Reproducible MPI Micro-Benchmarking Isn't As Easy As You Think *

Sascha Hunold
hunold@par.tuwien.ac.at

Alexandra Carpen-Amarie
carpenamarie@par.tuwien.ac.at

Jesper Larsson Träff
traff@par.tuwien.ac.at

Vienna University of Technology
Faculty of Informatics, Institute of Information Systems
Research Group Parallel Computing
Favoritenstrasse 16/184-5, 1040 Vienna, Austria

ABSTRACT

The Message Passing Interface (MPI) is the prevalent programming model for supercomputers. Optimizing the performance of individual MPI functions is therefore of great interest for the HPC community. However, a fair comparison of different algorithms and implementations requires a statistically sound analysis. It is often overlooked that the time to complete an MPI communication function does not only depend on internal factors such as the algorithm but also on external factors such as the system noise. Most noise produced by the system is uncontrollable without changing the software stack, e.g., the memory allocation method used by the operating system. Possibly controllable factors have not yet been identified as such in this context. We investigate several possible factors—which have been discovered in other microbenchmarks—whether they have a significant effect on the execution time of MPI functions. We experimentally and statistically show that results obtained with other common benchmarking methods for MPI functions can be misleading when comparing alternatives. To overcome these issues, we explain how to carefully design MPI micro-benchmarking experiments and how to make a fair, statistically sound comparison of MPI implementations.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Experimentation, Measurement

*This work was supported by the Austrian Science Fund (FWF): P26124 and P25530.
The title is a homage to A. Gupta, S. Im, R. Krishnaswamy, B. Moseley, and K. Pruhs. Scheduling heterogeneous processors isn't as easy as you think. In *SODA*, 2012.

EuroMPI/ASIA '14, September 9-12 2014, Kyoto, Japan

http://dx.doi.org/10.1145/2642769.2642785.

Keywords

MPI, benchmarking, statistical analysis

1. INTRODUCTION

Since the Message Passing Interface (MPI) was standardized in the middle of the 1990s, it has become the prevalent programming model on the world's largest parallel computers. As MPI is a major building block of high-performance applications, performance problems in the MPI library have direct consequences on the overall runtime of the applications. Today, the two most commonly used MPI libraries are the open-source libraries MPICH and OpenMPI, and thus, everyone can implement new communication algorithms and test them on a target system.

A question for library developers is: which algorithm is better suited for a certain communication problem, e.g., which implementation of broadcast is faster on $p = 128$ cores and a message size of $m = 128$ Bytes? As today's parallel systems are too complex to be modeled mathematically, empirical evaluations using runtime tests of MPI functions are required to compare different MPI implementations. It is fundamental, but not self-evident, that a fair comparison can only be made if runtimes have been measured correctly.

We consider the problem of obtaining the necessary amount of experimental data to carry out the right statistical analysis for answering the question which MPI implementation performs better. First, we summarize related works on rigorous statistical analysis in benchmarking in Section 2. Then, in Section 3 we look at common practices of benchmarking MPI functions, where we point out the statistical methods applied in each benchmark. We also discuss possible pitfalls when performing statistical analysis on data sets for which the necessary assumptions (e.g., independently and identically distributed observations) have not been or can hardly be verified. We show in Section 4 the steps required to obtain a set of measurements that allows a sound statistical comparison of algorithmic and implementation alternatives. In Section 5, we propose a statistical method for the robust analysis of MPI function timings before we conclude in Section 6.

2. RELATED WORK

The statistically rigorous analysis of experimental data has received more attention over the last couple of 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 showed that a correct experimental design paired with the right statistical tests are the cornerstone for reproducible experimental results [21]. The authors stress the fact that it is crucial to know and understand the controllable and uncontrollable factors of the experiment.

Georges et al. closely looked at the state of performance evaluation in Java benchmarking [3]. They examined the performance of different garbage collectors for the Java Virtual Machine (JVM). Interestingly, depending on the statistical method applied to analyze the garbage collectors, the results would tell a different story. The statistical methods used to compare the alternatives were: presenting (1) the mean runtime, (2) the median, (3) the best (fastest), (4) the second best, or (5) the worst. The authors then show how a statistically rigorous analysis of JVM micro-benchmarks can be conducted. In particular, they show how to compute confidence intervals for the mean and how to apply the Analysis of Variance (ANOVA) in cases where many parameters are modified in the experiments.

