Reproducible MPI Benchmarking Is Still Not As Easy As You Think

Sascha Hunold and Alexandra Carpen-Amarie

Abstract—The Message Passing Interface (MPI) is the prevalent programming model used on today's supercomputers. Therefore, MPI library developers are looking for the best possible performance (shortest run-time) of individual MPI functions across many different supercomputer architectures. Several MPI benchmark suites have been developed to assess the performance of MPI implementations. Unfortunately, the outcome of these benchmarks is often neither reproducible nor statistically sound. To overcome these issues, we show which experimental factors have an impact on the run-time of blocking collective MPI operations and how to measure their effect. Finally, we present a new experimental method that allows us to obtain reproducible and statistically sound measurements of MPI functions.

Index Terms—MPI, benchmarking, reproducibility, statistical analysis

1 Introduction

Since the Message Passing Interface (MPI) was standardized in the 1990s, it has been the prevalent programming model on the majority of supercomputers. As MPI is an essential building block of high-performance applications, performance problems in the MPI library have direct consequences on the overall run-time of applications.

Library developers and algorithm designers have one question in common: which algorithm works better (is faster) for a given communication problem? For example, which implementation of broadcast is faster on $p = 128$ processors using a payload of $m = 64\,\text{Bytes}$? As today's parallel systems can hardly be modeled analytically, empirical evaluations using run-time tests of MPI functions are required to compare different MPI implementations. It is therefore important to measure the run-time of MPI functions correctly.

Library developers rely on benchmark suites to test their implementations. The problem is that the results of these benchmarks may vary significantly, where Table 1 shows one example. The table compares the minimum and maximum of the average run-times reported for 30 invocations (`mpiruns`) of the Intel MPI Benchmarks for an `MPI_Bcast` on 16 nodes. The third column lists the difference between the minimum and maximum run-time in percent. We can see that for payloads of up to $512\,\text{Bytes}$, the run-times have an error of roughly $10\,\%$. One solution might be to change the default parameters of the Intel MPI Benchmarks. For example, one could force the benchmark to perform more measurements. But the question then becomes: how many runs are sufficient to obtain reproducible results?

It is a common practice—when comparing MPI implementations as part of a scientific publication—to choose one of the available benchmarks and compare the results. Many MPI benchmarks report either the mean, the median, or the minimum run-time. The problem is that without using a dispersion metric and a rigorous statistical analysis, we can hardly determine whether an observation is repeatable or the result of chance.

Table 1
Minimum and maximum run-time of `MPI_Bcast` obtained from 30 different runs of the Intel MPI Benchmarks on 16×1 processes with NEC MPI 1.2.8 on *TUWien*.

Msg. size $[Bytes]$	min(avg) $[\mu s]$	max(avg) $[\mu s]$	diff $[\%]$
1	2.93	3.12	6.09
2	2.83	3.20	11.56
4	2.82	3.06	7.84
8	2.86	3.13	8.63
16	2.84	3.14	9.55
32	3.13	3.44	9.01
64	3.15	3.51	10.26
128	3.62	4.03	10.17
256	4.34	4.90	11.43
512	5.41	5.91	8.46
1024	6.88	7.05	2.41

We make the following contributions to the problem of *accurately benchmarking blocking collective MPI operations*:

1) We establish a list of (experimental) factors that, we show, do significantly influence the outcome of MPI performance measurements.
2) We propose a novel benchmarking method, including an experimental design and an appropriate data analysis strategy, that allows for a fair comparison of MPI libraries, but which is most importantly (1) statistically sound and (2) reproducible.

We start by summarizing other MPI benchmarking approaches in Section 2 and discuss their strengths and shortcomings. Section 3 introduces our general experimental framework, which has been used for all experiments conducted as part of this article. Section 4 takes a closer look at factors that may influence the experimental outcome (the run-time of an MPI function). Section 5 describes our method for comparing the performance of MPI libraries in a statistically

• *S. Hunold and A. Carpen-Amarie are with the Vienna University of Technology, Faculty of Informatics, Research Group for Parallel Computing, Favoritenstrasse 16/184-5, 1040 Vienna, Austria*
E-mail: {hunold,carpenamarie}@par.tuwien.ac.at

This work was supported by the Austrian Science Fund (FWF): P26124 and P25530

Table 2
Comparison of metrics for several MPI benchmark suites.

benchmark name	ref.	version	mean	min	max	measure of dispersion
Intel MPI Benchmarks	[1]	4.0.2	✓	✓	✓	✗
MPIBench	[2]	1.0beta	✓	✓		sub-sampled data
MPIBlib	[3]	1.2.0	✓			conf. interv. of the mean (default 95%)
`mpicroscope`	[4]	1.0	✓	✓	✓	✗
`mpptest`	[5]	1.5	min of means			✗
NBCBench	[6]	1.1	✓(+median)	✓	✓	✗
OSU Micro-Benchmarks	[7]	4.4.1	✓	✓	✓	✗
Phloem MPI Benchmarks	[8]	1.0.0	✓	✓	✓	✗
SKaMPI	[9]	5.0.4	✓			std. error

sound manner. We summarize related work in Section 6 with an emphasis on statistically sound experiments, before we conclude in Section 7.

2 A Brief History of MPI Benchmarking

We now give a history of MPI benchmarking. Ever since the first MPI standard was announced in 1995, several MPI benchmark suites have been proposed. Some of the best-known MPI benchmark suites are summarized in Table 2. The table includes information about the measures (e.g., min, max) that each benchmark uses to present run-times and which measure of dispersion is provided. It is complemented with the run-time measurement approaches implemented by each benchmark, which are separately summarized in the four pseudocode listings in Table 3. Furthermore, Table 4 details the methods selected by each of the investigated benchmarks for computing and presenting the measured run-times. The data in Table 2 was gathered to the best of our knowledge, since some benchmarks, like the Special Karlsruher MPI-Benchmark (SKaMPI), have been released in many incarnations and some other ones, like the MPIBench, are currently not available for download[1]. We therefore also rely on the respective articles describing the benchmarks.

`mpptest` was one of the first MPI benchmarks [5] and was a part of the MPICH distribution. Gropp and Lusk carefully designed `mpptest` to allow reproducible measurements for realistic usage scenarios. They pointed out common pitfalls when conducting MPI performance measurements, such as ignoring cache effects. In particular, to ensure cold caches when sending a message, `mpptest` uses a send and a receive buffer which are twice as big as the cache level that should be "cold". Then, the starting address of a message to be sent is always advanced in this larger buffer, trying to ensure that the data accessed are not cached. If a starting address does not leave enough space for the message to be sent, it is reset to the beginning of the buffer. At the time of designing `mpptest`, most of the hardware clocks were coarse-grained and therefore did not allow a precise measurement of one call to a specific MPI function (as this would have often resulted in obtaining a 0). To overcome this problem and to improve the reproducibility of results, `mpptest` measures the time t of *nrep* consecutive calls to an MPI function and computes the mean $\bar{t}_i = t/nrep$ of these *nrep* observations. This measurement is repeated k times and the minimum over these k samples is reported, i.e., $\min_{1 \leq i \leq k} \bar{t}_i$.

The SKaMPI benchmark is a highly configurable MPI benchmark suite [9] and features a domain-specific language for describing individual MPI benchmark tests. SKaMPI also allows to record MPI timings by using a window-based process synchronization approach, in addition to the commonly used `MPI_Barrier` (cf. measurement schemes (MS1) and (MS4) in Table 3). SKaMPI reports the arithmetic mean and the standard error of the run-times of MPI functions. It uses an iterative measuring process for each test case, where a test is repeated until the current relative standard error is smaller than a predefined maximum.

MPIBlib by Lastovetsky *et al.* [3] works similarly to SKaMPI, as it computes a confidence interval of the mean based on the current sample. It stops the measurements when the sample mean is within a predefined range (e.g., a 5 % difference) from the end of a 95% confidence interval. MPIBlib implements multiple methods for computing the sample mean, as shown in schemes (PS2) and (PS4) in Table 4. It also provides an additional scheme that measures the run-time on the root process only, but which we omitted for reasons of clarity.

`mpicroscope` [4] and OSU Micro-Benchmarks [7] perform repeated measurements of one specific MPI function for a predefined number of times. They report the minimum, the maximum, and the mean run-time of a sample. `mpicroscope` attempts to reduce the number of measurements using a linear (or optionally exponential) decay of repetitions, i.e., if no new minimum execution time in a sample of *nrep* consecutive MPI function calls was found, the remaining number of repetitions is decreased.

The Intel MPI Benchmarks [1] use a measurement method similar to `mpptest`, i.e., the time is taken before and after executing *nrep* consecutive calls to an MPI function. Then, the benchmark computes the mean of the run-times over these *nrep* consecutive calls for each MPI rank. The final report includes the minimum, maximum, and average of these means across all ranks.

The Phloem MPI Benchmarks [8] for MPI collectives measure the total time to execute *nrep* consecutive MPI function calls and compute the mean run-time for each rank. In addition, the Phloem MPI Benchmarks can be configured to interleave the evaluated MPI function calls with calls to `MPI_Barrier` in each iteration. Minimum, maximum, and average run-times across MPI ranks are provided upon benchmark completion.

Grove and Coddington developed MPIBench [2], which,

1. We obtained the source code of MPIBench 1.0beta through private communication.

Table 3
Measurement schemes (MS) for blocking MPI collectives found in different MPI benchmarks. In scheme (MS4), depending on the implementation, `Get_Time` returns the local time (measured with `MPI_Wtime` or `RDTSC`) or a logical global time.

