *Irene*. During the analysis, we will specify the number of nodes and the number of cores per node for each experiment using the notation $p = N \times n$. N will always denote the number of nodes and n the number of cores per node. The product of these two values is therefore always the total number of processes p used.

Moreover, we would like to define the two evaluation methods that will be used. We have already mentioned that in the output of the *comparer MPI_Barrier* warnings will be issued if the runtime of a *MPI_Barrier* option has a significant impact on the runtime of the algorithms. For the first evaluation method, which we call *brave*, we will list and count all violations found against the *MPI* performance guidelines. In contrast, we will use the *cautious* method, when we will remove all violations against the guidelines where *MPI_Barrier* warnings occurred. Basically, we will count less violations using the *cautious* variant, but we can be certain that the violation is significant.

4.2 *pgchecker* Analysis Workflow by Example

In this section, we would like to go through the workflow of analyzing *MPI* implementations using *pgchecker* by example. Before starting, we need to make sure that both *PGMPITuneLib* and the extension *pgchecker* are downloaded and available on the machine. Subsequently, we can use the commands presented in Listing 4.1 to build *pgchecker*. Using the first command, we can dynamically generate the *Makefiles*. Please note the preconditions we have already explained in Chapter 3. With the next command we can start the build process. After the build process is finished, the binary of *pgchecker* can be found inside the *bin*-folder. The last command creates the configuration file we have already presented in Listing 3.3. For our analysis we have modified the configuration file so that *pgchecker* does not test for lane and hierarchical topology *mockup-ups*, allowing us to focus on the pattern guidelines. If a *MPI* implementation is available on the machine, *pgchecker* is already executable. Throughout the workflow example we will use the *Open MPI* implementation.

The first step is to create an input file in which all collective operations to be tested are listed. The document we use for the tests is shown in Listing 4.2. Looking at the file, we notice that we want to test the collectives *MPI_Allgather*, *MPI_Allreduce*, *MPI_Bcast*, *MPI_Gather*, *MPI_Reduce*, and *MPI_Scatter*. Moreover, it can be seen that we use two different sets of message sizes. For *MPI_Allgather*, *MPI_Gather* and

Listing 4.2: Content of the *pgmpi_conf.csv*-file used for the analysis.

```
MPI_Allgather --msizes-list=1,2,4,8,32,64,512,1024,2048,4096,8192,16384
    ↪ --nrep=5000 --rt-bench-time-ms=3000
    ↪ --proc-sync=roundtime --rt-barrier-count=0
MPI_Gather    --msizes-list=1,2,4,8,32,64,512,1024,2048,4096,8192,16384
    ↪ --nrep=5000 --rt-bench-time-ms=3000
    ↪ --proc-sync=roundtime --rt-barrier-count=0
MPI_Scatter   --msizes-list=1,2,4,8,32,64,512,1024,2048,4096,8192,16384
    ↪ --nrep=5000 --rt-bench-time-ms=3000
    ↪ --proc-sync=roundtime --rt-barrier-count=0
MPI_Allreduce --msizes-list
    ↪ =1,2,8,64,512,1024,8192,16384,65536,131072,1048576,8388608
    ↪ --nrep=5000 --rt-bench-time-ms=3000
    ↪ --proc-sync=roundtime --rt-barrier-count=0
MPI_Bcast     --msizes-list
    ↪ =1,2,8,64,512,1024,8192,16384,65536,131072,1048576,8388608
    ↪ --nrep=5000 --rt-bench-time-ms=3000
    ↪ --proc-sync=roundtime --rt-barrier-count=0
MPI_Reduce    --msizes-list
    ↪ =1,2,8,64,512,1024,8192,16384,65536,131072,1048576,8388608
    ↪ --nrep=5000 --rt-bench-time-ms=3000
    ↪ --proc-sync=roundtime --rt-barrier-count=0
```

Listing 4.3: The *pgchecker* execution command used for the experiments.

```
./bin/pgchecker -f ./examples/pgcheck/input.txt -c 5
    ↪ -o ../experiments/hydra/36_32 -v -m -s
```

MPI_Scatter we use the sizes 1 B, 2 B, 4 B, 8 B, 32 B, 64 B, 512 B, 1 KiB, 2 KiB, 4 KiB, 8 KiB, 16 KiB. These are the collective operations where the total message size scales by the amount of p processes used. Thus, with $p = 256 \times 48$ processes and a message size of $m = 16$ KiB, we would already receive a total message size of 192 MiB. With *MPI_Allreduce*, *MPI_Bcast*, and *MPI_Reduce* the message size is not scaled, so we can send larger messages using these operations. We will use message sizes of 1 B, 2 B, 8 B, 64 B, 512 B, 1 KiB, 8 KiB, 16 KiB, 64 KiB, 128 KiB, 1 MiB, and 8 MiB. Thus, in total, we test twelve message sizes per collective. Each collective and each of its respective message sizes and *mockup-ups* should be executed and measured 5000 times. Furthermore, in Listing 4.2, we notice that a runtime limit of 3 s is defined. This means that the program tries to execute all 5000 runs, but stops when the runtime limit of 3 s is exceeded. Finally, we recognize two options that describe the measurement runtime settings. We will not go into detail about how these work in this thesis, but we would like to briefly mention that the `--proc-sync=roundtime` applies the round-time measurement developed by Hunold and Carpen-Amarie [15] and that the option `--rt-barrier-count=0` determines when to switch to synchronization using *MPI_Barrier* functions.

Table 4.1: *pgchecker* output table for *OpenMPI, Version 4.1.4* using the `GroupedViolationComparer`. All lines where no violation against the guidelines occurred are removed. The fastest *mockup-up* implementation for each collective operation and message size combination is displayed. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 1$ core per node.

collective	m_size	nrep	def_med	slowd	mockup	alt_med
Gather	4 KiB	5000	0.0367 ms	1.0122	Gatherv*	0.0362 ms
Allgather	1 KiB	5000	0.0419 ms	1.3321	Allgatherv	0.0314 ms
Allgather	2 KiB	5000	0.0522 ms	1.2477	Allgatherv	0.0418 ms
Allgather	4 KiB	5000	0.0733 ms	1.2936	Alltoall	0.0566 ms
Allgather	8 KiB	5000	0.1070 ms	1.1414	Alltoall	0.0938 ms
Allgather	16 KiB	5000	0.1897 ms	1.0904	Alltoall	0.1739 ms
Scatter	512 B	5000	0.0153 ms	1.3175	Scatterv*	0.0116 ms
Scatter	1 KiB	5000	0.0167 ms	1.2661	Scatterv*	0.0132 ms
Scatter	2 KiB	5000	0.0196 ms	1.2036	Scatterv*	0.0163 ms
Scatter	8 KiB	5000	0.0693 ms	1.6697	Scatterv	0.0415 ms
Bcast	64 KiB	5000	0.0773 ms	1.0666	Scatter+Allgather*	0.0725 ms
Bcast	128 KiB	5000	0.1269 ms	1.2330	Scatter+Allgather	0.1029 ms
Bcast	1 MiB	1899	1.5628 ms	1.7791	Scatter+Allgather	0.8784 ms
Bcast	8 MiB	249	12.0681 ms	2.1633	Scatter+Allgather	5.5785 ms
Reduce	64 KiB	5000	0.1036 ms	1.3251	Reducescatterblock +Gather	0.0782 ms
Reduce	128 KiB	5000	0.1545 ms	1.2318	Reducescatter +Gatherv	0.1254 ms
Reduce	1 MiB	2414	1.2119 ms	1.8484	Reducescatter +Gatherv	0.6556 ms
Reduce	8 MiB	205	14.5306 ms	2.8810	Reducescatter +Gatherv	5.0436 ms
Allreduce	64 B	5000	0.0918 ms	6.2009	Reduce+Bcast	0.0148 ms
Allreduce	512 B	5000	0.0220 ms	1.3604	Reduce+Bcast*	0.0162 ms
Allreduce	1 KiB	5000	0.0224 ms	1.2626	Reduce+Bcast*	0.0177 ms
Allreduce	128 KiB	5000	0.1771 ms	1.0921	Reducescatter +Allgatherv	0.1622 ms
Allreduce	1 MiB	3301	0.8792 ms	1.0548	Reducescatter +Allgatherv	0.8335 ms
Allreduce	8 MiB	467	6.3540 ms	1.1338	Reducescatter +Allgatherv	5.6039 ms

Next, we need to configure *pgchecker* by considering which arguments we want to apply. The command we will execute in this example can be seen in Listing 4.3. The first option defines the input file that we have already shown in Listing 4.2. Moreover, we decided to use the `GroupedViolationComparer`. The obtained data should be written to the output folder `../experiments/hydra/36_32`. We also define that all generated data should also be written to the terminal, that the obtained tables should be merged, and that the tables should be written to *csv*-files. Since no statistical test is specified, the `TTest` is executed by default.

If we execute this command on our *Hydra* system utilizing $N = 36$ compute nodes and $n = 1$ core per node, we obtain the table shown in Table 4.1. We have omitted all message sizes where no violations against the guidelines occurred to present the table more clearly. Furthermore, we have shortened the textual descriptions of the collective operations, removed the number of compute units and cores per node, and rounded the numerical value to four decimal places. In this table, it is apparent that we could find several violations against the performance guidelines. Since we use only one core per compute note, meaning $p = 36 \times 1$ processes in total, the runtime of all collective operations is rather low, and we see that a number of *mockup-ups* have been marked with a *MPI_Barrier* warning. For instance, the *MPI_Bcast* implementation at 8 MiB is particularly noticeable in this table. It can be observed that the *default* implementation is more than twice as slow as the *mockup-up* from *MPI_Scatter* and *MPI_Allgather*. The highest slowdown we observe is with *MPI_Allreduce* at a message size of 64 B. It can be seen that the *mockup-up* from *MPI_Reduce* and *MPI_Bcast* is six times faster than the *default* implementation. Overall, it can be stated that 24 violations out of 72 possible combinations of collective operations and message sizes are found. Using a more *cautious* approach and removing all violations that contain a *MPI_Barrier* warning, we still count 17 violations out of 72 combinations.

However, if we would like to have more information about the individual violations, it is useful to create a graph that displays the measured values. Figure 4.1 shows line plots for all six collective operations tested. For a better overview, we have highlighted the area where *pgchecker* would issue a violation in red and the area where there is no significant violation against the guidelines in green. Looking at the plot for *MPI_Scatter*, we can observe that violations occurred for the message sizes 512 B, 1 KiB, 2 KiB, and 8 KiB. If we compare the values obtained from the raw runtimes with Table 4.1, it can be seen that *pgchecker* has reported the correct violations for the data. For *MPI_Gather*, however, we notice a point at 4 KiB that falls very close to the boundary. Since we applied the *t-test* in this analysis, even this very minor violation is considered significant for a large number of runs. It is possible that this case would not be considered significant in other statistical tests, such as the *Wilcoxon Rank-Sum Test*. If we compare the *MPI_Bcast* plot with the result of *pgchecker*, we can also state that the four violations that occurred are correctly detected by *pgchecker* for this collective operation. In the plot for *MPI_Reduce*, we can recognize that the slowdown of the *default* implementation rises with an increasing

Figure 4.1: *Brave* slowdown comparison for the *OpenMPI, Version 4.1.4* implementation. Comparison of the runtime of the *default* algorithm to the *mockup-ups*. The higher the slowdown of the *default* algorithm, the more severe the violation found. If the slowdown of *default* compared to a *mockup-up* is in the red area, a violation against the guidelines occurs. In the green area no significant violation is detected. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 1$ core per node.