Mytkowicz et al. dedicated an entire article to the problem of measurement bias in micro-benchmarks [14]. The authors focused on the runtime measurement of several SPEC CPU2006 benchmarks when each benchmark is compiled either by adding the compilation flag `-O2` or `-O3`. In theory, the program compiled with `-O3` should run faster than the one 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 conspicuous factors such as the link order of object files or the size of the UNIX environment. One possible solution proposed by the paper is applying a randomized experimental setup. Please refer to the books of Box et al. [1] and Montgomery [13] for more details on randomizing experiments.

Touati et al. developed a statistical protocol called Speedup-Test, which can be used to determine the overall speedup of a factor (e.g., the compilation flag `-O3`) over a set of benchmarks (e.g., SPEC OMP 2001) [19]. The article presents two tests, one to compare the mean and one to compare the median execution time of two sets of observations. For a statistically sound analysis, they base their protocol on well-known tests like the Student's t-test to compare means or the Kolmogorov-Smirnov test to verify 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 over a set of benchmarks [2]. The authors first contend that it is generally unknown how many observations a sample needs to include so that the central limit theorem holds. They show that for some distributions a sample size of 280 is required to apply statistical tests that require normally distributed data. Since such a high number of experiments seems infeasible for them, they propose a non-parametric framework to compare the performance improvement of computer systems. The HPT framework employs both the non-parametric 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.

Table 1: Overview of statistical methods applied in MPI benchmarks.

benchmark	mean	min	max	dispersion metric
mpptest [5]	min of means			✗
SKaMPI [17]	✓			std. error
OSU [15]	✓	✓	✓	✗
Intel MPI [9]	✓	✓	✓	✗
MPIBlib [10]	✓			CI of the mean (default 95%)
MPIBench [7]	✓	✓		sub-sampled data
`mpicroscope` [20]	✓	✓	✓	✗
Phloem MPI Benchmarks [16]	✓	✓	✓	✗

Gil et al. presented a study on micro-benchmarking on the JVM, in which they show that the mean execution time over several JVM invocations may significantly differ [4]. The described effect is very relevant to the work presented here as our micro-benchmark also needs to start an environment (the MPI environment using `mpirun`), which in turn might affect the observed mean runtime.

3. MPI BENCHMARKING IN A NUTSHELL

We now summarize existing methods for benchmarking MPI implementations.

3.1 Common MPI Measurements

Several MPI benchmark suites have been proposed in the literature. We summarize commonly known benchmarks in Table 1, which also lists the statistical methods used to present benchmark results. The information shown in this table was gathered to the best of our knowledge since some benchmarks like SKaMPI have been released in many incarnations and some other ones like MPIBench are currently not available for download. Thus, for some of them we rely on the respective articles describing the benchmarks.

The program `mpptest` was one of the first MPI benchmarks [5] and has been part of the MPICH distribution. Gropp and Lusk carefully designed `mpptest` to allow reproducible measurements for realistic usage scenarios, and common pitfalls of MPI performance measurements were pointed out, e.g., ignoring cache effects. In order to get reproducible measurements, n consecutive calls to an MPI function are timed and the mean $\bar{t}_i = t/n$ of these n observations is computed. The 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 [17]. It features a domain-specific language for describing MPI benchmarks, which helps developers extending SKaMPI for their own measurements. SKaMPI also provides synchronization primitives in addition to the commonly used `MPI_Barrier`. For each benchmark executed in SKaMPI, the arithmetic mean and the standard error are reported. SKaMPI uses an iterative measuring process, and thus measurements of specific MPI functions are repeated until the current standard error is below some predefined maximum relative standard error, i.e., the coefficient of variation of the sample mean is small.

MPIBlib by Lastovetsky et al. [10] works similarly to SKaMPI as it computes a confidence interval of the mean based on the current sample and stops the measurement if the sample mean is within a predefined range from the confidence interval, e.g., 5% difference.

The benchmarks Intel MPI [9], `mpicroscope` [20] and OSU [15] repeat measuring the runtime of a specific MPI function for a predefined number of times. Then, they report the minimum, the maximum, and the mean runtime from that sample. `mpicroscope` attempts to reduce the number of measurements using a linear (or exponential) decay of repetitions, i.e., if in a sample of n consecutive MPI calls no new minimum execution time can be found, the number of repetitions is decreased.