(MS1) SKaMPI, NBCBench, MPIBlib, MPIBench, `mpicroscope`, OSU Micro-Benchmarks	(MS2) Intel MPI Benchmarks, `mpptest`	(MS3) Phloem MPI Benchmarks	(MS4) SKaMPI, NBCBench
1: **for** *obs* in 1 to *nrep* **do** 2: `MPI_Barrier` 3: $l^{s\_time}[obs]$ = `MPI_Wtime` 4: execute MPI function 5: $l^{e\_time}[obs]$ = `MPI_Wtime` 6: t += $l^{e\_time}[obs] - l^{s\_time}[obs]$ // *OSU Micro-Benchmarks*	1: `MPI_Barrier` // *or omitted* 2: s_time = `MPI_Wtime` 3: **for** *obs* in 1 to *nrep* **do** 4: execute MPI function 5: e_time = `MPI_Wtime` 6: $t = (e\_time - s\_time)/nrep$	1: `MPI_Barrier` 2: s_time = `MPI_Wtime` 3: **for** *obs* in 1 to *nrep* **do** 4: execute MPI function 5: `MPI_Barrier` // *or omitted* 6: e_time = `MPI_Wtime` 7: $t = (e\_time - s\_time)/nrep$	1: SYNC CLOCKS() 2: DECIDE on *start_time* and *win_size* 3: **for** *obs* in 1 to *nrep* **do** 4: WAIT_UNTIL($start\_time + obs \cdot win\_size$) 5: $l^{s\_time}[obs]$ = GET_TIME() 6: execute MPI function 7: $l^{e\_time}[obs]$ = GET_TIME()

Table 4
Commonly used data processing schemes (PS) for benchmarking blocking collective MPI operations.

(PS1) SKaMPI	(PS2) MPIBlib, `mpicroscope`	(PS3) OSU Micro-Benchmarks	(PS4) MPIBlib
1: **for** *obs* in 1 to *nrep* **do** 2: $l^{t\_local}[obs] = l^{e\_time}[obs] - l^{s\_time}[obs]$ 3: REDUCE *nrep* local run-times from each process on *root*: $l^{max}[obs] = \text{MAX}_p(l_p^{t\_local}[obs])$ 4: SORT(l^{max}) 5: $l^t = l^{max}[\, nrep/4 : (nrep - nrep/4)]$ 6: print $\text{MEAN}_{nrep/2}(l^t)$, $\text{STDEV}_{nrep/2}(l^t)$	1: **for** *obs* in 1 to *nrep* **do** 2: $l^{t\_local}[obs] = l^{e\_time}[obs] - l^{s\_time}[obs]$ 3: REDUCE *nrep* local run-times from each process on *root*: $l^{max}[obs] = \text{MAX}_p(l_p^{t\_local}[obs])$ 4: print $\text{MEAN}_{nrep}(l^{max})$, $\text{CI}_{nrep}(l^{max})$ // *MPIBlib* print $\text{MEAN}_{nrep}(l^{max})$, $\text{MIN}_{nrep}(l^{max})$, $\text{MAX}_{nrep}(l^{max})$ // `mpicroscope`	1: $local\_time = t/nrep$ 2: REDUCE *local_time* from each process to *root*: $min\_lat = \text{MIN}_p(local\_time)$ $max\_lat = \text{MAX}_p(local\_time)$ $mean\_lat = \text{SUM}_p(local\_time) / nrep$ 3: print *mean_lat*, *min_lat*, *max_lat*	1: $l^{global} = \text{NORMALIZE\_TIMES}(l^{e\_time})$ 2: **for** *obs* in 1 to *nrep* **do** 3: REDUCE $l^{global}[obs]$ from each process to *root*: $l^{max\_gl}[obs] = \text{MAX}_p(l_p^{global}[obs])$ 4: $l^t[obs] = l^{max\_gl}[obs] - l^{s\_time}[obs]$ 5: print $\text{MEAN}_{nrep}(l^t)$, $\text{CI}_{nrep}(l^t)$

(PS5) MPIBench	(PS6) NBCBench	(PS7) Intel MPI Benchmarks, Phloem MPI Benchmarks	(PS8) `mpptest`
1: $l^{s\_time} = \text{NORMALIZE\_TIMES}(l^{s\_time})$ 2: $l^{e\_time} = \text{NORMALIZE\_TIMES}(l^{e\_time})$ 3: GATHER l^{e_time}, l^{s_time} from each process on *root* into $l_{final}^{e\_time}$, $l_{final}^{s\_time}$ 4: **for** *rank* in 1 to *p* **do** 5: **for** *obs* in 1 to *nrep* **do** 6: $i = (rank - 1) \cdot nrep + obs$ 7: $l_{final}^t[i] = l_{final}^{e\_time}[i] - l_{final}^{s\_time}[i]$ 8: $l^t = \text{REMOVE\_OUTLIERS}_{p \cdot nrep}(l_{final}^t)$ 9: print $\text{MIN}(l^t)$, $\text{MAX}(l^t)$, $\text{MEAN}(l^t)$	1: **for** *obs* in 1 to *nrep* **do** 2: $l^{t\_local}[obs] = l^{e\_time}[obs] - l^{s\_time}[obs]$ 3: $t = \text{OP}(l^{t\_local})$ // $OP \in \{min, max, mean, median\}$ 4: GATHER t from each process on *root* into l^t 5: $t^{max} = \text{MAX}_p(l^t)$ 6: print t^{max}	1: GATHER average times t on *root* process into l^t 2: print $\text{MIN}_p(l^t)$, $\text{MAX}_p(l^t)$, $\text{MEAN}_p(l^t)$	1: BROADCAST t from the *root* process 2: collect $\text{MIN}_{reps}(t)$ over several repetitions of the measurement scheme

in addition to mean and minimum run-times, also plots a sub-sample of the raw data to show the dispersion of measurements. They discuss the problem of outlier detection and removal. In their work, the run-times that are bigger than some threshold time t_{thresh} are treated as outliers. To compute t_{thresh}, they determine the 99th percentile of the sample, denoted as t_{99}, and then define $t_{thresh} = t_{99} \cdot a$ for some constant $a \geq 1$ (default $a = 2$). Grove also shows the distribution of run-times obtained when measuring `MPI_Isend` with different message sizes [10, p. 127]. He highlights the fact that the execution time of MPI functions is not normally distributed.

NBCBench was initially introduced to assess the run-time of non-blocking collective implementations in comparison to their blocking alternatives [6]. Later, Hoefler *et al.* explained how blocking and non-blocking collective MPI operations could be measured scalably and accurately [11]. The authors show that calling MPI functions consecutively can lead to pipelining effects, which could distort the results. To address these problems, they implement a window-based synchronization scheme, requiring $\mathcal{O}(\log p)$ rounds to complete, compared to the $\mathcal{O}(p)$ rounds needed by SKaMPI, where p denotes the number of processes.

3 Experimental Setup

As our paper heavily relies on the empirical analysis of experimental data, we first introduce our scheme for measuring the run-time of blocking MPI collectives, the data processing methods applied, and the parallel machines used for conducting our experiments.

We use the following notation—borrowed from Kshemkalyani and Singhal [12]—in the remainder of the article. The *clock offset* is the difference between the time reported by two clocks. The *skew of the clock* is the difference in the frequencies of two clocks and the *clock drift* is the difference between two clocks over a period of time.

3.1 Timing Procedure and Process Synchronization

In all experiments presented in this article, we measured the time for completing a single MPI function using the method

Algorithm 1 MPI timing procedure.

1: **procedure** TIME_MPI_FUNCTION(*func*, *msize*, *nrep*)
// func - MPI function
// msize - message size
// nrep - nb. of observations
2: initialize time array l^t with *nrep* elements
3: **for** *obs* in 1 to *nrep* **do**
4: SYNC_PROCESSES() *// either* MPI_BARRIER *or window-based synch.*
5: $l^{s\_time\_local}[obs]$ = GET_TIME()
6: execute *func* (*msize*)
7: $l^{e\_time\_local}[obs]$ = GET_TIME()
8: **if** sync method == `MPI_Barrier` **then**
9: **for** *obs* in 1 to *nrep* **do**
10: $l^{t\_local}[obs] = l^{e\_time\_local}[obs] - l^{s\_time\_local}[obs]$
11: MPI_REDUCE(l^{t_local}, l^t, *nrep*, MPI_DOUBLE, MPI_MAX, *root*)
12: **else**
13: normalize $l^{s\_time\_local}$, $l^{e\_time\_local}$ to the global reference clock
14: MPI_REDUCE($l^{s\_time\_local}$, l^{s_time}, *nrep*, MPI_DOUBLE, MPI_MIN, *root*)
15: MPI_REDUCE($l^{e\_time\_local}$, l^{e_time}, *nrep*, MPI_DOUBLE, MPI_MAX, *root*)
16: **for** *obs* in 1 to *nrep* **do**
17: $l^t[obs] = l^{e\_time}[obs] - l^{s\_time}[obs]$
18: **if** *my_rank* == *root* **then**
19: **for** *obs* in 1 to *nrep* **do**
20: print $l^t[obs]$

shown in Algorithm 1. Before the start of a benchmark run, the experimenter can select the number of observations *nrep* (sample size) to be recorded for an individual test, where a test consists of an MPI function, a message size, and a number of processes. Before starting to measure the run-time of an MPI function, all processes need to be synchronized. We examine two synchronization approaches: (1) `MPI_Barrier` and (2) a window-based scheme.

The window-based synchronization scheme works as follows: (a) The distributed clocks of all participating MPI processes are synchronized relative to a reference clock. (b) The master process selects a start time, which is a future point in time, and broadcasts this start time to all participating processes. (c) Since each process knows the logical global time, it is now able to wait for this common start time before executing the respective MPI function synchronously with the others. (d) When one MPI function call has been completed, all processes will wait for another future point in time before starting the next measurement. The time period between these distinct points in time is called a "window".

3.1.1 Window-based Process Synchronization

To apply a window-based process synchronization scheme, a clock synchronization algorithm needs to be selected.

As mentioned in Section 2, the benchmarks SKaMPI and NBCBench provide window-based run-time measurements of MPI functions. However, the clock synchronization algorithms implemented in both benchmarks account for the clock offset only. The problem is that the clocks of compute nodes are quickly drifting over time. As a consequence, the logical global clocks implemented in SKaMPI and NBCBench lead to inaccurate results for a large number of repetitions [13]. For that reason, we are interested in clock synchronization algorithms that consider the clock drift, since we want to conduct several hundreds or thousands of measurements of individual MPI functions.

One such algorithm is the clock synchronization method proposed by Jones and Koenig [14], which assumes that the clock drift is linear in time. Each process learns a linear model of the clock drift by exchanging ping-pong messages with a single reference process. After computing the linear model of the logical global clock, each process can determine the logical global time by adjusting its local time relative to the time of the reference process.