Table 4.2: *pgchecker* output table for *OpenMPI, Version 4.1.4* using the `GroupedViolationComparer`. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 2$ cores per node.

collective	m_size	nrep	def_med	slowd	mockup	alt_med
Allgather	512 B	5000	0.0769 ms	2.3766	Allgatherv	0.0324 ms
Allgather	1 KiB	5000	0.0940 ms	2.0499	Allgatherv	0.0459 ms
Allgather	2 KiB	5000	0.1195 ms	1.5894	Allgatherv	0.0752 ms
Allgather	4 KiB	5000	0.1909 ms	1.5031	Alltoall	0.1270 ms
Allgather	8 KiB	5000	0.4580 ms	2.2016	Alltoall	0.2080 ms
Allgather	16 KiB	4110	0.6853 ms	1.7797	Alltoall	0.3851 ms
Scatter	512 B	5000	0.0304 ms	1.3497	Scatterv*	0.0225 ms
Scatter	1 KiB	5000	0.0332 ms	1.2967	Scatterv*	0.0256 ms
Scatter	2 KiB	5000	0.0391 ms	1.2336	Scatterv*	0.0317 ms
Scatter	4 KiB	5000	0.0514 ms	1.0866	Scatterv*	0.0473 ms
Bcast	1 MiB	1978	1.4923 ms	1.2431	Scatter+Allgather	1.2005 ms
Bcast	8 MiB	257	11.6514 ms	1.3572	Scatter+Allgather	8.5849 ms
Reduce	64 KiB	5000	0.1180 ms	1.2629	Reducescatterblock +Gather	0.0934 ms
Reduce	128 KiB	5000	0.1812 ms	1.0668	Reducescatterblock +Gather*	0.1699 ms
Reduce	1 MiB	2014	1.4526 ms	1.3371	Reducescatter +Gatherv	1.0864 ms
Reduce	8 MiB	169	17.6875 ms	1.8316	Allreduce	9.6569 ms
Allreduce	16 KiB	5000	0.0727 ms	1.4342	Reducescatterblock +Allgather	0.0507 ms
Allreduce	64 KiB	5000	0.2568 ms	2.2319	Reducescatter +Allgatherv	0.1150 ms
Allreduce	128 KiB	5000	0.4027 ms	1.8413	Reducescatter +Allgatherv	0.2187 ms

message size and that two *mockup-up* algorithms execute in the half of the runtime. Finally, the plot *MPI_Allreduce* should be considered. We observe, with a message size of 64 KiB, that the fastest *mockup-up* is six times faster than the *default* implementation. In many further tests, especially with the collective operation *MPI_Allreduce*, we could observe a significant performance decrease of the *default* implementation in *Open MPI* at different message sizes.

Furthermore, we would like to consider the output of the same example for $n = 2$ cores per node. In total, we use $p = 36 \times 2$ processes. The output of *pgchecker* is shown in Table 4.2. Again, we detect several violations against the guidelines. This time we find less significant violations for higher runtimes. Calculating the violations *brave* we count 19 out of 72 possible violations, while for the *cautious* variant without

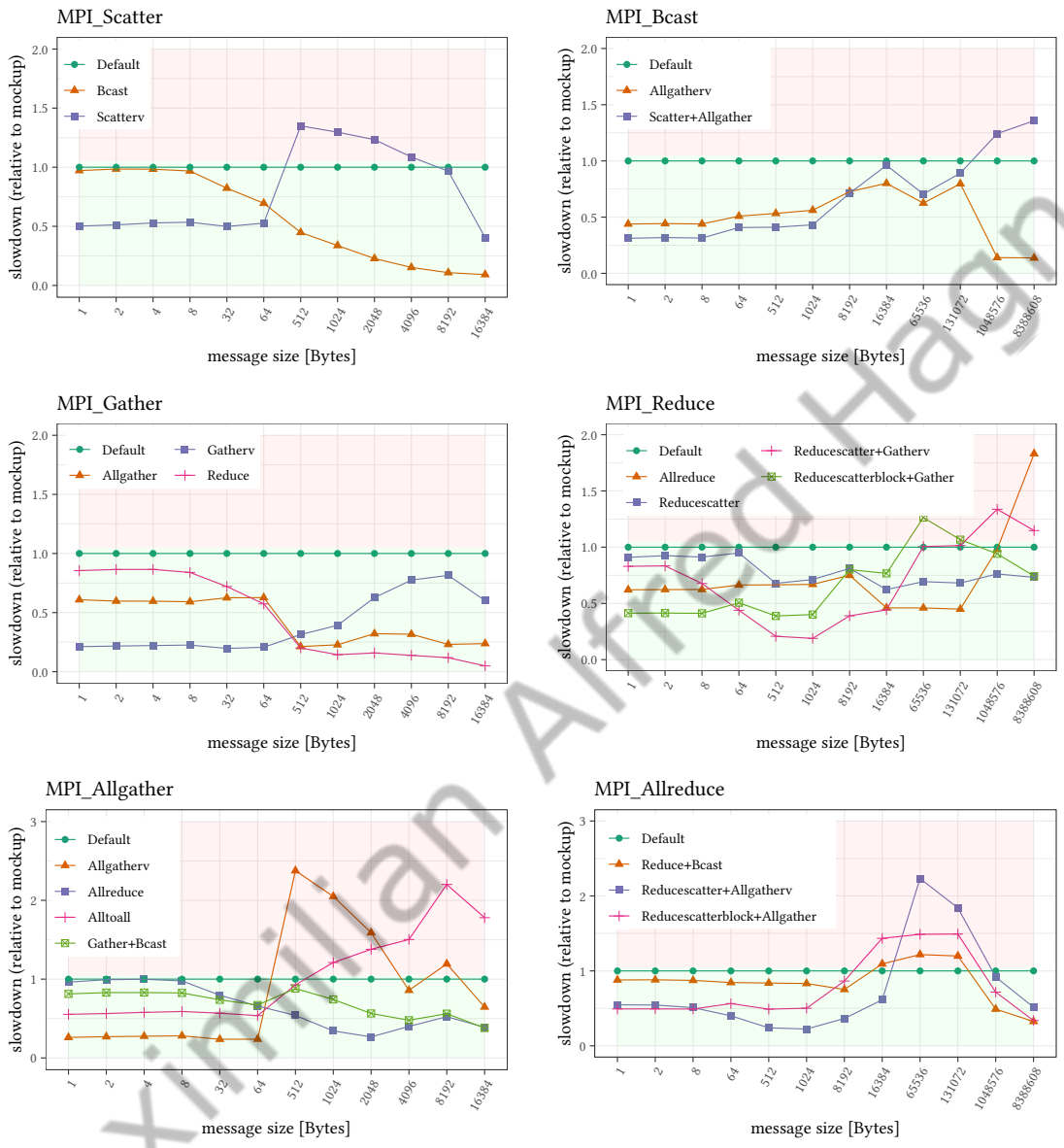

Figure 4.2: *Brave* slowdown comparison for the *OpenMPI, Version 4.1.4* implementation. Comparison of the runtime of the *default* algorithm to the *mockup-ups*. The higher the slowdown of the *default* algorithm, the more severe the violation found. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 2$ cores per node.

Table 4.3: *pgchecker* output table for *OpenMPI, Version 4.1.4* using the `GroupedViolationComparer`. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 32$ cores per node.

collective	m_size	nrep	def_med	slowd	mockup	alt_med
Allgather	512 B	2788	1.0233 ms	1.0181	Allgatherv*	1.0051 ms
Allgather	1 KiB	1284	2.2700 ms	1.1966	Allgatherv	1.8970 ms
Allgather	2 KiB	485	6.1125 ms	1.4876	Allgatherv	4.1089 ms
Scatter	512 B	4761	0.5887 ms	1.2443	Scatterv	0.4731 ms
Scatter	1 KiB	4232	0.6604 ms	1.1669	Scatterv	0.5659 ms
Bcast	64 KiB	5000	0.2189 ms	1.2373	Scatter+Allgather	0.1770 ms
Bcast	128 KiB	5000	0.4270 ms	1.3449	Scatter+Allgather	0.3175 ms
Bcast	8 MiB	114	25.2283 ms	1.2956	Scatter+Allgather	19.4722 ms
Reduce	64 KiB	5000	0.1948 ms	1.2810	Reducescatterblock +Gather	0.1521 ms
Reduce	128 KiB	5000	0.3191 ms	1.2268	Reducescatterblock +Gather	0.2601 ms
Reduce	1 MiB	840	3.4859 ms	1.1915	Reducescatterblock +Gather	2.9257 ms
Reduce	8 MiB	84	35.4844 ms	1.3348	Reducescatter +Gatherv	26.5844 ms
Allreduce	8 KiB	5000	0.1384 ms	1.7475	Reduce+Bcast	0.0792 ms

MPI_Barrier warnings, we count 14 out of 72. However, a similar pattern as already seen in Table 4.1 can be observed. What is noticeable, however, is that the slowdowns seem to have leveled out and we no longer encounter extremely large slowdowns. The most notable violation is the one of *MPI_Allgather* at a message size of 512 B. In this case, the *mockup-up* from *MPI_Allgatherv* is more than two times faster than the *default* implementation. Also, for this test run, the line plots of the individual *mockup-ups* can be seen in Figure 4.2. First, we again want to validate for *MPI_Scatter* and *MPI_Bcast* whether *pgchecker* returned the correct results. Indeed, in Table 4.2, there are four violations for *MPI_Scatter* and two violations for *MPI_Bcast* at the correct message sizes. For these two collectives, we can further observe a similar pattern as for the run with $p = 36 \times 1$ processes. Looking at *MPI_Gather*, we can state that no violation occurred and this collective has a solid *default* implementation. In the plot representing the slowdown of the *default* algorithm of *MPI_Reduce*, it can again be seen that the slowdown relative to the *mockup-up* implementation from *MPI_Allgather* increases with larger message sizes. For *MPI_Allgather* and *MPI_Allreduce*, it can be observed that the violations occur mainly in a certain message size range, but decrease again at higher message sizes.