Grove and Coddington developed MPIBench [7], which, in addition to mean and minimum runtime, 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 runtimes 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, which is denoted as t_{99} and then define $t_{thresh} = t_{99} \cdot a$ for some constant $a >= 1$ (default $a = 2$). Grove also shows the distribution of measurements obtained from benchmarking the `MPI_Isend` function with different message sizes [6, p. 127], highlighting the fact that the execution time of MPI functions is not normally distributed.

3.2 Rigorous Analysis

In the present article, we address the problem of determining which MPI implementation is better on a given machine through benchmarking, where "better" means faster. Currently, an experimenter would select one of the MPI benchmarks available and run the provided set of benchmarks for two or more MPI implementations. Then, the resulting graphs showing mean, minimum, or maximum runtime can be compared, and one can draw conclusions based on these results. The problem is that without a rigorous statistical analysis we cannot quantify a statistical confidence whether an observation is repeatable or a result of chance.

A possible solution for comparing variants is to apply methods of statistical hypothesis testing. The problem is that the informative value of the results provided by hypothesis tests strongly depends on fulfilling the test's assumptions. For example, the commonly applied Student's t-test assumes that the population follows the normal distribution, and that the individual observations of each sample are independent [11]. The non-parametric Wilcoxon–Mann–Whitney test for comparing two samples requires that observations are independent and additionally assumes that the sample distributions are similar.

4. EXPERIMENTAL FACTORS

To apply the right statistical hypothesis test, we need to understand our data first. This section is therefore dedicated to the discovery and analysis of experimental factors.

4.1 Experimental Setup

In all our experiments presented in this article, we measured the time for completing a single MPI function using the method shown in Algorithm 1. Before the start of a benchmark run, the experimenter chooses the number of observations *npe* (sample size) to be recorded for an individual test case. A test case consists of the triple MPI function, message (buffer) size, and number of processes. At the beginning of an experiment, all send and receive buffers of every process are allocated and initialized. Then, the selected MPI function is measured *npe* times, while the start of each individual measurement is synchronized using `MPI_Barrier`. Since each MPI process now holds an array containing *npe* measurements, we apply a reduction operation on that array (in this case MAX) and collect the results at the root process.

Algorithm 1 MPI timing procedure.

```
1: procedure TIME_MPI_FUNCTION(func, m, npe) // func - MPI function, m - message size, npe - nb. of observations
   // all processes
2:     initialize send and recv buffers according to m and func
3:     initialize time array 'times' with npe elements
4:     for obs in 1 to npe do
5:         MPI_Barrier()
6:         start = MPI_Wtime()
7:         execute func
8:         times[obs] = MPI_Wtime() - start
9:     MPI_Reduce(times, npe, MAX, root_proc)
   // at root process only
10:    for obs in 1 to npe do
11:        print times[obs]
```

We point out that the actual timing procedure of MPI functions has been subject to much previous research [8]. However, the actual timing procedure should not be in the focus of the present article, and thus for designing and analyzing experiments, we have used the method shown in Algorithm 1.

If not stated otherwise, we remove outliers from each sample by applying Tukey's outlier filter. When this filter is applied, all points from the sample are removed that are either smaller than $Q_1 - 1.5 \cdot IQR$ or larger than $Q_3 + 1.5 \cdot IQR$. IQR denotes the interquartile range between quartiles Q_1 and Q_3. In addition, we create exactly one process per compute node and pin this process to core 0 of a multisocket or multicore node.

The parallel machines used for conducting our experiments are summarized in Table 2. On the *TUWien* system we have dedicated access to the entire cluster. The *G5k (Edel)* system belongs to Grid'5000[1], which features the OAR job scheduler that allowed us to gain exclusive access to a full cluster comprising nodes connected to the same Infiniband switch. On *VSC-1*, we also made sure that our allocations lead to dedicated nodes, however we have no dedicated access to the switches as for the other two machines.

4.2 Sampling Distributions of MPI Timings

The most important requirement for meaningful results obtainable from hypothesis tests is obeying the assumptions of each statistical test. A common assumption of most hypothesis tests is that the data must follow a specific probability distribution to be applicable.

Commonly used dispersion measures such as the 95% confidence interval of the mean require that the data are normally distributed. For example, MPIBlib [10] measures the time for a given MPI function iteratively and estimates the confidence interval of the mean based on the t-distribution after each iteration.