As the algorithm of Jones and Koenig requires a relatively long synchronization time, we combined the advantages of the synchronization algorithms of Jones & Koenig and Hoefler *et al.* Our novel algorithm synchronizes distributed clocks in a hierarchical way, but also takes the clock drift into account [13]. Similarly to the algorithm of Jones and Koenig, our algorithm learns a linear model of the clock, defined by a slope (the clock drift) and an intercept. We proposed two variants of the new algorithm [13]: The *first variant* (HCA) computes the intercepts in $\mathcal{O}(p)$ rounds after completing the hierarchical computation of the clock models, which only requires $\mathcal{O}(\log p)$ rounds. The advantage of HCA is that the intercept is measured for each clock model separately, which leads to a more accurate global clock. The *second variant* (HCA2) computes the intercepts during the hierarchical computation of the clock model in $\mathcal{O}(\log p)$ rounds. The advantage of this method compared to the first approach is its better scalability (run-time). The downside is that relying on a combined intercept for the linear model increases the error of the logical global clock.

3.1.2 Measurement Scheme Applied

In this paper, we use the measurement schemes (MS1) and (MS4) of Table 3 for experiments with `MPI_Barrier`-based or window-based process synchronization, respectively. Both schemes are implemented in the ReproMPI benchmark suite[2].

When applying the window-based scheme (MS4), we employ either the algorithm of Jones and Koenig or the algorithm of Hunold and Carpen-Amarie (variant HCA). The accuracy of both algorithms is similar for the number of processes used in our experiments, but HCA reduces the time to conduct the experiments. For each figure showing performance results, we state which synchronization option has been used (`MPI_Barrier`, JK, or HCA).

Depending on the type of synchronization, we use different ways to compute the time to complete a collective MPI operation (see Algorithm 1, lines 8–17).

3.1.2.1 Run-times based on local times: When we apply the measurement scheme (MS1), each MPI process holds an array containing *nrep* (process-)local time measurements (run-times to complete a given MPI function). We apply a reduction operation (max) on that array and collect the results on the root process. Thus, the run-time of an MPI function *func* using p processes in iteration i, $1 \leq i \leq nrep$, is given as $l^t[i] = \max_{1 \leq r \leq p}\{l^{t\_local}[i]\}$. In other words, the run-time of an MPI function is defined as the maximum (process-)local run-time over all processes. We apply this run-time computation procedure when processes are synchronized using `MPI_Barrier`.

3.1.2.2 Run-times based on logical global times: When globally-synchronized clocks are available (MS4), we define the time to complete an MPI operation as the difference between the maximum finishing time and the minimum starting time among all processes. All *nrep* starting and finishing global timestamps from all processes are gathered

2. http://hunoldscience.net/reprompi

Table 5
Overview of parallel machines used in the experiments.

name	nodes	interconnect	MPI libraries	compilers
TUWien	36 Dual Opteron 6134 @ 2.3 GHz	IB QDR MT4036	NEC MPI/LX 1.2.{8,11} MVAPICH 2.1a	gcc 4.4.7
VSC-3	2000 Dual Xeon E5-2650V2 @ 2.6 GHz	IB QDR-80	MVAPICH 2.2b	gcc 4.4.7
Edel (G5k)	72 Dual Xeon E5520 @ 2.27 GHz	IB QDR MT26428	MVAPICH 1.9	gcc 4.7.2
JUQUEEN	IBM BlueGene/Q, 28.672 nodes @ 1.6 GHz	5D Torus	MPICH2-based	IBM XL C/C++ V12.1

as vectors on the *root* node. Then, the *root* node computes the time of an MPI function *func* using p processes in iteration i like this $l^t[i] = \max_{1 \leq r \leq p}\{l^{e\_time}[i]\} - \min_{1 \leq r \leq p}\{l^{s\_time}[i]\}$. We use this method to compute the completion time in all our experiments where a window-based process synchronization method is employed.

3.2 On the Resolution of Clocks

Hoefler *et al.* discussed the problem that the resolution of `MPI_Wtime` is typically not high enough for measuring short time intervals [15]. They therefore use the CPU's clock register to count the number of processor cycles since reset. More specifically, the authors implement a time-measurement mechanism based on the atomic `RDTSC` instruction, which provides access to the TSC register and which is supported by the x86 and x86-64 instruction set architectures (ISA). However, several problems can arise when using this mechanism. First, Hoefler *et al.* point out that dynamic frequency changes, which are automatically enabled in modern processors, can modify the CPU clock rate and thus compromise the time measurements. Second, in multi-processor systems, CPU clocks are not necessarily synchronized, requiring the processes to be pinned to their allocated core to guarantee valid cycle counter values.

Since we use Linux-based experimental platforms, we can read the TSC-related flags from `/proc/cpuinfo`. Except on the *JUQUEEN*, all our systems had the flags `constant_tsc` and `nonstop_tsc` set, indicating that updates of the TSC register are independent of the current core frequency. We also made sure that processes are pinned to cores throughout the measurements to avoid accidentally reading the TSC register of another core.

On the *JUQUEEN*, we use `MPI_Wtime`, as it provides a high clock resolution of 6.25×10^{-10} s. For all the other experiments presented in this article, we performed our measurements using the `RDTSCP` call, which guarantees instruction serialization, i.e., `RDTSCP` makes sure that all instructions have been executed when the timestamp counter is read. Unless otherwise specified, we fixed the frequency to the highest available value and pinned each process to a specific CPU in all our experiments involving `RDTSCP`-based time measurements.

3.3 Data Processing and Outlier Removal

Most of the benchmarks listed in Table 2 use some form of implicit outlier removal (e.g., by taking the minimum time recorded and dismissing the other data points). In addition, several benchmarks perform a number of warm-up rounds to fill caches or to set up communication links. After an initial warm-up phase has been completed, the measurements taken are used to compute the final statistics. One problem is that the operating system noise can lead to relatively large variations of the measured run-time at any moment within the benchmark execution. A second problem is that it is hard to estimate how many warm-up rounds are sufficient to reach a "steady state". To make our benchmark method robust against these two problems, we use Tukey's outlier filter to remove outliers after all measurements have been recorded [16, p. 126]. When applying this filter, we remove all measurements from the sample that are either smaller than $Q_1 - 1.5 \cdot IQR$ or larger than $Q_3 + 1.5 \cdot IQR$. The IQR denotes the interquartile range between quartiles Q_1 and Q_3.

3.4 Parallel Machines

The parallel machines used for conducting the experiments are summarized in Table 5. On the *TUWien* system, we have dedicated access to the entire cluster. The *Edel (G5k)* system belongs to Grid'5000[3], which features the OAR job scheduler that allows us to gain exclusive access to a set of nodes connected to the same InfiniBand (IB) switch. On *VSC-3* and *JUQUEEN*, the processor allocations include dedicated nodes only, but a dedicated access to the switches is not guaranteed.

4 Influencing Factors of MPI Benchmarks

After examining the MPI benchmarking process, we now turn to characterizing and analyzing the performance data. A good understanding of the performance data is essential for selecting and applying the right statistical test for comparing MPI alternatives. In addition, for a rigorous statistical analysis, we need a deeper insight into our system and the factors that influence the run-times to be measured. Le Boudec points out that "knowing all factors is a tedious, but necessary task. In particular, all factors should be incorporated, whether they interest you or not" [17].

Hence, we will first examine the shape of sampling distributions of run-time measurements. Then, we will analyze potential experimental factors in the remainder of this section. However, we decided to exclude "obvious" factors of MPI performance experiments, such as the communication network, the number of processes, and the message size.

Due to space limitations, we show only a subset of the measurements conducted throughout our study. More information on individual experiments, pseudocodes of experiments, or performance plots for other machines can be found in our arXiv report [18].

3. http://www.grid5000.fr

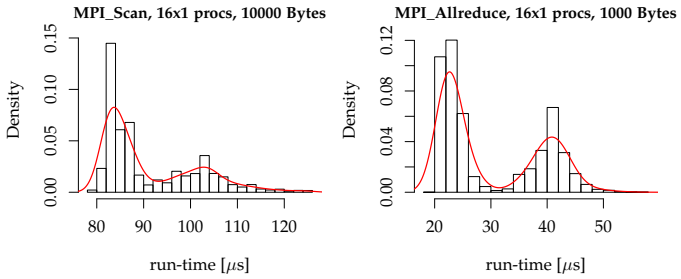


Figure 1. Histogram of the time needed to complete a call to `MPI_Scan` with 10 000 Bytes (left) and to `MPI_Allreduce` with 1000 Bytes (right) on *TUWien* (`MPI_Barrier` synchronization, NEC MPI 1.2.8).

4.1 Sampling Distributions of MPI Timings

To apply a statistical hypothesis test, we need to make sure that all its assumptions are met. Many tests assume that the data follow a specific probability distribution, e.g., the dataset is normally distributed. We now examine the experimentally obtained distributions of MPI function run-times.

We first ran a large number of MPI experiments to investigate various sampling distribution of MPI timings. The experiments were conducted for several MPI functions such as `MPI_Bcast`, `MPI_Allreduce`, `MPI_Alltoall`, or `MPI_Scan`. Figure 1 shows the distribution of run-times for 10 000 calls to `MPI_Scan` with a payload of 10 000 Bytes and to `MPI_Allreduce` with a payload of 1000 Bytes, both for 16×1 processes. We used a kernel density estimator (`density` in R) to obtain a visual representation of the sampling distribution. The figure indicates that the sampling distributions are clearly not normal, and interestingly, in both distributions we can see two distinct peaks. The peak on the right is much smaller, but it appears in many other histograms for small execution times (less than 200 µs). Similar distributions were obtained for other MPI collectives in experiments on different parallel machines [18].

Since the measured run-times do not follow a normal distribution, we must be careful when computing statistics such as the confidence interval for the mean. The central limit theorem (CLT) states that sample means are normally distributed if the sample size is large enough. In practice, we most often do not know in advance how large the sample size should be such that the CLT holds. Many textbooks, like the books by Lilja [19] or Ross [20], state that a sample size of 30 is large enough to obtain a normally distributed mean. However, Chen *et al.* [21] report in a recent study that samples need to include at least 240 observations, such that the sample means follow a normal distribution.