Figure 4.3: *Brave* slowdown comparison for the *OpenMPI, Version 4.1.4* implementation. Comparison of the runtime of the *default* algorithm to the *mockup-ups*. The higher the slowdown of the *default* algorithm, the more severe the violation found. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 32$ cores per node.

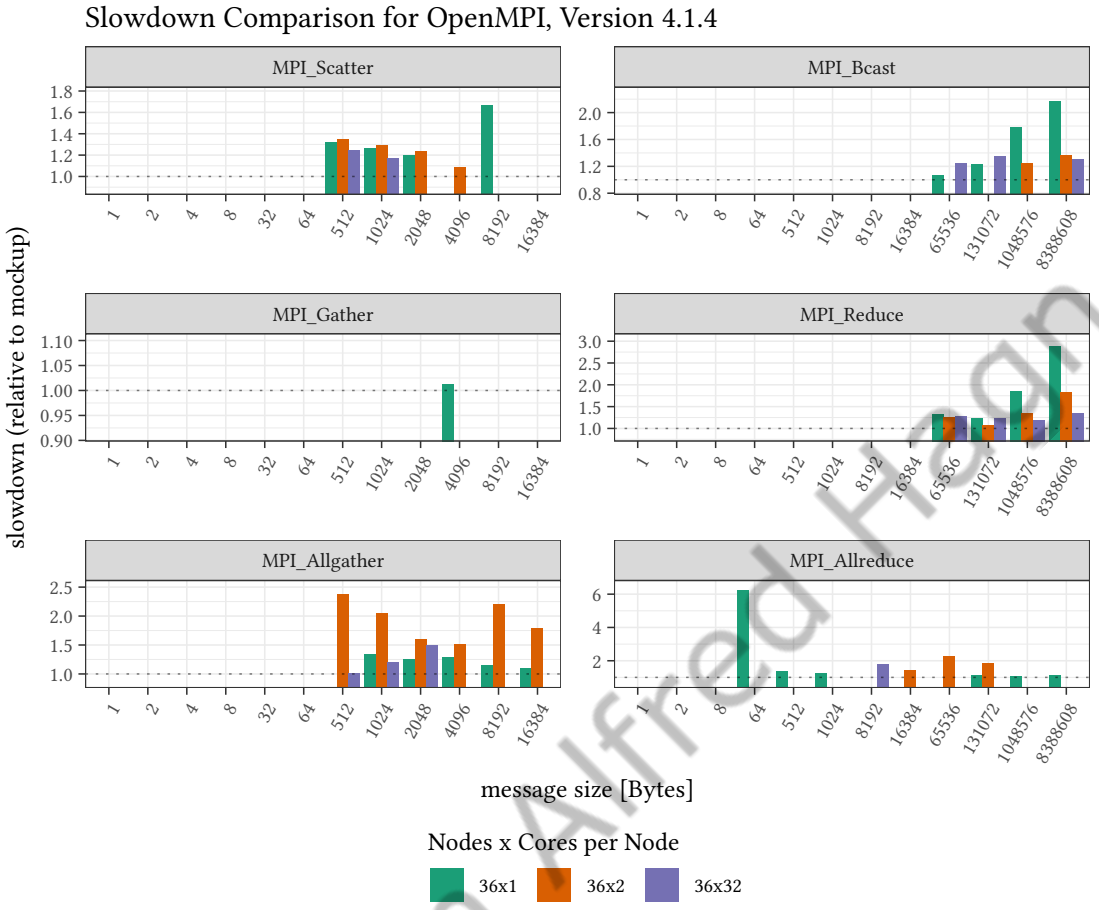

Figure 4.4: *Brave* slowdown comparison for the *Open MPI* implementation. Comparison of the slowdown of the *default* algorithm to the *mockup-ups* by message sizes. The higher the slowdown, the more severe the violation found. Always the *mockup-up* with the shortest runtime is considered. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 1$, $n = 2$, and $n = 32$ cores per node.

Lastly, we would like to look at the same test using $N = 36$ compute nodes and $n = 32$ cores per node. The table generated by *pgchecker* is demonstrated in Table 4.3. We can notice that much less violations are found. Only one *MPI_Barrier* warning is issued and thus we can note that a total of 13 violations using the *brave* evaluation and 12 using *cautious* are found out of 72 possible combinations. It can be observed that the most severe violation occurs for *MPI_Allreduce* at a message size of 8 KiB. The respective line plots of the different collectives for this test can be seen in Figure 4.3. Again, a quick check of the functionality of *pgchecker* reveals that in both, the raw data and in *pgchecker*, two significant violations are found for *MPI_Scatter* and three significant violations for *MPI_Bcast*. In the plot representing the slowdown comparison

for the *MPI_Scatter* operation, the same pattern as observed before is noticeable. Using a message size of 512 B, there is a sharp increase to a very high slowdown, which decreases again for even larger message sizes. For *MPI_Bcast*, the *mockup-up* consisting of *MPI_Scatter* and *MPI_Allgather* speeds up with larger message sizes relative to the *default* implementation, and the slowdown thus increases. Again, no violation could be recognized for the algorithms of *MPI_Gather*. For *MPI_Reduce*, it can be noticed that also for $p = 36 \times 32$ processes several violations occur at larger message sizes. With *MPI_Allgather* again, a peak in slowdown between a message size of approximately 0.5 KiB and 4 KiB is visible. What we did not see in the test with $p = 36 \times 2$ processes for *MPI_Allreduce* is the peak of slowdown for an arbitrary message size. As already seen in Figure 4.1, this peak can again be detected at *MPI_Allgather* for a message size of 8 KiB.

We would like to present the collected data of the performances of the *Open MPI* implementation with $N = 36$ compute nodes and $n = 1$, $n = 2$, and $n = 32$ cores per node on *Hydra* further aggregated in one graph. The aim is to better analyze which collective operations most frequently violate the guidelines. For this purpose, let us consider Figure 4.4. For each collective operation, a bar graph grouped by the number of total processes is shown. The higher the slowdown, the more severe the violation against the guidelines. As already gathered from the tables, there are almost no violations for *MPI_Gather*, and we can conclude that the *default* algorithm of this operation has been implemented well. Also, for *MPI_Allreduce*, except for a few outliers, few violations are found and these are rather of minor concern. For *MPI_Bcast* and *MPI_Reduce*, it is noticeable that the violations occurred mainly for larger message sizes. For *MPI_Allgather*, not only the most, but also the most severe violations are found. Especially, using $N = 36$ compute nodes and $n = 32$ cores per node several severe violations are found in the medium message size range.

Thus, we have completed the analysis of the *Open MPI* implementation on *Hydra* and familiarized ourselves with the workflow of the analysis with *pgchecker*. Furthermore, we were able to verify the tables generated by *pgchecker* for correctness by comparing them to the plots generated from raw runtime data. In the next section, we will decrease the level of detail and look at the results of the other *MPI* implementations on *Hydra* and *Irene*.

4.3 *MPI* Implementation Analysis

In this section, we would like to present the results of several tests of *MPI* implementations conducted on the two systems *Hydra* and *Irene*. Through the analysis, we intend to check whether the *MPI* implementations *Intel® MPI*, *MPICH*, *MVAPICH*, and *Open MPI* are of high quality and which implementations could still be improved. For this purpose, we will separately review the individual *MPI* implementations and highlight violations and anomalies found. Finally, we will compare the results of the different implementations and draw a conclusion about their performance.

4.3.1 Methodology Used in the Experiments

We will carry out the analysis on the systems *Hydra* and *Irene*. What is the same for both is that we will test the collective operations *MPI_Allgather*, *MPI_Allreduce*, *MPI_Bcast*, *MPI_Gather*, *MPI_Reduce*, and *MPI_Scatter*. Furthermore, in all experiments we will consider twelve different message sizes. Thus, in total, there are always $|collectives| * |message\_sizes| = 72$ possible combinations for which *pgchecker* can detect violations against the performance guidelines. For *MPI_Allgather*, *MPI_Gather*, and *MPI_Scatter*, we use the sizes 1 B, 2 B, 4 B, 8 B, 32 B, 64 B, 512 B, 1 KiB, 2 KiB, 4 KiB, 8 KiB, 16 KiB since the message size of these operations scales with the number of processes. For *MPI_Allreduce*, *MPI_Bcast*, and *MPI_Reduce*, the message size is not scaled and thus we will use a set of larger sizes: 1 B, 2 B, 8 B, 64 B, 512 B, 1 KiB, 8 KiB, 16 KiB, 64 KiB, 128 KiB, 1 MiB, 8 MiB. Moreover, we use the same input file we have already presented in Listing 4.2. For these experiments, we have adjusted the configuration file so that the lane and hierarchical topology *mockup-ups* are not tested. Instead, we focus on the pattern *mockup-ups* in this thesis.

The first machine we will use for testing is the *Hydra* system. For the tests performed on *Hydra*, we will use $N = 36$ compute nodes and $n = 1$, $n = 2$, and $n = 32$ cores per node. On the *Hydra* system, we will test the *MPI* implementations *Intel®MPI*, *MPICH*, and *MVAPICH*. We have already shown the evaluation of the *Open MPI* implementation on *Hydra* in the previous section. The second machine we will use for testing is the *Irene* system. We will test *Intel®MPI* and *Open MPI* on *Irene* utilizing $N = 128$ and $N = 256$ compute nodes and $n = 1$, $n = 2$, and $n = 48$ cores per node. In total, six different amounts of processes for each implementation.