As mentioned above, Grove already examined several resulting distributions of MPI function timings [6, p. 127]. In his dissertation, the experimental data presenting the distribution of times for `MPI_Isend` looked far from being normally distributed. In order to test whether we would observe similar distributions on our machines, we first ran a large set of MPI experiments to investigate the distribution of timings. The

[1] `http://www.grid5000.fr`

Table 2: Overview of parallel machines used in the experiments.

name	nodes	interconnect	MPI libraries
TUWien	36 Dual-Socket Opteron 6134 @ 2.3 GHz	Infiniband QDR MT4036	NEC MPI/LX 1.2.8, MVAPICH 2-1.9
VSC-1	436 Dual-Socket Xeon 5550 @ 2.66 GHz	Infiniband QLogic 12200	Intel MPI 4.1
G5k (Edel)	72 Dual-Socket Xeon E5520 @ 2.27 GHz	Infiniband QDR MT26428	MVAPICH 2-1.9

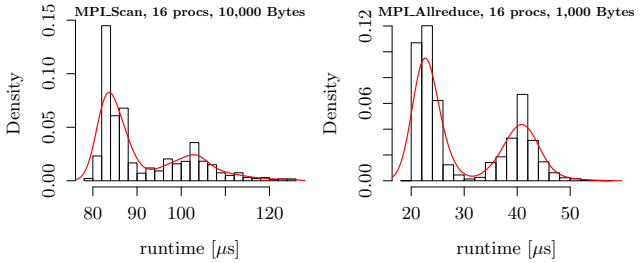


Figure 1: Histogram of the time needed to complete a call to `MPI_Scan` with 10,000 Bytes (left) and to `MPI_Allreduce` with 1,000 Bytes (right) on *TUWien*.

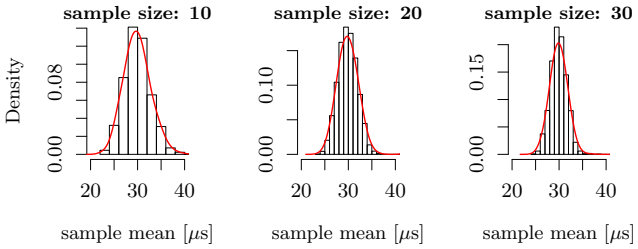


Figure 2: Distribution of sample means when sampling using different sample sizes from the probability distribution for `MPI_Allreduce` (cf. Figure 1).

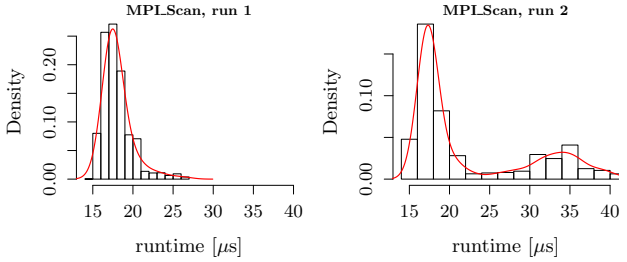


Figure 3: Distribution of the time needed to complete a call to `MPI_Scan` (with 250 Bytes and 16 processes) for two distinct calls to `mpirun` (run 1/run 2). Sample contains 10,000 observations and was taken on *TUWien*.

experiments were conducted for various MPI functions such as `MPI_Bcast`, `MPI_Allreduce`, `MPI_Alltoall`, or `MPI_Scan`. Figure 1 shows the distribution of runtimes for 10,000 calls to `MPI_Scan` with 10,000 Bytes and to `MPI_Allreduce` with 1,000 Bytes, both for 16 processes on 16 nodes (one process per node). We used the kernel density estimator from R (`density`) to obtain a visual representation of the sampling distribution. The sampling distribution is 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 almost all tests with small execution times; in this example the largest times were about 120 μs. Similar distributions are visible for experiments with `MPI_Alltoall` and `MPI_Bcast`, and also on the two other parallel machines *VSC-1* and *G5k (Edel)*.