We are interested in how many repetitions of a single measurement are needed within one call to `mpirun` such that the CLT holds for the computed sample mean. To answer this question, we analyzed distributions of sample means by randomly sampling from the set of 10 000 previously measured MPI run-times of `MPI_Allreduce` (cf. Figure 1). In particular, we drew 500 samples containing 10, 20, ..., 500 observations each, computed the mean of each sample, and built a histogram of the sample means for each sample size. Figure 2 shows two histograms and their corresponding Q-Q plots for a sample size of 30 and 500. The data provides evidence that a sample size of 30 is not large enough to

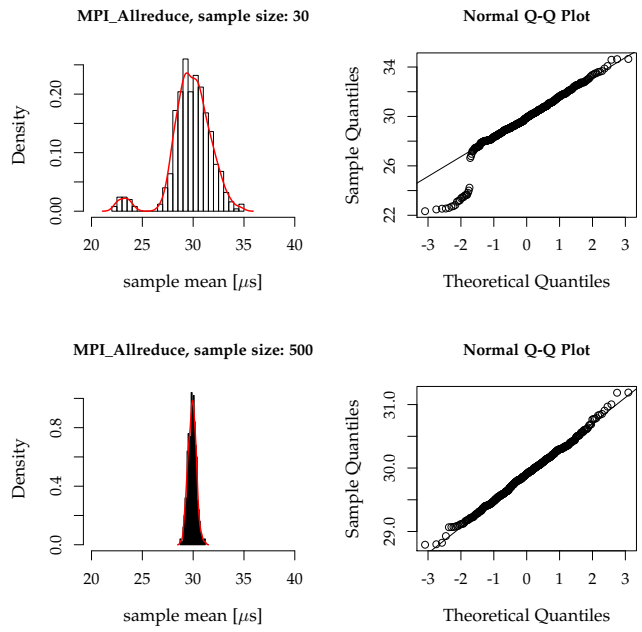


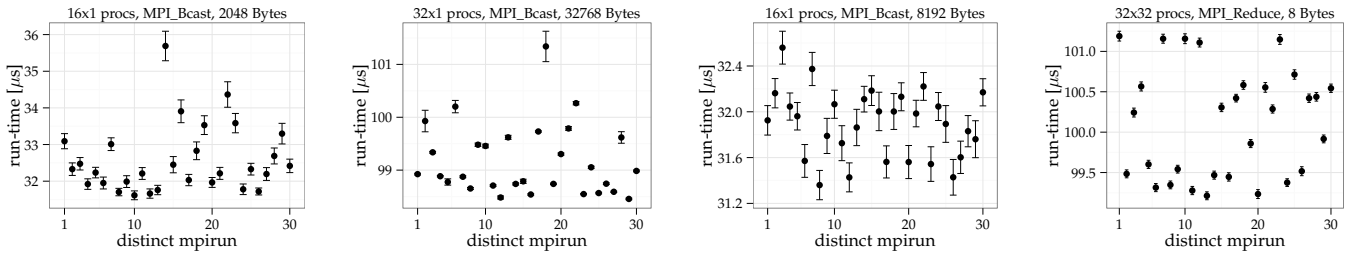
Figure 2. Distribution of sample means and Q-Q plots when sampling using different sample sizes from the probability distribution for `MPI_Allreduce` (cf. Figure 1).

obtain a normally distributed sample mean. In our particular case, 500 observations were required so that the distribution of sample means was normally shaped. We therefore advise scientists to carefully verify the sample distributions in order to compute meaningful confidence intervals of the sample mean when benchmarking MPI functions. A similar suggestion has been made recently by Hoefler and Belli [22].

4.2 Factor: The Influence of `mpirun`

When investigating the results of the sampling experiment presented in Section 4.1, we noticed that the sample means were slightly changing between calls to `mpirun`. To investigate the effect of `mpirun`, we conducted a series of experiments to determine whether distinct calls to `mpirun` produce different sample means (statistically significant). We ran the following experiment: We executed 30 distinct calls to `mpirun` and within each `mpirun` measured 1000 times the individual run-time of a given MPI function. All 30 calls to `mpirun` were executed as part of the same compute job of the queuing system (e.g., SLURM, PBS, etc.), which means that all `mpirun`s used the same node and processor allocation. We removed outliers as described in Section 3.3. The Figures 3(a)–3(c) present the experimental results for the three parallel machines *TUWien*, *VSC-3*, and *Edel (G5k)*. The graphs compare the arithmetic means and their 95 % confidence intervals for 30 distinct calls to `mpirun`, a given MPI function, and a message size. The data yielded by this experiment provide evidence that the means of the run-times obtained from distinct calls to `mpirun` are different. The differences between these means, however, are often not very large (3 %–5 %), yet they are statistically significant.

We performed a similar experiment on a BlueGene/Q to test whether `mpirun` also has an effect on specialized



(a) *TUWien*, NEC MPI 1.2.8 (b) *VSC-3*, MVAPICH 2.2b (c) *Edel (G5k)*, MVAPICH 1.9 (d) *JUQUEEN*, IBM MPI

Figure 3. Mean and 95 % confidence interval of the time to complete a call to `MPI_Bcast` or `MPI_Reduce` (with different payloads) for 30 distinct calls to `mpirun`, using `MPI_Barrier` for synchronization.

supercomputers. Again, the different calls to `mpirun` are executed as part of one compute job, and thus, the processor allocation and the pinning policy between different runs were identical. Figure 3(d) shows the means and confidence intervals obtained from 30 calls to `mpirun`, in which we repeated the measurement of `MPI_Reduce` 300 times. In this case, we also observe varying means between subsequent calls to `mpirun`, yet the relative difference between them was only 1 %–2 %.

Our finding that `mpirun` can significantly affect the experimental outcome is very important for designing MPI benchmarks. As a consequence, it is insufficient for an MPI benchmark to collect MPI run-time measurements only from a single call to `mpirun`. Instead, several calls to `mpirun` are required to account for the variance between different calls.

We still need to answer the question of how many calls to `mpirun` are required to obtain meaningful results when means or medians vary between calls? Answering this question without an explicit assumption about the underlying distribution is impossible. So far, we have tested on a handful of parallel machines with various MPI functions, number of processes, and message sizes. In our experiments, 10 calls to `mpirun` seemed to be the minimum number of repetitions needed to obtain a good estimate of the variance between `mpirun`s, while 30 repetitions were sufficient in all cases. Nevertheless, performing 10 to 30 `mpirun`s should be considered a rule of thumb, and experimenters need to carefully investigate the variance introduced by `mpirun` for their measurements.

4.3 Factor: Synchronization Method

After introducing and discussing several clock synchronization methods in Section 3.1, we now want to evaluate their effect on MPI benchmarking results.

We start by looking at the evolution of run-time measurements over a longer period of time in Figure 4. The graph compares the run-times of `MPI_Allreduce` measured using a window-based scheme (applying the clock synchronization algorithm of Jones and Koenig) with the run-times obtained using `MPI_Barrier` for process synchronization. The plot exposes two critical issues when measuring and analyzing MPI performance data. First, we see a significant difference between the mean and median run-times, which were computed for each bin of 10 000 runs. The difference is also present even though the outliers were removed (using

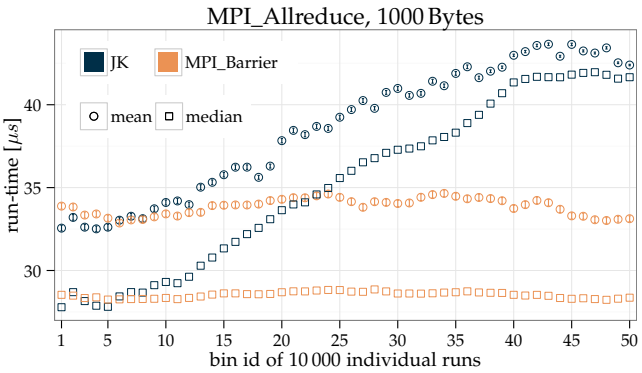


Figure 4. Mean and median of run-times of `MPI_Allreduce` when synchronizing either with the method of Jones and Koenig (JK) (window size: 1 ms) or an `MPI_Barrier` (1000 Bytes, 16 × 1 processes, 500 000 runs, bin size: 10 000, MVAPICH 2.1a, *TUWien*).

Tukey's method, cf. Section 3.3). Second, the use of a window-based synchronization method might allow the experimenter to obtain more accurate results, but even with a very precise clock synchronization method, such as the algorithm of Jones and Koenig, the run-times will gradually drift apart.

From the experiments shown above, we also see that the differences between the run-times measured with either a window-based or an `MPI_Barrier`-based scheme are relatively small. Now, the practitioner may ask the question: Is `MPI_Barrier` good enough to reasonably compare MPI measurements? In our opinion, the question cannot be answered generally as it depends on the actual goal of an experiment and the implementation of `MPI_Barrier`. If the experimenter seeks to obtain the most accurate timings for short-running MPI functions, the use of a window-based scheme should be considered. For a fair comparison of MPI implementations, relying on `MPI_Barrier` may be completely sufficient if the same `MPI_Barrier` algorithm is used by all of them (cf. Hunold and Carpen-Amarie [13]).

4.4 Factor: Pinning MPI Processes

It is well-known that the performance of MPI applications might be sensitive to the way processes are pinned to CPUs, as pinning can influence several performance-relevant properties, such as the cache reuse or the applicability of intra-node communication.

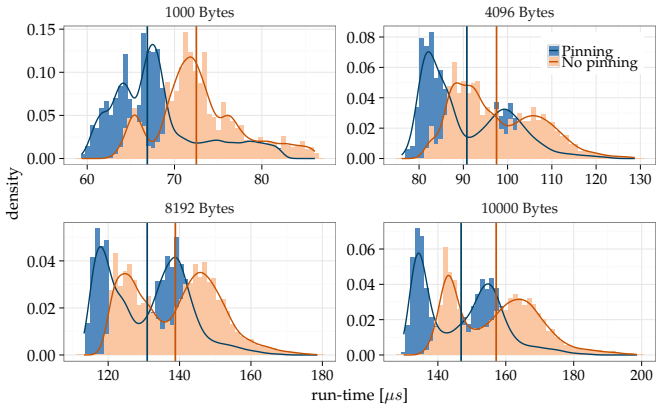


Figure 5. Pinning effect on `MPI_Allreduce`; vertical lines denote means (16×16 processes, 1000 measurements, 10 calls to `mpirun`, `MPI_Wtime`, HCA synch., window size: 1 ms, NEC MPI 1.2.11, *TUWien*).