4.3.2 Results of the *MPI* Implementation Analysis

In this section, we would like to present the results of the experiments. Therefore, we will remain using our *Hydra* system and look at the *MPI* implementations *Intel®MPI*, *MPICH*, and *MVAPICH* on up to $p = 36 \times 32$ processes. The first implementation we will consider is *MVAPICH, Version 2.3.7*. The results seen in Figure 4.5 are taken directly from the *csv*-files created by *pgchecker*. When looking at the first plot for *MPI_Scatter*, it become noticeable that the violations are found mainly in the lower message size ranges. Considering the differences between the individual process numbers, it can be observed that the number of violations decreases for a higher number of processes. We were already able to observe this behavior in the analysis of the *Open MPI* implementation, and this can also be explained by the fact that *MPI_Barrier* warnings occur more frequently for shorter runtimes, which we have included in this graph. It also shows that there are no significant violations for message sizes larger than 512 B for the *MPI_Scatter* operation. Next, if we consider *MPI_Bcast*, we notice that only four violations of $|environments| * |message\_sizes| = 36$ occurred. These violations occur at higher message sizes of 64 KiB and 8 MiB. It is interesting to note that the slowdowns for $n = 2$ cores per node are larger than those for $n = 1$ core per node tests. For $n = 32$ cores per node, no violations for *MPI_Bcast* are found at all. For *MPI_Gather*, we detect

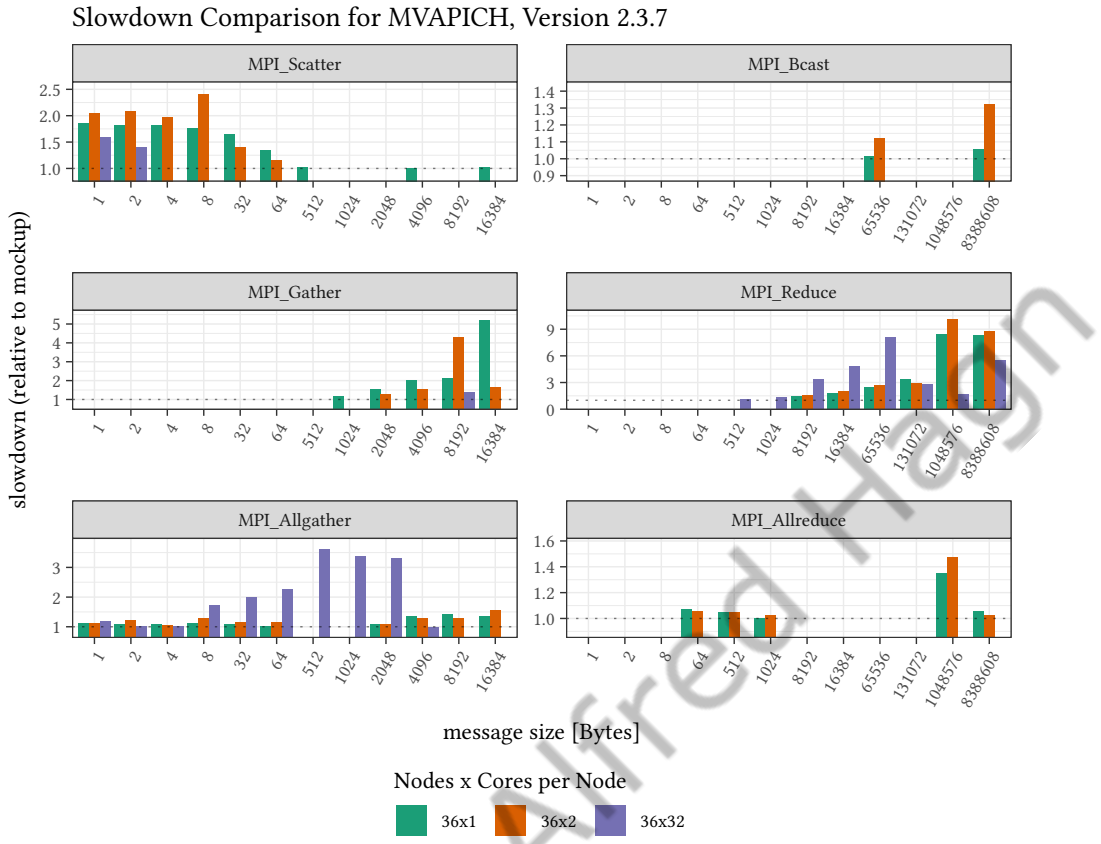

Figure 4.5: *Brave* slowdown comparison for the *MVAPICH* implementation. Comparison of the slowdown of the *default* algorithm to the *mockup-ups* by message sizes. The higher the slowdown, the more severe the violation found. Always the *mockup-up* with the shortest runtime is considered. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 1$, $n = 2$, and $n = 32$ cores per node.

some violations for larger message sizes, with these occurring primarily for $p = 36 \times 1$ processes and $p = 36 \times 2$ processes. In the *MVAPICH* implementation, *MPI_Reduce* definitely encounters the most severe violations against the guidelines. Starting at a message size of 8 KiB, violations occurred in every test for all numbers of nodes and cores per node. For the worst violation, the *default* implementation is more than nine times slower than the *mockup-up* implementations. Focusing on *MPI_Allgather*, the *mockup-up* algorithms are consistently slightly faster than the *default* implementation for $p = 36 \times 1$ processes and $p = 36 \times 2$ processes. In the middle message size range, several violations occurred for $p = 36 \times 32$ processes. Finally, we can examine the plot for *MPI_Allreduce*. In this we notice rather few violations, and these occurred only for $p = 36 \times 1$ processes and $p = 36 \times 2$ processes. Finally, if we compare *MVAPICH* with *Open MPI*, it becomes clear that we could find more violations against the *MPI* performance guidelines for *MVAPICH*.

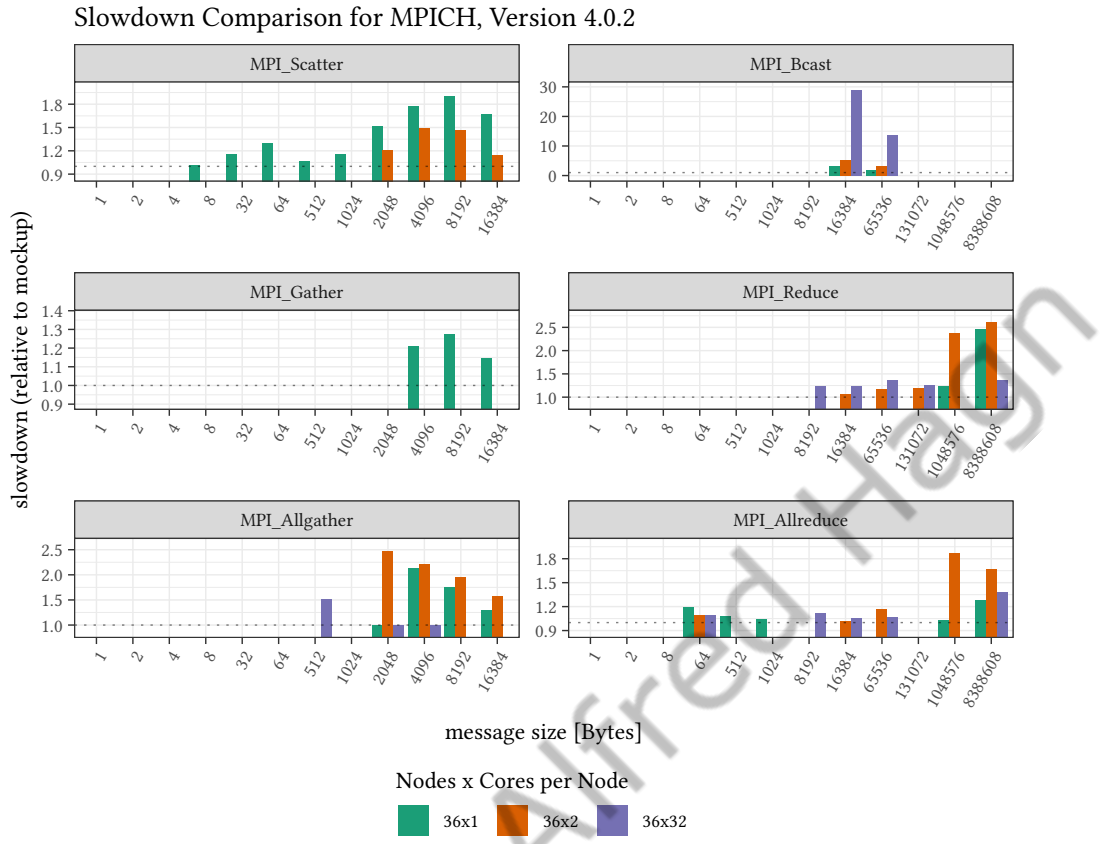

Figure 4.6: *Brave* slowdown comparison for the *MPICH* implementation. Comparison of the slowdown of the *default* algorithm to the *mockup-ups* by message sizes. The higher the slowdown, the more severe the violation found. Always the *mockup-up* with the shortest runtime is considered. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 1$, $n = 2$, and $n = 32$ cores per node.

Next, we would like to analyze the implementation *MPICH, version 4.0.2*, in more detail. For this purpose, we considered the plots of the individual collective operations which can be seen in Figure 4.6. Starting with *MPI_Scatter*, we observe that the violations occurred mainly for $p = 36 \times 1$ processes. Also, for $p = 36 \times 2$ processes there are several violations in the higher message size range. For $N = 36$ compute nodes and $n = 32$ cores per node, no violation against the guidelines occurred.

Looking at the plot for the collective operation *MPI_Bcast*, it becomes obvious that for a message size of 16 KiB and 64 KiB the *default* algorithm is up to thirty times slower than the *mockup-up*. Since this is the largest slowdown we will encounter in our analysis, we would like to examine this anomaly in more detail. Using the output of *pgchecker*, we could determine that the violating *mockup-up* is the sequence of *MPI_Scatter* and *MPI_Allgatherv*. We tested the collective operation *MPI_Bcast* again, testing just this

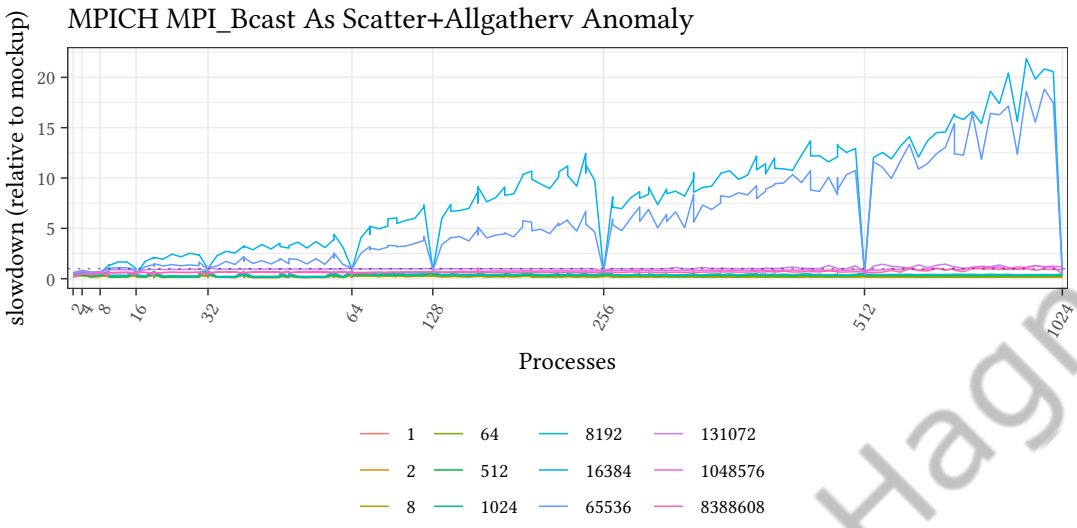

Figure 4.7: Detailed analysis of the *MPI_Bcast* anomaly in *MPICH*. Performance comparison of the *default* implementation of the collective operation *MPI_Bcast* against the *mockup-up* algorithm consisting of the sequence of *MPI_Scatter* and *MPI_Allgatherv*. The experiment was performed on the *Hydra* system utilizing a total of 160 different numbers of processes.

mockup-up for all message sizes and different numbers of processes. The result can be seen in Figure 4.7. It can be observed that violations really occur mainly at message sizes of 16 KiB and 64 KiB. The slowdown of the *default* implementation increases according to the number of processes used. What is particularly interesting is that no violations occur for processor numbers that are a multiple of two. For $p = 32 \times 32$ processes, it can be seen that the runtime of *default* is more than twenty times longer than the *mockup-up* implementation. It can be stated that we have clearly found an inefficient implementation of the *default* algorithm of *MPI_Bcast*.

Looking again at Figure 4.5, concerning the collective operation *MPI_Gather*, we recognize that only three violations occurred for larger message sizes at $p = 36 \times 1$ processes. For *MPI_Reduce*, we see a similar pattern as already observed for *MVAPICH*. For message sizes larger than 8 KiB, there are constant violations against the guidelines for all numbers of processes. In the plot for *MPI_Allgather*, we notice that especially for larger message sizes violations occur for $p = 36 \times 1$ processes and $p = 36 \times 2$ processes. If we finally analyze *MPI_Allreduce*, we observe minor violations distributed across all message sizes. In the higher message size range, the significance of the slowdown of the *default* implementation increases.

The last implementation we have tested on *Hydra* and would like to consider separately is *Intel ® MPI, Version 2021.7*. The results of each collective operation are shown in Figure 4.8. Starting again with *MPI_Scatter*, we recognize that there are violations

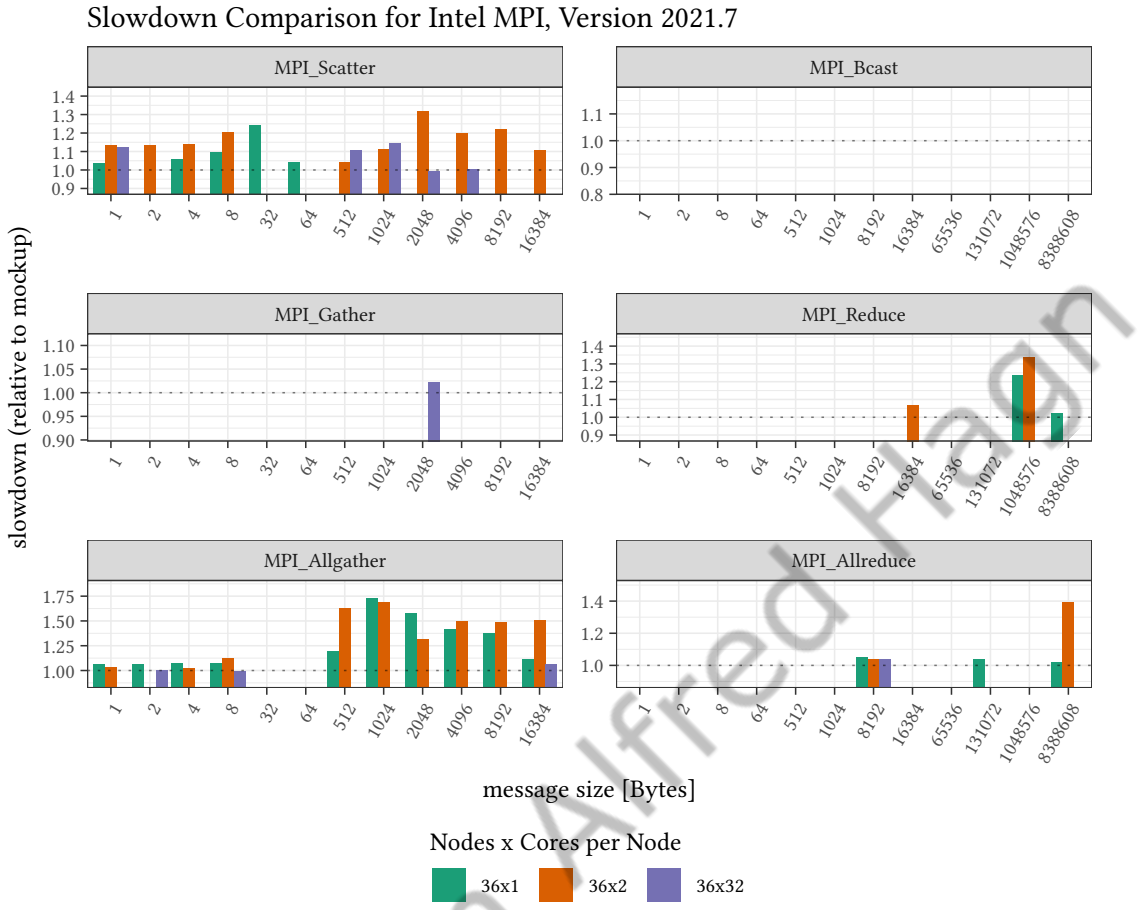

Figure 4.8: *Brave* slowdown comparison for the *Intel®MPI* implementation. Comparison of the slowdown of the *default* algorithm to the *mockup-ups* by message sizes. The higher the slowdown, the more severe the violation found. Always the *mockup-up* with the shortest runtime is considered. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 1$, $n = 2$, and $n = 32$ cores per node.

found consistently across all numbers of processes and for all message sizes tested. For *MPI_Bcast*, no violation is measured, leading us to conclude that a solid *default* algorithm is implemented. Also, for *MPI_Gather* only a very slight violation is found. In the plot for *MPI_Reduce* we observe, especially in comparison to the previously considered implementations, that much fewer violations are found. Utilizing a number of $p = 36 \times 32$ processes not even a single violation could be found. For the plot of the collective operation *MPI_Allgather*, we again recognize several violations. These occur mainly using $p = 36 \times 1$ processes and $p = 36 \times 2$ processes. The violations are more severe in the higher message size range. Finally, we can analyze *MPI_Allreduce*, where we also only detect one serious violation at 8 MiB. Recalling the results of the previous implementations, we must conclude that by far the fewest violations could be found in

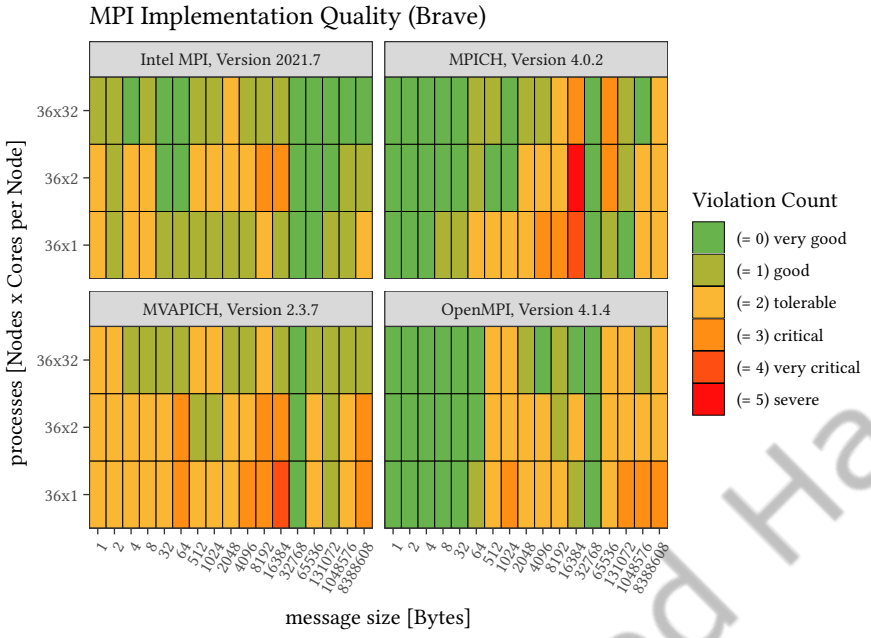

Figure 4.9: *Brave* quality comparison for *Intel* ®*MPI*, *MPICH*, *MVAPICH*, and *Open MPI*. Comparison of the violations detected across all collective operations by message size and number of cores used. For *Brave*, all violations found are counted. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 1$, $n = 2$, and $n = 32$ cores per node.

Intel ®*MPI*. To illustrate this, we would like to summarize the violations found in all four tested implementations in one graph and analyze them in more detail. We will distinguish between *brave*, the variant in which violations marked with *MPI_Barrier* warnings are also included, and *cautious*, the variant in which we have removed all violations marked with *MPI_Barrier* warnings.

In Figure 4.9, the evaluation according to the *brave* variant can be seen. Starting with *Intel* ®*MPI*, we notice that the algorithms with $p = 36 \times 32$ processes are implemented quite well. For a number of $p = 36 \times 2$ processes, it is interesting to note that there are more violations than for $p = 36 \times 1$ processes. Moreover, *Intel* ®*MPI* performed better with larger message sizes than with smaller ones. *MPICH*, performed better on smaller message sizes. Obviously, for message size 16 KiB and $p = 36 \times 2$ processes, it stands out that a violation occurred in five out of six collective operations tested. *MVAPICH* generally performed the worst with the smallest count of $p = 36 \times 1$ processes. For *MVAPICH*, we counted the most violations against the performance guidelines of all implementations. Especially for $p = 36 \times 1$ processes and $p = 36 \times 2$ processes, at least one violation of the guidelines occurred for almost every message size. With $p = 36 \times 32$ processes, the performance is better, but it is not as high as that of the competitors. *Open MPI* delivered solid performance across the board and is thus the

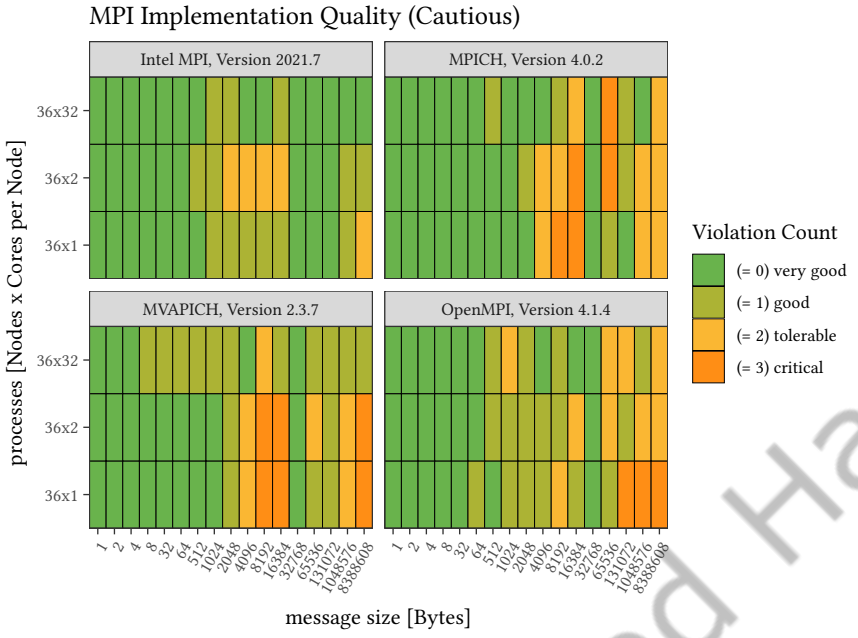

Figure 4.10: *Cautious* quality comparison for *Intel ® MPI*, *MPICH*, *MVAPICH*, and *Open MPI*. Comparison of the violations detected across all collective operations by message size and number of cores used. For *Cautious*, all violations subject to a *MPI_Barrier* warning are removed. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 1$, $n = 2$, and $n = 32$ cores per node.

best performer after *Intel ® MPI*. For *Open MPI*, it is interesting to note that it is not so much the number of processes, but rather the size of the message that determines the performance. For instance, with *Open MPI*, no message size less than or equal to 32 B violations occurred. Furthermore, all violations found for an implementation can be summed up. In total, there are 216 possible combinations, which can be multiplied out of 12 different message sizes, 6 collective operations and 3 process counts. For *Intel ® MPI*, with 53, the fewest violations are counted. The second fewest violations are found with a count of 56 for *Open MPI*. Followed by *MPICH* with 60 and *MVAPICH* with 91 out of 216 violations. We would like to adjust this diagram by removing the violations marked with *MPI_Barrier* warnings.