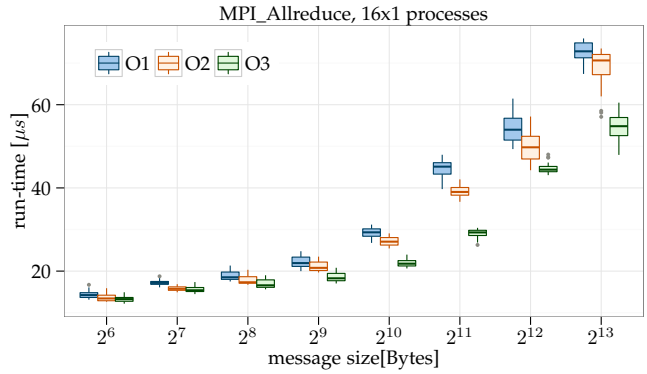


Figure 6. Compiler effect on the run-time of `MPI_Allreduce` (median run-time distribution of 30 calls to `mpirun`, 16×1 processes, 1000 measurements, HCA synch., window size: 1 ms, MVAPICH 2.1a, *TUWien*).

In the context of MPI benchmarking, CPU pinning is certainly required if we want to use the `RDTSC` instruction to measure the run-time, since results obtained using unpinned processes might be erroneous. Yet, the more general question is: Does pinning affect the execution time of MPI functions?

Figure 5 shows the results of an experiment in which we investigate whether the run-time of an MPI function changes if processes are pinned to CPUs or not (using `MPI_Wtime` for time measurements). The figure presents the histograms of run-times for `MPI_Allreduce` and various message sizes. Each histogram is generated by accumulating all run-time measurements from 10 different calls to `mpirun`. We can clearly see a significant difference in the shape of the histograms and the mean run-times, which are marked with a vertical line. Even though there could be cases where pinning has no effect on measurements, we have shown that pinning is an experimental factor to be considered in the context of MPI benchmarking.

4.5 Factor: Compiler and Compiler Flags

It seems self-evident to consider the compiler and the compiler flags as being significant experimental factors of MPI benchmarking applications. We still need to measure this effect to support our claim.

We conducted an experiment in which we measured the run-time of a call to `MPI_Allreduce` with the same version of MVAPICH. We recompiled the entire library (MVAPICH 2.1a) with gcc 4.4.7, but for each experimental run we changed the optimization flag to either `-O1`, `-O2`, or `-O3`. Figure 6 clearly shows that compiling the library using `-O3` outperforms the versions with other optimization flags. Even though it seemed obvious, our message is: If an MPI benchmarking experiment does not clearly state the compiler and the compilation flags used, the results will not be comparable or might not even be trustworthy.

4.6 Factor: DVFS

The majority of today's processors offer dynamic voltage and frequency scaling (DVFS) capabilities to reduce the energy consumption of the chip. Changing the core frequency is therefore an obvious factor for computationally-intensive workloads. In this work, we investigate whether the choice of the DVFS level may alter the run-times of MPI operations.

We conducted an experiment on *TUWien*, in which we compared the run-times of `MPI_Allreduce` for two different MPI implementations, MVAPICH 2.1a and NEC MPI 1.2.11, and for two different DVFS levels, 2.3 GHz and 0.8 GHz. Figure 7 presents the results of this experiment. The upper graph shows that MVAPICH outperforms NEC MPI for message sizes of up to 2^{10} Bytes when all processors are running at a fixed frequency of 2.3 GHz. In contrast, when we change the frequency to 0.8 GHz on all the processors, NEC MPI dominates MVAPICH for all message sizes. Additionally, we see that the individual run-times of `MPI_Allreduce` increase significantly when reducing the cores' frequencies.

The key observation is that the DVFS level needs to be carefully stated. Two MPI implementations may compare and behave differently depending on the chosen DVFS level.

4.7 Factor: Warm vs. Cold Cache

Gropp and Lusk [5] had already named the problem of "ignor[ing] cache effects" among the perils of performance measurements. They pointed out that the time to complete a send or receive operation depends on whether the send and receive buffers are in the caches or not. Therefore, `mpptest` uses larger arrays for sending and receiving messages, but the offset from where messages are sent or received is changed in a block-cyclic way at every iteration, to reduce the chance that data resides in cache.

The influence of caching was shown by Gropp and Lusk using `mpptest` for measuring the run-time of point-to-point communication. In the present work, we investigate how large the effect of caching is on blocking collective MPI operations. Instead of using buffer-cycling, we implemented another approach: we assume that the size of the last level of data cache, which is private to each core, is known. On current hardware this is often cache level 2. Let the size of this data cache be S^{LLC} Bytes. We allocate an auxiliary buffer buf_{aux} containing S^{LLC} Bytes. Now, we alter our MPI benchmark as follows: we overwrite the entire buffer buf_{aux} (e.g., using a for loop or `memset`) after each iteration, i.e.,

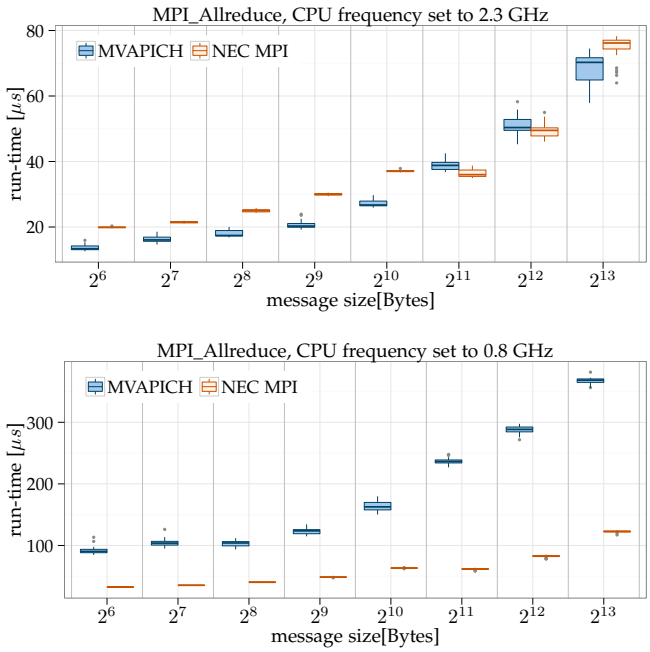


Figure 7. DVFS configuration effect on `MPI_Allreduce` run-times (median run-time distribution of 30 calls to `mpirun`, 16×1 processes, 1000 measurements, HCA synchronization, window size: 1 ms, MVAPICH 2.1a vs. NEC MPI 1.2.11, *TUWien*).

when one measurement of a collective MPI call has been completed. This way we attempt (since we do not know the hardware details) to ensure that our message buffer used for the MPI operation is not cached.

The effect of caching is shown in Figure 8, in which we can see that the reuse of message buffers between subsequent MPI calls, in this case `MPI_Allreduce`, has a significant impact on the run-time. As a result, MPI benchmarks must clearly state whether and how the caching of messages (buffers) is controlled.

4.8 Summarizing Experimental Factors

Our initial goal was to allow a fair and reproducible comparison of the performance of MPI implementations. A well-defined experimental design is one requirement to achieve that goal, and therefore, the analysis of experimental factors is of major importance. We have analyzed factors, such as compiler flags or cache control, and evaluated whether they have a significant effect on the experimental outcome. The influence of some factors on the performance measurements was not surprising, for example, we had expected that the DVFS level would affect the run-times.

However, the experiments led to two main results: The first lesson we learned was that the execution time of MPI benchmarks varies significantly between calls to `mpirun`. As a consequence, a reproducible and fair comparison of run-time measurements requires that performance data are recorded from different calls to `mpirun`. The second lesson, that we found quite surprising, was that determining which MPI implementation is better for a given case depends on the configuration of the experimental factors. For example, the run-time of `MPI_Bcast` might be shorter with library A

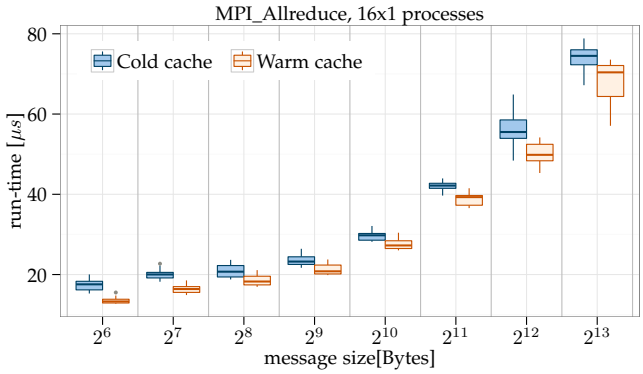


Figure 8. Cold cache vs. warm cache: Median run-times for `MPI_Allreduce` (16×1 processes, 30 calls to `mpirun`, 1000 measurements, HCA synch., window size: 1 ms, MVAPICH 2.1a, *TUWien*).

using DVFS level "low", but library B will provide a faster `MPI_Bcast` implementation in DVFS level "high".

Of course, our list of examined experimental factors is not exhaustive, and we are aware that other factors could also impact the experimental outcome. One such example is the operating system. Since many of such factors are often uncontrollable, we need to address them statistically.

In conclusion, we advise MPI experimenters to carefully list the settings of all experimental factors, besides the obvious factors such as number of processes, message size, and parallel machine. Benchmarking results should also include the setting of library-specific variables (often done through environment variables), which can influence the run-time of MPI libraries, e.g., by selecting a certain algorithm.

5 Statistically Rigorous and Reproducible MPI Benchmarking

After investigating the factors that may influence results of MPI benchmarks, we now propose a method to compare MPI implementations by using statistical hypothesis testing. Our goal is to establish an experimental methodology that aims to reproduce the test outcome between several experiments.