The result of the *cautious* evaluation can be seen in Figure 4.10. It is evident that we are counting fewer violations against the guidelines overall, as we remove those with *MPI_Barrier* warnings. It is also apparent that these violations are removed primarily in the smaller message size ranges. Even after the adjustment, we recognize that *Intel ® MPI* encountered the lowest number of violations. However, it is interesting to see that some violations occur for $p = 36 \times 2$ processes in the medium message size range. For *MPICH*, we can state that no more violations are found for a large number of tests in the lower message size range. However, there are still several violations in the upper message size

range. For *MVAPICH*, it can be observed that for a number of $p = 36 \times 2$ processes much less violations could be found. For $p = 36 \times 32$ processes, however, there is one violation for almost every message size. It is also interesting to note that some violations occur for message sizes of 8 KiB, 16 KiB, and 8 MiB. Finally, let us consider *Open MPI*. In contrast to *MPICH*, fewer violations are removed in the lower message size range, but we observe better performance in the upper range. Nevertheless, the number of violations found is still noticeable with low processes and a high message size.

In conclusion, we can count the total number of violations per *MPI* implementation. By far the best performance could be achieved by *Intel ® MPI* with only 23 violations out of a possible 216. *MPICH* follows *Intel ® MPI* with 39 violations found. For *Open MPI*, we could count 43 violations, and for *MVAPICH* 46 violations out of 216 possible. If we summarize the results of the *brave* and *cautious* evaluations, we find that *Intel ® MPI* outperformed the other competitors. *Open MPI* and *MPICH* performed about the same overall, each with an average of approximately 50 violations across both variants. In the *cautious* variant, we saw that *MVAPICH* in particular had a large number of violations with *MPI_Barrier* warnings removed. *MVAPICH* still performed the worst in both variants. In all implementations, we found that *MPI_Gather* had the best *default* implementations. Apart from the anomaly observed when using *MPICH*, the *default* algorithm of *MPI_Bcast* performed the second best. The *default* implementations for *MPI_Reduce* and *MPI_Allgather* produced the most violations overall.

Before concluding the analysis of the tests performed on the *Hydra* system, we would like to take a look at the violations found according to the number of compute nodes and the core per node. In Figure 4.11, the results are shown for both the *brave* variant and the *cautious* variant. If we look at the *brave* variant first, we notice that the number of violations of the guidelines decreased for all implementations with a higher number of processes. However, it is interesting to note that for *Intel ® MPI*, more violations occurred for $p = 36 \times 2$ processes than for $p = 36 \times 1$ processes. Again, in this plot we observe that *Intel ® MPI* performed the best overall and *MVAPICH* the worst. The implementations of *Open MPI* and *MPICH* encountered approximately the same number of violations. Obviously, overall we notice less violations for the *cautious* variant. However, it is noticeable that the number of violations only decreases for *Open MPI* with a greater number of processes. For all other implementations, worse values are recorded for $p = 36 \times 2$ processes than for $p = 36 \times 1$ processes. For $p = 36 \times 32$ processes, we still see the best performance in each implementation. Furthermore, we notice that the ratio between $p = 36 \times 1$ processes and $p = 36 \times 32$ processes remains about the same, even after removing the violations marked with *MPI_Barrier* warning. Finally, we can conclude that the *cautious* variant mainly removes violations for very small numbers of processes, which is especially evident for *MVAPICH*. We have evaluated all possible results on the *Hydra* system and would like to focus on the *Irene* system in the following.

On the *Irene* system, we would like to consider the implementation *OpenMPI, Version 4.1.4*, the same one we have already tested on *Hydra*, with up to $p = 256 \times 48$ processes. In Figure 4.12, we have created a separate heat-map for each number of processes,

Figure 4.11: Violation comparison by compute nodes and cores per node. Comparison of the violations found against the performance guidelines of the *Intel ® MPI*, *MPICH*, *MVAPICH*, and *Open MPI* implementations by number of compute nodes and cores per node. The experiment was performed on the *Hydra* system utilizing $N = 36$ compute nodes and $n = 1$, $n = 2$, and $n = 32$ cores per node.

indicating how large the slowdown of the *default* implementation is relative to the fastest *mockup-up* for each collective operation and message size. For this purpose, let us first consider the plot for $N = 128$ compute nodes and $n = 1$ core per node. It can be seen that for *MPI_Scatter* and *MPI_Gather*, no violations against the guidelines occurred. If we also analyze all other plots for *MPI_Gather*, we can see that no more than one violation against the guidelines could be found for any number of processes, and we can therefore already conclude that *MPI_Gather* has a solid *default* implementation, even for larger numbers of processes in *Open MPI*. For *MPI_Reduce* and *MPI_Bcast*, we detect severe violations at message sizes of 1 MiB and 8 MiB. For 16 KiB and 64 KiB we

recognize that the *default* algorithm had over twice the runtime of the *mockup-ups* for *MPI_Bcast* and *MPI_Allreduce*. If we look at the plot for $N = 128$ compute nodes and $n = 2$ cores per node, we see that severe violations only occur for the message sizes 1 MiB and 8 MiB for all collective operations. It is particularly interesting to compare this plot with the $N = 256$ compute nodes and $n = 1$ core per node variant, which also amounts to $p = 256 \times 1$ processes. We recognize that with $N = 128$ nodes 9 violations and with the $N = 256$ variant only 8 violations are found. However, the violations are more significant by using $N = 256$ nodes. For $N = 256$ compute nodes and $n = 2$ cores per node, the severe violations of the *MPI_Reduce* and *MPI_Bcast* operations are noticeable in the higher message size range. If we look at the tests for $N = 128$ and $N = 256$ compute nodes and $n = 48$ cores per node, we recognize a very similar pattern in both plots. Most noticeable are the severe violations for *MPI_Allgather*, which we do not recognize for smaller numbers of processes. These violations are more severe for the variant with $p = 256 \times 48$ processes than for the variant with $p = 128 \times 48$ processes, so it can be stated that the slowdowns of the *default* algorithms seem to increase with the number of processes.

In summary, we can conclude that *Open MPI* performs quite poorly in the upper message size implementations of *MPI_Reduce* and *MPI_Bcast*. For *MPI_Scatter*, we see some slight violations, especially at message sizes from 5 B to 8 KiB. For the collective operation *MPI_Allreduce*, we can find several severe violations in the medium message size range, especially for large process counts. Particularly severe violations could be observed constantly for *MPI_Allgather* with a large number of processes. For *MPI_Gather*, we could only find very few minor violations in *Open MPI* across all tests.

We would also like to perform this analysis for *Intel® MPI, Version 2021.5*. Looking at Figure 4.13, we immediately notice that we could find much less violations against the *MPI* performance guidelines. If we consider the tests using $N = 128$ and $N = 256$ compute nodes and $n = 1$ core per node, then the violations for *MPI_Allreduce* for the message sizes 1 MiB and 8 MiB are noticeable. For these violations, the *default* implementation had twice the runtime of the *mockup-ups*. For the $N = 128$ and $N = 256$ compute nodes and $n = 2$ cores per node variants, we see isolated, slight violations against the performance guidelines for the message sizes 16 KiB and 1 MiB for the collective operations *MPI_Scatter*, *MPI_Reduce*, and *MPI_Bcast*. For the variants with $p = 128 \times 48$ processes and $p = 256 \times 48$ processes, we observe similar violation patterns for these collective operations. However, what is also noticeable when using high process counts are the severe violations in *MPI_Allgather*, which we could also already observe in the *Open MPI* implementation. With *Intel® MPI*, however, these very serious violations are more likely to be detected at lower message sizes. It is also striking in all plots that there is not one violation for either *MPI_Gather* or *MPI_Allreduce* in any of the tests. In summary, we can state that we only found a few irregularities in *Intel® MPI*, and that only the collective operation *MPI_Allgather* shows performance weaknesses in the lower message size range.

We would like to compare the *MPI* implementations tested on the *Irene* system, as