We motivate the need for a more robust evaluation method by showing the results in Figure 9. On the left-hand side, we see a comparison between the run-time of MVAPICH and NEC MPI when executing `MPI_Allreduce` with various message sizes. Each bar represents the mean run-time computed for 1000 individual measurements of a single call to `mpirun`. One might say that such a comparison is fair (due to the large number of repetitions) and we contend this is common practice when analyzing experimental results in the context of MPI benchmarking. However, when we look at the results shown on the right-hand side, for another `mpirun`, the outcome changes significantly. For example, the ratio of mean run-times for a message size of 2^1 Bytes has now changed. This observation matches the result of our factor analysis, in which we have discovered that the call to `mpirun` is an experimental factor (cf. Section 4.2). Therefore, we emphasize again that an MPI benchmark needs to collect data from multiple `mpirun` calls to be fair and reproducible.

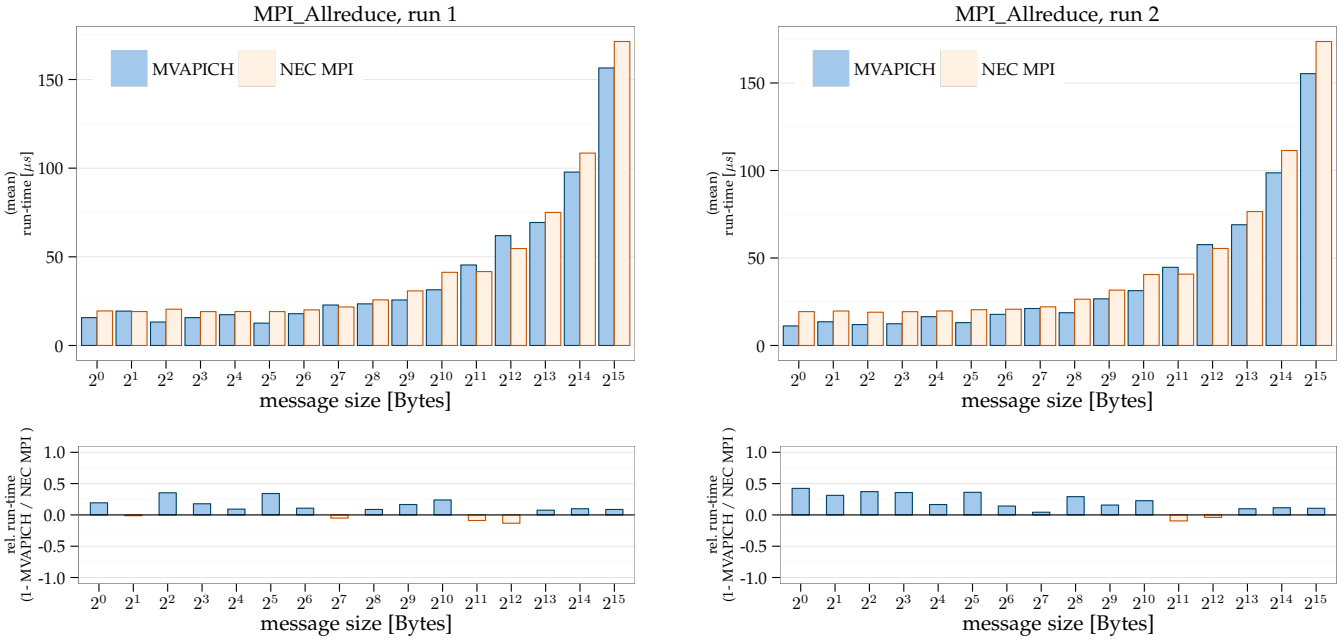


Figure 9. Comparison of mean run-times of `MPI_Allreduce` and different message sizes for two distinct calls to `mpirun` (16×1 processes, 1000 measurements per message size, HCA synch. with window sizes adapted to each message size, MVAPICH 2.1a vs. NEC MPI 1.2.11, *TUWien*).

5.1 Design of Reproducible Experiments

Our experimental design for measuring MPI performance data is shown in Algorithm 2. The procedure BENCHMARK generates the experimental layout and has five parameters, two of them being essential for the statistical analysis: (1) n denotes the number of distinct calls to `mpirun` for each message size, and (2) *nrep* specifies the number of measurements taken for each message size in each call to `mpirun`. In total, we measure the execution time of a specific MPI function for every message size $n \cdot nrep$ times. In the BENCHMARK procedure, we issue n calls to `mpirun`, where the number of processes p stays fixed. To respect the principles of experimental design (randomization, replication, blocking) as stated by Montgomery [23], we randomize the experiment by shuffling the order of experiments within a call to `mpirun`. The procedure SCAN_OVER_MPI_FUNCTIONS has three parameters: the list of message sizes (l^{msize}), the list of MPI functions to be tested (l^{func}), and the number of observations (*nrep*) to be recorded for each message size. The procedure creates a list (l^{exp}) containing the experiments covering all message sizes and MPI functions. The order of elements in this list is shuffled before the experiment starts. The procedure BENCHMARK of Algorithm 2 is executed for each MPI implementation. After the measurement results have been gathered, we apply the data-analysis procedure shown in Algorithm 3. Here, we group run-time measurements by the message size, the type of MPI function, the number of processes, and the `mpirun` id. We remove statistical outliers from each of these measurement groups. Last, we compute the median and the mean of each group of measurements and store them in a table. By applying this data-analysis method, we obtain a distribution of averages (medians or means) over n calls to `mpirun` for each message size, MPI function, and number of processes.

Algorithm 2 Design of an MPI experiment.

procedure BENCHMARK(p, n, l^{msize}, l^{func}, $nrep$)
  // p - nb. of processes
  // n - nb. of `mpiruns`
  // l^{msize} - list of message sizes
  // l^{func} - list of MPI functions
  // nrep - nb. of measurements per run
  for i in 1 to n **do**
    mpirun -np p SCAN_OVER_MPI_FUNCTIONS(l^{msize}, l^{func}, $nrep$)

procedure SCAN_OVER_MPI_FUNCTIONS(l^{msize}, l^{func}, $nrep$)
  $l^{exp} \leftarrow ()$
  for all $msize$ in l^{m} **do**
    for all $func$ in l^{func} **do**
      l^{exp}.add(TIME_MPI_FUNCTION($func$, $msize$, $nrep$))
  shuffle(l^{exp}) *// create random permutation of calls in place*
  for all exp in l^{exp} **do**
    call exp

Algorithm 3 Analysis of benchmark data.

procedure ANALYZE_RESULTS(l^{msize}, l^{p}, l^{func}, n)
  // l^{msize} - list of message sizes
  // l^{p} - list of processes
  // l^{func} - list of MPI functions
  // n - nb. of `mpiruns`
  for all $msize \in l^{msize}, p \in l^{p}, func \in l^{func}$ **do**
    for i in 1 to n **do**
      $l_i^t = \{ l^t[msize][p][func][i][j] \quad \text{for all } 1 \leq j \leq nrep \}$
      $l_i^t = \text{remove\_outliers}(l_i^t)$
      $v[msize][p][func][i] = (\text{median}(l_i^t), \text{mean}(l_i^t))$
  print v *// table with results*

5.2 Fair Performance Comparison Through Statistical Data Analysis

Now, the situation is as follows: we run our benchmark on two MPI implementations A and B and perform the data analysis according to Section 5.1, which gives us a distribution of averages for each measurement point. The question then becomes: How can we compare the measured results in a statistically sound way? We could reduce all

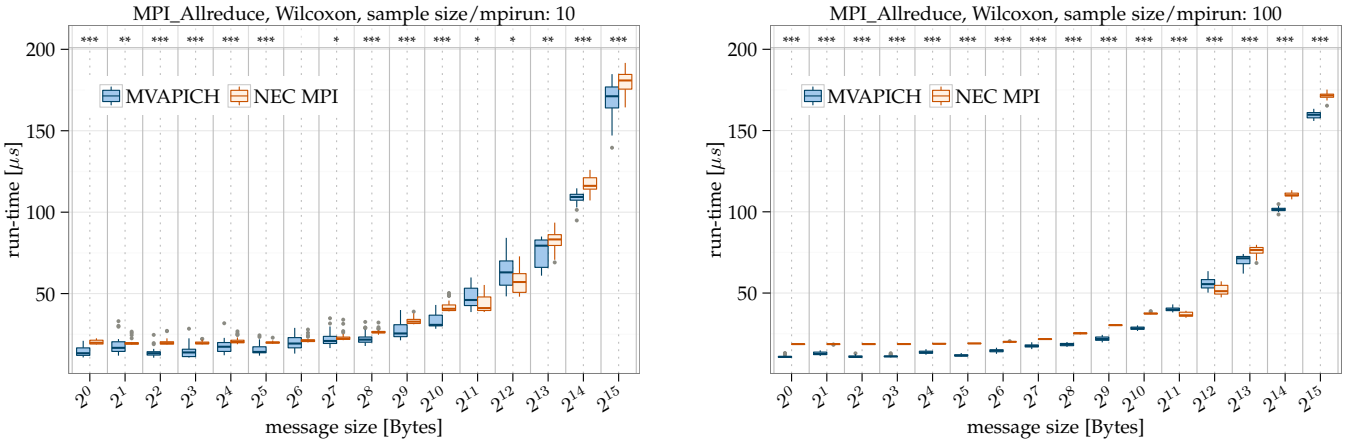


Figure 10. Comparison of the run-time distributions obtained when measuring `MPI_Allreduce` with a sample size of 10 (left) and 100 (right) per message size. The statistical significance was computed using the WILCOXON TEST (MVAPICH 2.1a vs. NEC MPI 1.2.11, 16×1 processes, 30 calls to `mpirun`, HCA synchronization with window sizes adapted to each message size, *TUWien*).

the values from the distribution to a single value using the minimum, the maximum, or the average, and then compare two MPI implementations based on this single value. However, our goal is to find evidence that a measured performance difference is reproducible.

Since we have two averages (the mean and the median) for each message size, we have several options for selecting a statistical test. If we use the computed median in a hypothesis test, we could employ the nonparametric Wilcoxon–Mann–Whitney test (Wilcoxon sum-of-ranks, in the remainder: WILCOXON TEST) for comparing alternatives [24]. The advantage of the WILCOXON TEST (besides being nonparametric) is that it makes no assumption on the underlying distribution; in particular, it "does not require the assumption of normality" [20]. We could also employ the WILCOXON TEST on the distribution of means, but in this case the T-TEST for two independent samples is also a promising candidate. The T-TEST assumes that the underlying population is normally distributed and that the variances of both populations are equal [25]. In our experiments, the distributions of sample means (over `mpiruns`) often followed a normal distribution, but unfortunately not all of them. We therefore chose to apply the WILCOXON TEST on median run-times exclusively in our experiments, and we used the R statistical environment for the analysis [26].

We now demonstrate how to apply the WILCOXON TEST to our data and discuss why the test helps us to provide a fair comparison of MPI implementations. Figure 10 shows our statistical comparison method applied to run-times measured for `MPI_Allreduce` with both NEC MPI and MVAPICH. Let us focus first on the graph on the left-hand side of this figure, where we compare the distributions of means recorded for different message sizes. Each distribution contains 30 elements, which are the median run-times measured in the 30 calls to `mpirun` with a sample size of 10 measurements. We apply the WILCOXON TEST on the two distributions of medians for each message size. The test does not only report whether the null hypothesis (both population means are equal) is rejected or not, but it also provides a *p-value*. To obtain a graphical representation of the p-value and therefore the statistical significance, we represent the p-value by a sequence of asterisks. One asterisk ($*$) represents a p-value of $p \leq 0.05$, two asterisks denote $p \leq 0.01$, and three asterisks denote $p \leq 0.001$. It also means that if asterisks are absent in a specific case, the null hypothesis could not be rejected, and thus, the statistical test does not provide sufficient evidence which implementation is better. We used a significance level of 0.05 (5 %) for all experiments.

When we look at the left graph of Figure 10, for which we applied the WILCOXON TEST, we see that using a hypothesis test can indeed help to separate cases, for which a decision can hardly be made only by looking at the distributions. For example, the differences between the distributions for 2^6 and 2^7 Bytes seem to be negligible. However, the WILCOXON TEST reveals that there is evidence that the sample medians are different in the case of 2^7 Bytes, but not in the case of 2^6 Bytes. The graph on the right-hand side of Figure 10 presents the results of another experiment, in which we used a sample size of 100 measurements per `mpirun`. It is not surprising that the variances of the distributions of the averages decrease, and thus, a larger sample size helps the hypothesis test to separate averages with a higher significance.

The graphs in Figure 10 compare run-time distributions of two MPI implementations and show the statistical test results of whether the population averages are equal. Yet, in a practical scenario one might rather ask a question like: Is MPI library X faster than library Y for MPI function F? To answer this question, we change the alternative hypothesis of the test to "less" (null hypothesis: $H_0 : \mu_A = \mu_B$, alternative hypothesis: $H_a : \mu_A < \mu_B$, where μ denotes the average). Figure 11 presents the results of the same experiment as shown on the right of Figure 10, but here we check whether the run-time of `MPI_Allreduce` is smaller with MVAPICH than with NEC MPI. We see that for the two cases, 2^{11} and 2^{12} Bytes, the null hypothesis could not be rejected, and thus, in these cases the run-time of `MPI_Allreduce` using MVAPICH is not smaller than when using NEC MPI. We note that this result does not immediately imply that NEC MPI is faster than MVAPICH in these cases. To verify this, we need to test using "greater" as the alternative hypothesis.

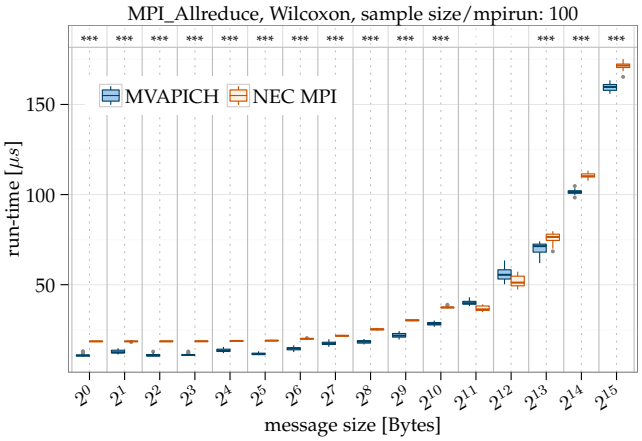


Figure 11. Comparison of the run-times of `MPI_Allreduce` while applying the WILCOXON TEST with "less" as an alternative hypothesis (experimental setup as in Figure 10).

5.3 Evaluating the Outcome Reproducibility

Until now, we have investigated the factors that potentially influence the benchmarking results of MPI functions and have shown how statistical hypothesis tests help us to fairly compare the performance of two MPI libraries. One of our initial goals was to develop a benchmarking method that leads to a reproducible experimental outcome (cf. example in Table 1).

To examine the reproducibility of our benchmarking method, we conducted the following experiment: We ran our benchmarking method (cf. Algorithm 2) for *ntrial*=30 times. Each of the *ntrial* runs gave us one distribution of run-times per message size, which contains n=30 values. Since we obtain a distribution of distributions, we collapse the inner distribution into a single value. To do so, we compute the mean of the n=30 values measured for one message size in each of the *ntrial* distributions. Then, we normalize the run-time values by computing the ratio of each mean to the minimum mean. We obtain a distribution of *ntrial*=30 normalized run-time values for our benchmarking method, presented in Figure 12(c). We can observe that the maximum relative difference between the 30 runs is very small (less than 5 % for 2^{14} Bytes).

As a comparison, we also conducted *ntrial*=30 runs of the Intel MPI Benchmarks 4.0.2 and SKaMPI 5. We used the standard configuration of the two benchmark suites (in particular, we used the default values of the number of repetitions for each message size). We compute the *normalized run-time* of each measurement for a specific message size as follows: $t^{norm}_{msize,i} = t_{msize,i}/t^{*}_{msize}$, for all $i, 1 \leq i \leq ntrial = 30$, where $t^{*}_{msize} = \min_{1 \leq i \leq ntrial} t_{msize,i}$. We can see in Figure 12(a) and Figure 12(b) that the normalized run-times of Intel MPI Benchmarks and SKaMPI exhibit a significantly larger variance for smaller message sizes than our benchmarking approach. The higher variance can be explained by the influence of system noise on experiments with small message sizes. In such cases, an MPI benchmark needs to record a sufficiently large number of repetitions across several calls to `mpirun`. The Intel MPI Benchmarks and SKaMPI simply lack such reproducibility policies.

We also show the results of a similar experiment in Figures 12(d)–12(f), where we assess the reproducibility of the measured run-times of `MPI_Allgather` using 1024 processes on *VSC-3*. Here, the Intel MPI Benchmarks also demonstrated a good reproducibility of results due to two reasons. First, the run-time of `MPI_Allgather` on 1024 processes is relatively large compared to the influencing system noise. Second, as the Intel MPI Benchmarks use the local times of each process to compute the final run-time, system noise is not always propagated through all processes, as it is the case when synchronizing between MPI function calls.

More generally, experimenters want to optimize two benchmarking criteria: accuracy and reproducibility. The run-time results of each MPI function measured with the Intel MPI Benchmarks are often much smaller than the ones obtained with SKaMPI's or our method due to the design of the timing procedure, where processes are not synchronized between subsequent calls (cf. Table 3). This measurement scheme may, and often does, cause pipelining effects [11], which can lead to substantially smaller mean run-times and thus to a decreased accuracy.

Overall, we can state that our benchmarking approach notably improves the reproducibility of the performance results compared to the Intel MPI Benchmarks and SKaMPI. The price for a better reproducibility, however, is a longer run-time of the overall benchmark, caused by the need to record a larger number of measurements per message size and to execute multiple `mpirun`s.

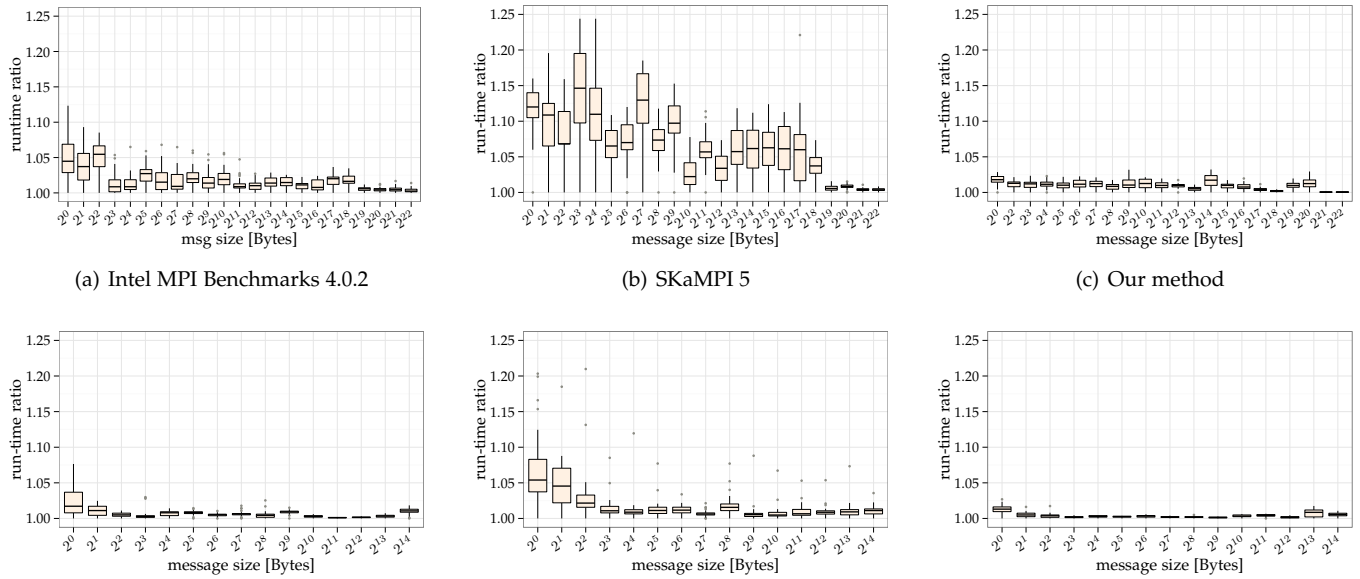
6 RELATED WORK

The statistically rigorous analysis of experimental data has been the focus of numerous studies over the last years, driven by the need for establishing a fair comparison of algorithms across different computing systems.