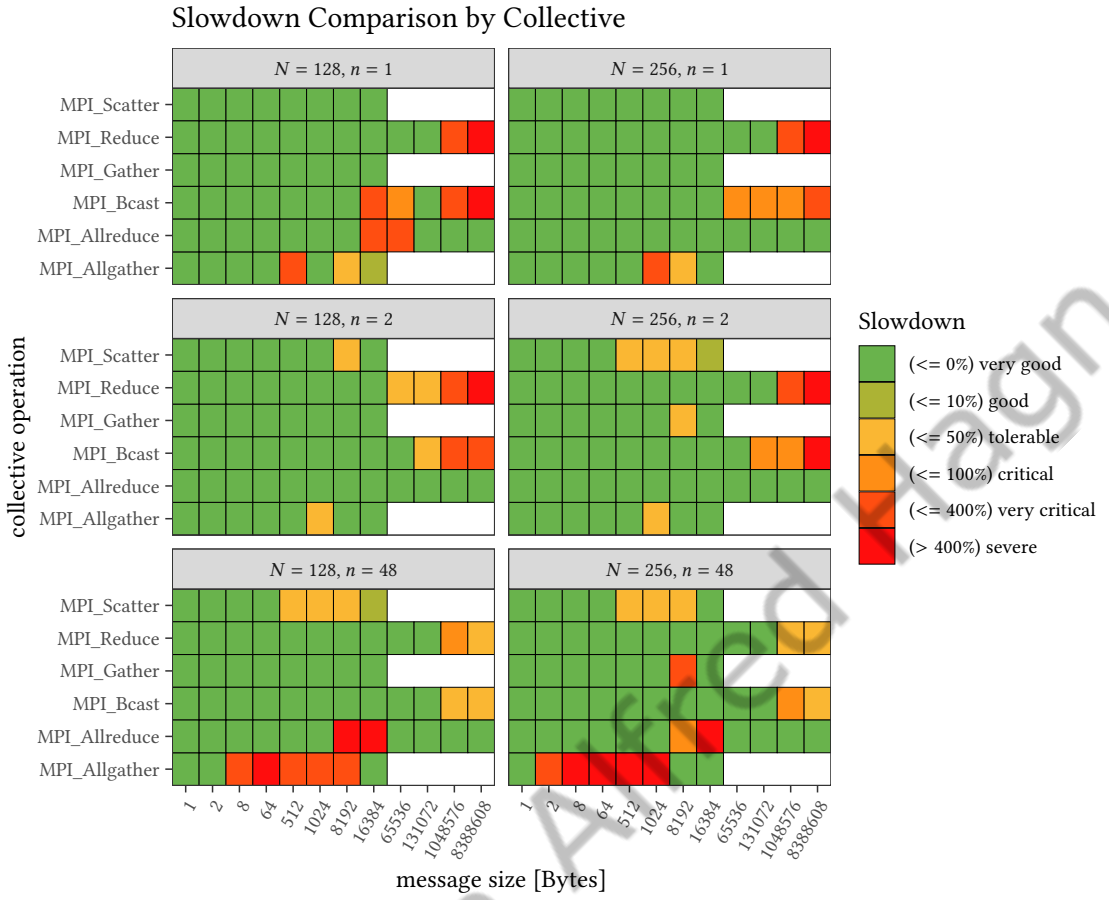

Figure 4.12: Slowdown comparison for *Open MPI* by collective operation. Comparison of the slowdown of the *default* algorithm to the *mockup-ups* by message sizes and collective operations. Always the *mockup-up* with the shortest runtime is considered. The results are *cautious* and thus do not contain any violations which are subject to a *MPI_Barrier* warning. The experiment was performed on the *Irene* system utilizing $N = 128$ and $N = 256$ compute nodes and $n = 1$, $n = 2$, and $n = 48$ cores per node.

we have already done for the implementations tested on the *Hydra* system. For this purpose, we can consider the results of *brave* evaluation presented in Figure 4.14. At first glance, it is obvious that *Intel ® MPI* performed better overall. Looking at *Intel ® MPI* separately, we see that there are fewer violations against the guidelines for large message sizes than for smaller ones. No particular pattern is visible for the different numbers of processes. What is noticeable, however, is the poor performance for $N = 256$ compute nodes and $n = 1$ core per node. It is also interesting that this variant performs worse than the pendant with $N = 128$ compute nodes and $n = 2$ cores per node, which also results in $p = 128 \times 2$ processes used. By looking at the heat-map for the *Open MPI* implementation, it is obvious that it performs better for smaller message sizes than

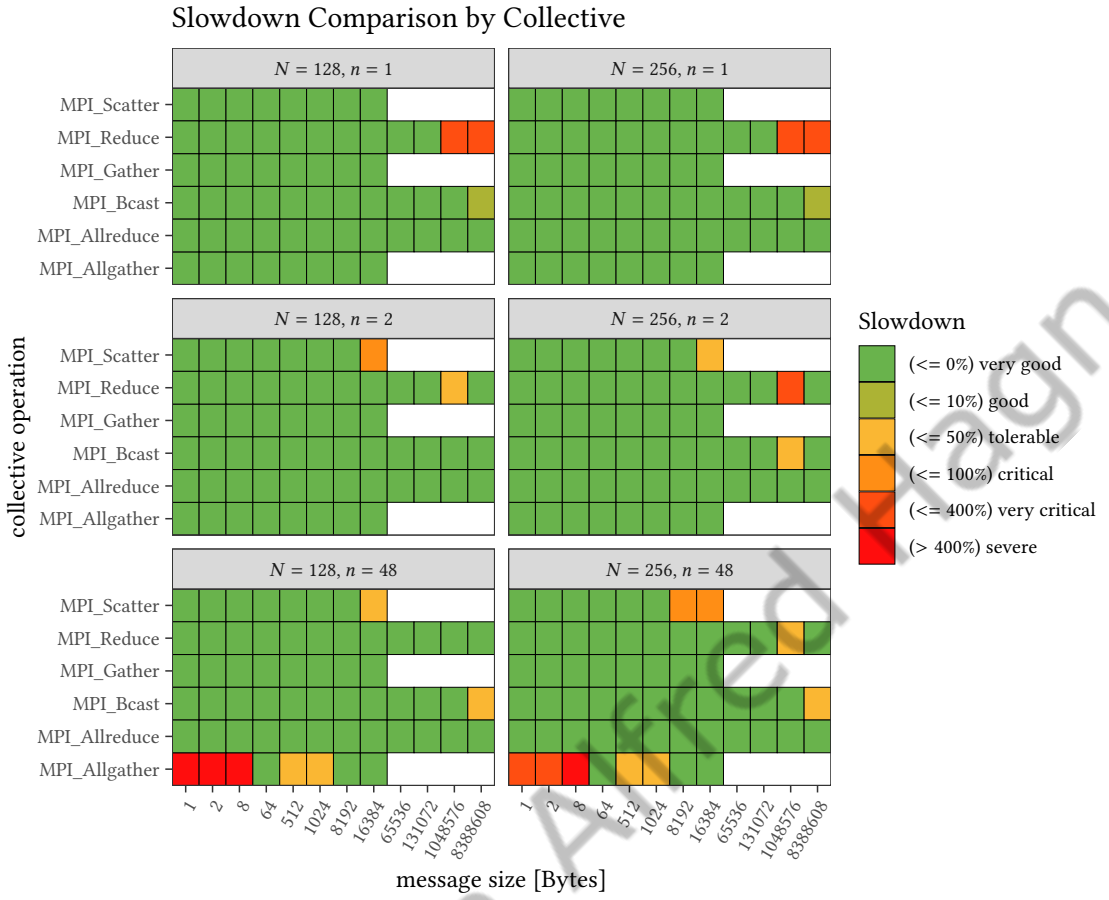

Figure 4.13: Slowdown comparison for *Intel®MPI* by collective operation. Comparison of the slowdown of the *default* algorithm to the *mockup-ups* by message sizes and collective operations. Always the *mockup-up* with the shortest runtime is considered. The results are *cautious* and thus do not contain any violations which are subject to a *MPI_Barrier* warning. The experiment was performed on the *Irene* system utilizing $N = 128$ and $N = 256$ compute nodes and $n = 1$, $n = 2$, and $n = 48$ cores per node.

for larger message sizes. Particularly noticeable are the many violations in the middle message size range. Comparing the two variants which amount to a total of $p = 256 \times 1$ processes, we do not see any major differences, although more violations occur using only one nodes. In summary, we can have a look at the total number of violations found. In total, there are 432 possible combinations, which can be multiplied out of 12 different message sizes, 6 collective operations and 6 process counts. For *Intel®MPI*, we found fewer violations amounting to 88. For *Open MPI*, we are able to find 157 out of 432 violations using the *brave* evaluation. Next, we would like to consider the results of the *cautious* evaluation.

The corresponding diagram for the *cautious* evaluation can be seen in Figure 4.15. We

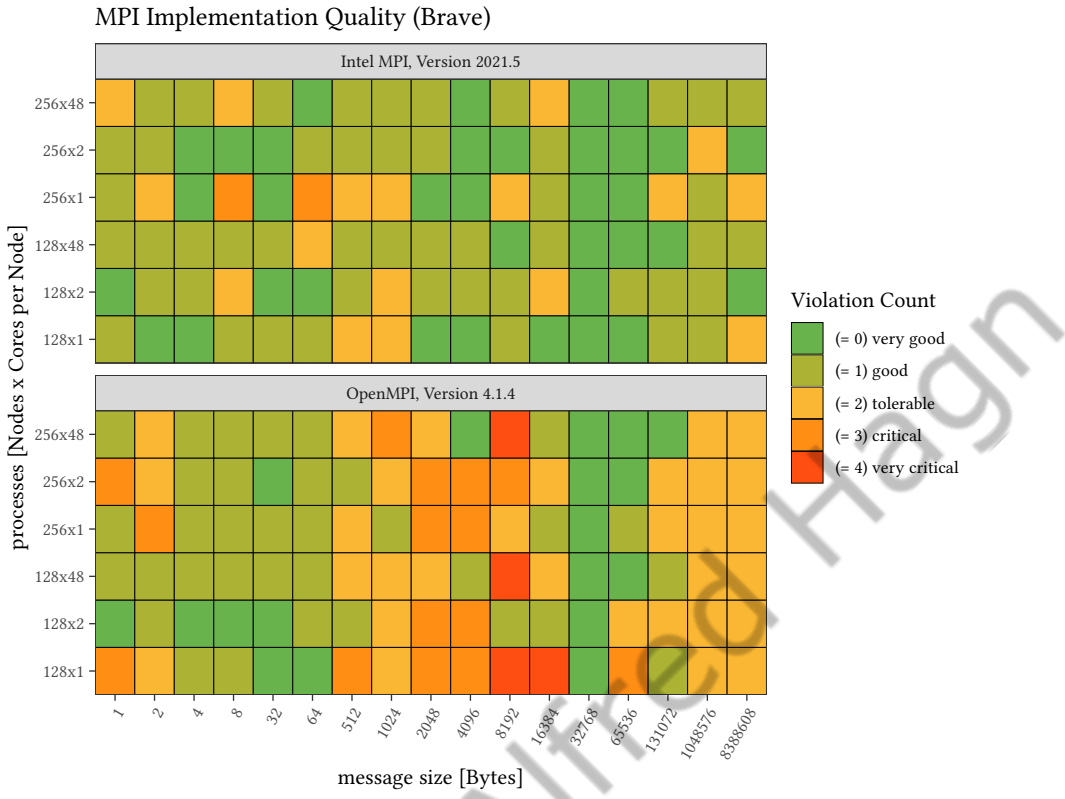

Figure 4.14: *Brave* quality comparison for *Intel ® MPI* and *Open MPI*. Comparison of the violations detected across all collective operations by message size and number of cores used. The experiment was performed on the *Irene* system utilizing $N = 128$ and $N = 256$ compute nodes and $n = 1$, $n = 2$, and $n = 48$ cores per node.

consider this to be a summary of Figure 4.12 and Figure 4.13, which we have already analyzed. Regarding the heat-map for the *Intel ® MPI* implementation, it can be stated that most violations occur at larger message sizes. Furthermore, it stands out that the most violations could be found for the variants with $N = 128$ and $N = 256$ compute nodes and $n = 48$ cores per node. For *Open MPI*, it is evident that only few violations could be found for small message sizes. However, we see more violations for message sizes ranging from 512 B to 16 KiB and for the message sizes 1 MiB and 8 MiB. Again, for *Open MPI* it can be seen that most of the violations occurred at high process numbers. If we again consider the total violations for both *MPI* implementations, then we can state that we could find 35 out of 432 possible violations using the *cautious* evaluation for *Intel ® MPI*. For the *Open MPI* implementation, we could find 88 violations out of 432. In conclusion, for the quality comparison of high amounts of nodes and cores per node of the implementations, we can state that *Intel ® MPI* performed better than *Open MPI* in both evaluation methods. We are able to arrive at the same conclusion when analyzing on our *Hydra* system, on which we performed tests with a lower number of processes.