Vitek and Kalibera contend that "[i]mportant results in systems research should be repeatable, they should be reproduced, and their evaluation should be carried with adequate rigor". They show that a correct experimental design paired with the right statistical tests is the cornerstone for reproducible experimental results [27]. The authors stress the fact that knowing and understanding the controllable and uncontrollable factors of the experiment is crucial for obtaining sound experimental results.

The state of performance evaluation in Java benchmarking was investigated by Georges *et al.* [28]. They examined the performance of different garbage collectors for the Java Virtual Machine (JVM). The paper demonstrates that the answer to the question of which garbage collector is faster changes completely depending on the performance values investigated (e.g., mean, median, fastest, etc.). The authors show how to conduct a statistically rigorous analysis of JVM micro-benchmarks. In particular, they explain the need for considering confidence intervals of the mean and show that the Analysis of Variance (ANOVA) can be used to compare more than two alternatives in a sound manner.

Mytkowicz *et al.* dedicated an entire article to the problem of measurement bias in micro-benchmarks [29]. The authors examine the run-time measurements of several SPEC CPU2006 benchmarks, when each benchmark is either compiled with the optimization flag `-O2` or `-O3`. In theory,



(a) Intel MPI Benchmarks 4.0.2 (b) SKaMPI 5 (c) Our method

(d) Intel MPI Benchmarks 4.0.2 (e) SKaMPI 5 (f) Our method

Figure 12. Distributions of normalized run-times reported by the Intel MPI Benchmarks (left), SKaMPI (center), and our method (right). Top row: `MPI_Bcast`, 16×1 processes, MVAPICH 2.1a, *TUWien*; Bottom row: `MPI_Allgather`, 64×16 processes, MVAPICH 2.2b, *VSC-3*; HCA synchronization used in our method.

the programs compiled with `-O3` should run faster than the ones compiled with `-O2`. However, the authors discovered that the resulting performance not only depends on obvious factors such as the compilation flags or the input size, but also on less obvious factors, such as the link order of object files or the size of the UNIX environment. A possible solution is to apply a randomized experimental setup. Please refer to the books of Box *et al.* [30] and Montgomery [23] for more details on randomizing experiments.

Touati *et al.* developed a statistical protocol called Speedup-Test that can be used to determine whether the speedup obtained when modifying an experimental factor, such as the compilation flag (`-O3`), is significant [31]. The article presents two tests, one to compare the mean and one to compare the median execution times of two sets of observations. For a statistically sound analysis, they base both Speedup-Test protocols on well-known tests, such as the Student's t-test to compare means or the Kolmogorov-Smirnov test to check whether two samples have a common underlying distribution.

Chen *et al.* proposed the Hierarchical Performance Testing (HPT) framework to compare the performance of computer systems using a set of benchmarks [21]. The authors first contend that it is generally unknown how large the sample size needs to be, such that the central limit theorem holds. They show that for some distributions a sample size "[i]n the order of 160 to 240" is required to apply statistical tests that require normally distributed data [21]. Since such a high number of experiments seems infeasible for them, they propose a nonparametric framework to compare the performance improvement of computer systems. The HPT framework employs the nonparametric Wilcoxon Rank-Sum Test to compare the performance score of a single benchmark and the Wilcoxon Signed-Rank Test to compare the scores over all benchmarks.

7 Conclusions

We have revisited the problem of benchmarking MPI collectives. Our work was motivated by the need (1) to fairly compare MPI implementations using a sound statistical analysis and (2) to allow the reproducibility of results.

We have analyzed experimental factors of MPI experiments, for example, we have demonstrated that changing the DVFS level or the compiler flags can alter the outcome of the MPI benchmark. However, our most important finding is that a call to `mpirun` is a factor of the experiment, i.e., different calls to `mpirun` can produce significantly different means (or medians), even if all other factors and the input data stay unmodified.

After investigating the impact of various synchronization methods and experimental factors, we have proposed a novel MPI benchmarking method. We have shown how to apply hypothesis tests such as the Wilcoxon test to increase the fairness and the evidence level when comparing benchmarking data. Last, we have demonstrated that our benchmarking method also improves the reproducibility of results in such a way that the measured performance values exhibit a much smaller variance across different experiments compared to other MPI benchmark suites.

In general, benchmarking MPI collective communication operations is affected by the allocated processors and the current workload of the system. We therefore advise experimenters to perform all measurements temporally close to each other when comparing the run-time of MPI functions, to ensure that measurements are exposed to similar workload conditions. It is also important to conduct run-time experiments on the same set of nodes and using the same process pinning to allow for a fair and meaningful comparison of experimental results.

Acknowledgments

We thank Maciej Drozdowski (Poznan University of Technology), Peter Filzmoser (Vienna University of Technology), Friedrich Leisch and Bernhard Spangl (University of Natural Resources and Life Sciences, Vienna), Jesper Larsson Träff (Vienna University of Technology), Alf Gerisch (TU Darmstadt), and Thomas Geenen (SURFsara) for discussions and comments. We are also grateful to Bernd Mohr (Jülich Supercomputing Centre) for providing us access to a IBM BlueGene/Q. Experiments presented in this paper were carried out using the Grid'5000 testbed and the Vienna Scientific Cluster (VSC). Last, we thank the referees for their valuable comments.

References

[1] "Intel(R) MPI Benchmarks," http://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[2] D. A. Grove and P. D. Coddington, "Communication benchmarking and performance modelling of MPI programs on cluster computers," *The Journal of Supercomputing*, vol. 34, no. 2, pp. 201–217, 2005.

[3] A. L. Lastovetsky, V. Rychkov, and M. O'Flynn, "MPIBlib: Benchmarking MPI communications for parallel computing on homogeneous and heterogeneous clusters," in *EuroPVM/MPI*, 2008, pp. 227–238.

[4] J. L. Träff, "mpicroscope: Towards an MPI benchmark tool for performance guideline verification," in *EuroMPI*, 2012, pp. 100–109.

[5] W. Gropp and E. L. Lusk, "Reproducible measurements of MPI performance characteristics," in *EuroPVM/MPI*, 1999, pp. 11–18.

[6] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC)*. ACM, 2007, pp. 1–10.

[7] "OSU MPI benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks/.

[8] "Phloem MPI Benchmarks," https://asc.llnl.gov/sequoia/benchmarks/.

[9] R. Reussner, P. Sanders, and J. L. Träff, "SKaMPI: a comprehensive benchmark for public benchmarking of MPI," *Scientific Programming*, vol. 10, no. 1, pp. 55–65, 2002.

[10] D. Grove, "Performance modelling of message-passing parallel programs," Ph.D. dissertation, University of Adelaide, 2003. [Online]. Available: http://hdl.handle.net/2440/37915

[11] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately measuring collective operations at massive scale," in *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, PMEO'08 Workshop*, 04 2008.

[12] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[13] S. Hunold and A. Carpen-Amarie, "On the impact of synchronizing clocks and processes on benchmarking MPI collectives," in *EuroMPI*. ACM, 2015, pp. 8:1–8:10. [Online]. Available: http://doi.acm.org/10.1145/2802658.2802662

[14] T. Jones and G. A. Koenig, "Clock synchronization in high-end computing environments: a strategy for minimizing clock variance at runtime," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 6, pp. 881–897, Jul. 2012.

[15] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 25, no. 4, pp. 241–258, 2010.

[16] R. Hogg and E. Tanis, *Probability and Statistical Inference*. Prentice Hall, 2006.

[17] J.-Y. Le Boudec, *Performance Evaluation of Computer and Communication Systems*, ser. Computer and communication sciences. EFPL Press, 2010.

[18] S. Hunold and A. Carpen-Amarie, "MPI benchmarking revisited: Experimental design and reproducibility," *CoRR*, vol. abs/1505.07734, 2015. [Online]. Available: http://arxiv.org/abs/1505.07734

[19] D. Lilja, *Measuring Computer Performance*. Cambridge University Press, 2005.

[20] S. Ross, *Introductory Statistics*. Elsevier Science, 2010.

[21] T. Chen, Y. Chen, Q. Guo, O. Temam, Y. Wu, and W. Hu, "Statistical performance comparisons of computers," in *HPCA*. IEEE, 2012, pp. 399–410.

[22] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015*, 2015, pp. 73:1–73:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807644

[23] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[24] M. Hollander and D. Wolfe, *Nonparametric Statistical Methods*, ser. Wiley Series in Probability and Statistics. Wiley, 1999.

[25] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, 4th ed. Chapman & Hall/CRC, 2007.

[26] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org/

[27] J. Vitek and T. Kalibera, "R3: Repeatability, reproducibility and rigor," *SIGPLAN Not.*, vol. 47, no. 4a, pp. 30–36, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2442776.2442781

[28] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in *OOPSLA*, 2007, pp. 57–76.

[29] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*, 2009, pp. 265–276.

[30] G. Box, J. Hunter, and W. Hunter, *Statistics for Experimenters: Design, Innovation, and Discovery*. Wiley-Interscience, 2005.

[31] S. A. A. Touati, J. Worms, and S. Briais, "The Speedup-Test: A Statistical Methodology for Program Speedup Analysis and Computation," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 10, pp. 1410–1426, 2013.

Sascha Hunold is an assistant professor at the Vienna University of Technology. He holds a PhD in Computer Science from the University of Bayreuth and an M.Sc. in Computer Science from the University of Halle-Wittenberg, Germany.

Alexandra Carpen-Amarie is a postdoctoral researcher at Vienna University of Technology. She received her PhD degree in Computer Science from Ecole Normale Superieure de Cachan (France) and her engineering degree in Computer Science from University "Politehnica" of Bucharest (Romania). Her research interests include parallel and distributed systems, reproducible research, high-performance computing.