Figure 4.15: *Cautious* quality comparison for *Intel ® MPI* and *Open MPI*. Comparison of the violations detected across all collective operations by message size and number of cores used. The experiment was performed on the *Irene* system utilizing $N = 128$ and $N = 256$ compute nodes and $n = 1$, $n = 2$, and $n = 48$ cores per node.

Finally, we would like to evaluate the contrasts between the different numbers of compute nodes and cores per node. We can consider Figure 4.16, where the detected violations are grouped by the number of processes. First, we examine the *Intel ® MPI* implementation using the *brave* evaluation. It is noticeable that most violations occurred using $N = 256$ compute nodes and $n = 1$ core per node. We have already discussed that it is interesting that this variant has performed worse than the other variant which also amounts to $p = 256 \times 1$ processes. The fewest violations are found in the variant using $N = 256$ compute nodes and $n = 2$ cores per node. It can be seen that using the *brave* evaluation, it is not the highest process counts that generate the most violations. Looking at the plot for *Open MPI*, we see that the most violations are found for $N = 128$ compute nodes and $n = 1$ core per node. It is also intriguing that the variant using $p = 256 \times 48$ processes performed better than the variant using $p = 128 \times 48$ processes. Also, for *Open MPI*, it can be stated that the variant of the tests with $p = 128 \times 2$ processes performed better than using $p = 256 \times 1$ processes.

Figure 4.16: Violation comparison by compute nodes and cores per node. Comparison of the violations found against the performance guidelines of the *Intel ® MPI* and *Open MPI* implementations by number of compute nodes and cores per node. The experiment was performed on the *Irene* system utilizing $N = 128$ and $N = 256$ compute nodes and $n = 1$, $n = 2$, and $n = 48$ cores per node.

If we look at the results from the *cautious* evaluation, we recognize that all tests up to $p = 256 \times 2$ processes produced approximately the same number of violations for the *Intel ® MPI* implementation. At $p = 128 \times 48$ processes, we notice the second highest number of violations, whereas at $p = 256 \times 48$ processes, the most violations could be counted. For *Open MPI*, it should also be mentioned that more violations are found in the test using $p = 128 \times 48$ processes than in the variant with $p = 256 \times 48$ processes. In summary, it can be stated that the number of violations mainly rise in the *cautious* evaluation according to the number of processors used, whereby this development stabilizes at higher numbers of processes, and in some cases even a decrease can be observed. We have completed the analysis on the *Irene* system and would like to summarize the most important findings again in the next section.

4.4 Summary of the Results

In this chapter, we analyzed the *MPI* implementations *Intel®MPI*, *MPICH*, *MVAPICH*, and *Open MPI* in detail for violations against the *MPI* performance guidelines. For this analysis, we considered the implementations executed on $N = 36$, $N = 128$, and $N = 256$ compute nodes and $n = 1$, $n = 2$, $n = 32$, and $n = 48$ cores per node. Across all tests, we noticed that *Intel®MPI* consistently delivered the best performance. On *Hydra*, we found that *Open MPI* and *MPICH* always produced about the same amount of guideline violations. For the *MVAPICH* implementation, we had to count the most violations across all tests.

Regarding the individual collective operations, we noticed that the *default* algorithm of the collective operation *MPI_Gather* performs best in all implementations. This operation is followed by the collective operations *MPI_Bcast* and *MPI_Allreduce* with the second and third fewest violations counted. However, we found a faulty implementation of the *default* algorithm in the *MPICH* implementation, for which we measured a slowdown of over thirty times. For *MPI_Scatter* and *MPI_Reduce*, we consistently found violations across all tests. We observed that *MPI_Allreduce* has the worst *default* algorithm in all implementations, performing poorly at both low and high process counts.

Moreover, it is also worth mentioning that when we analyzed the tests on *Hydra*, we noticed that a higher number of processes implies that fewer violations could be found. On *Irene*, however, we achieved the opposite result and saw that above a certain number of processes, an increased number of processes results in more violations counted.

Finally, we would like to point out that the violations found are always relative to the tested *MPI* implementation. This means that we looked at the number of violations, but not on how fast the implementations are compared to each other. Thus, we cannot conclude from our analysis that *Intel®MPI* has the fastest runtime, only that the fewest violations occurred within the implementation. There are other parameters that must be considered to determine the overall performance of an implementation. For instance, the startup time of an implementation on a supercomputer is often of interest. In our tests, we found that *Intel®MPI* does not provide the fastest startup time and that other competitors perform better. In summary, the number of detected violations against the *MPI* performance guidelines is not the only factor that determines the quality of an implementation.

CHAPTER 5

Conclusion

In this thesis, we presented the tool *pgchecker*, which can be used to automatically scan *MPI* implementations for violations against self-consistent *MPI* performance guidelines. In order to develop the tool, we first introduced *MPI* and the performance guidelines. Afterwards, we considered the software we could use in order to carry out parts of the analysis. In this section, we introduced the *PGMPITuneLib* library and discussed its functionality. Subsequently, we showed the way our developed *pgchecker* tool works and where it needs to access the *PGMPITuneLib* library. We also reviewed all the arguments that can be passed on to our tool, and the various comparisons and statistical tests in detail. Once we gathered all the underlying information about the *pgchecker* tool, we used *Open MPI* as an example to show how the analysis workflow is performed using our system. We applied this workflow to check the *MPI* implementations *Intel ® MPI*, *MPICH*, *MVAPICH*, *Open MPI* for violations against the performance guidelines, first on the smaller *Hydra* machine and then on the 133rd fastest supercomputer in the world, *Irene*, using our *pgchecker* tool. In the course of these tests, we were able to find out that our tool is able to generate the correct results from the raw runtime data for all inputs. We could also determine that the *MPI* implementation from *Intel* performed at the highest level of quality of the tested candidates. Additionally, we found that there is a faulty *default* implementation of the collective operation *MPI_Bcast* in *MPICH* and that *MPI_Gather* has by far the best implemented *default* algorithm in all tested *MPI* implementations.

As mentioned in different sections of this thesis, there are limitations of the tool that are not included in the research question. It is particularly crucial to mention that the different comparisons and statistical tests have been developed towards our personal interests. However, in this thesis, we have already shown that *pgchecker* is developed using an object-oriented design, and therefore the user can modify individual parts of the tool or add new comparisons and statistical tests.

Furthermore, it can be considered that the most advanced comparison of *pgchecker* provides a table of violations against the guidelines, always indicating the violation of the fastest *mockup-up*. Apart from this information, it would be desirable if *pgchecker* outputs further information such as the total number of violations found for the *brave* and for the *cautious* evaluation, the worst and the best performing collective operation or rankings of the most severe slowdowns. Additionally, it would be conceivable that a quality score is implemented which indicates the performance of the used *MPI* implementation. This score could be compared to other executions with different numbers of compute nodes or cores per node of the same or even compared to other *MPI* implementations.

Moreover, it can be stated that *pgchecker* is limited by the algorithms implemented by the *PGMPITuneLib* library. In other words, using *pgchecker* we can only test algorithms that have already been implemented in the *PGMPITuneLib* library. In this thesis, we have analyzed several of the algorithms implemented in *PGMPITuneLib*, but also discussed that there are even more *mockup-up* algorithms available in the *PGMPITuneLib* library. However, besides those already implemented, there are a variety of other possible *mockup-up* implementations that could also be analyzed for violations against the guidelines. This variety of further algorithms is not limited to the *mockup-ups* defined by the pattern guidelines, but also alternative implementation approaches for algorithms, environment-specific algorithms, or specialized communication topologies could be defined as *mockup-ups*. To give an example, we could also analyze how well the point-to-point communication functions defined in *MPI* perform, such as *MPI_Send* and *MPI_Recv*, compared to the collective operations.

Finally, regarding the analysis of the *MPI* implementations we performed ourselves, we limited ourselves to certain collective operations and some message sizes. For instance, the function *MPI_Alltoall* is also implemented in the *PGMPITuneLib* library and thus also available for *pgchecker*. We could have checked this operation as well, but we decided to limit ourselves to the six fundamental collective operations. Additionally, during the analysis we pointed out that we removed algorithms that refer to different communication topologies from the configuration document. This allowed us to exclusively focus on the pattern guidelines in this thesis. The lane and hierarchical topologies implemented in the *PGMPITuneLib* library also could have been considered.

Bibliography

[1] S. Pellegrini, R. Prodan, and T. Fahringer, "Tuning MPI runtime parameter setting for high performance computing," in *2012 IEEE International Conference on Cluster Computing Workshops*, pp. 213–221, 2012.

[2] S. Hunold, A. Carpen-Amarie, F. Lübbe, and J. Träff, "Automatic verification of self-consistent MPI performance guidelines," in *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing*, pp. 433–446, 08 2016.

[3] J. Larsson Träff, W. D. Gropp, and R. Thakur, "Self-consistent MPI performance guidelines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 5, pp. 698–709, 2010.

[4] S. Hunold and 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 2018, (New York, NY, USA), p. 64–74, Association for Computing Machinery, 2018.

[5] "Open MPI: Open Source High Performance Computing." `https://www.open-mpi.org/`. Accessed: 2023-02-26.

[6] "MPICH | High-Performance Portable MPI." `https://www.mpich.org/`. Accessed: 2023-03-05.

[7] "MVAPICH :: Home." `https://mvapich.cse.ohio-state.edu/`. Accessed: 2023-03-05.

[8] "Intel® MPI library." `https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html#gs.rpm57a`. Accessed: 2023-03-05.

[9] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[10] "MPI forum - background." `https://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/node2.htm#Node2`. [accessed: 11-February-2022].

[11] J. L. Träff and S. Hunold, "Decomposing MPI collectives for exploiting multi-lane communication," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 270–280, 2020.

[12] S. Hunold and A. Carpen-Amarie, "Reproducible MPI benchmarking is still not as easy as you think," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 1–1, 12 2016.

[13] "CMake." `https://cmake.org/`. Accessed: 2023-02-26.

[14] "GSL - GNU scientific library - GNU project - free software foundation." `https://www.gnu.org/software/gsl/`. Accessed: 2023-02-26.

[15] S. Hunold and A. Carpen-Amarie, "Hierarchical clock synchronization in MPI," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 325–336, 2018.

[16] "JOLIOT-CURIE SKL, mellanox edr | TOP500." `https://www.top500.org/system/179411/`. Accessed: 2023-03-10.

[17] "November 2022 | TOP500." `https://www.top500.org/lists/top500/2022/11/`. Accessed: 2023-03-10.