Since these skewed distributions of runtimes do not follow a normal distribution, we must be careful when computing statistics on the data such as the confidence interval for the mean. A confidence interval for the mean is only meaningful when the distribution is normal, which is clearly not the case. However, the central limit theorem (CLT) states that the distribution of means of repeated samples is normal if the sample size is large enough. The problem in practice is usually to determine how large the sample size should be such that the CLT holds. Most textbooks like Lilja [12] or Ross [18] state that a sample size of 30 should be large enough to obtain a normally distributed mean. However, in a recent study by Chen et al. [2], the authors drew 150 samples with varying sizes from 10 to 280 randomly, and claimed that only sample means obtained from samples of size 280 follow a normal distribution. In order to examine how large the samples should be in the case of our MPI runtime distribution, such that the means are normally distributed, we have experimented with the density functions obtained previously. We drew 3,000 samples with 10, 20, or 30 observations each, and built a histogram of the sample means. Figure 2 presents the histograms of sample means for different sample sizes for the `MPI_Allreduce` runtimes recorded for the experiments shown in Figure 1. We conclude that a sample size of 30 should be large enough that the normality assumption holds. We also contend that in the study of Chen et al. [2], the number of sample means (150) was too small to judge whether the sample means are normally distributed. In sum, we can say that if we want to compute a confidence interval for the population mean for a sequence of MPI function timings, the sample size should be at least 30.

4.3 Factor: The Influence of mpirun

When we conducted the sampling experiment shown before, we noticed that the distributions were slightly different between calls to `mpirun`. Note that Gil et al. had seen a significant shift in distribution means when micro-benchmarking JVM functions [4]. Thus, we examined next whether distinct calls to `mpirun` produce different sample means (statistically significant) or not.

This investigation is motivated by the results presented in Figure 3, which shows the sampling distribution of two different calls to `mpirun`. In each execution of `mpirun`, we measured the time of 10,000 individual calls to `MPI_Scan` with a buffer size of 250 Bytes (per process). The two distributions look similar, yet they have significant differences. The most obvious one is the occurrence of a second peak in the distribution on the right-hand side, whereas the left distribution has almost no right tail.

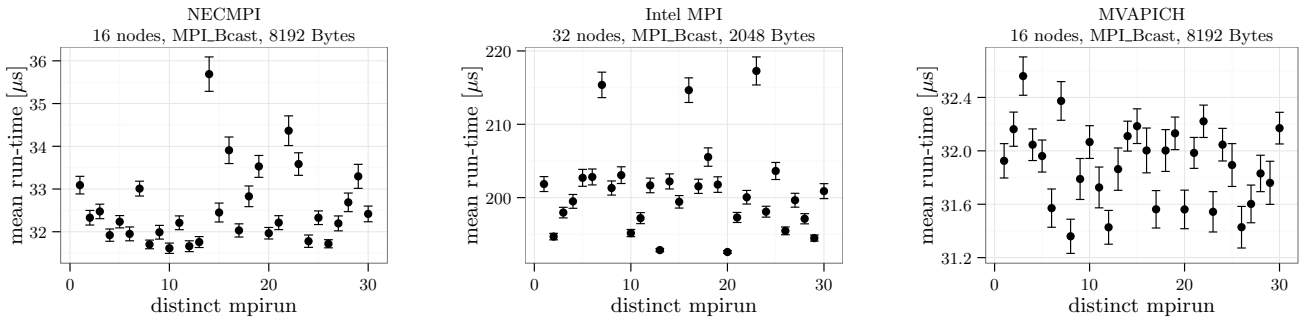


Figure 4: Mean and 95% confidence interval of the time to complete a call to `MPI_Bcast` for 30 distinct calls to `mpirun`. The measurements have been made with different MPI implementations and message sizes on *TUWien*, *VSC-1* and *G5k (Edel)* (from left to right).

To further investigate this finding, we conducted a series of experiments as follows: We execute 30 distinct calls to `mpirun` and in each `mpirun` we measure 1,000 times the execution time of a given MPI function. A subset of the results is shown in Figure 4[2], which presents the mean and the 95% confidence interval for the mean for the 30 distinct calls to `mpirun` for different problem instances. All experimental data was filtered using Tukey's approach as discussed above to avoid a bias due to large outliers. Figure 4 points out that the means obtained from distinct calls to `mpirun` are different, where the difference between means is usually not large (3-5%), yet significant. Such finding is a major obstacle to our goal of applying a hypothesis test soundly, as we can now not purely rely on a single call to `mpirun` to obtain experimental data. The call to `mpirun` leads to a specific system state (software or hardware), which affects the experimental outcome. As a result, the call to `mpirun` is a significant factor that influences the mean execution time of an MPI function and therefore needs to be considered in the experimental design.

4.4 Factor: Uncontrollable System Noise

Many runtime distributions shown so far exhibited a longer tail or a second smaller peak on the right. The question then becomes whether different measurements taken in sequence are independent from each other. The verification of the independence of measurements is important as virtually all statistical hypothesis tests assume that random variables are independent and identically distributed (iid). In case this assumption is violated, statistical measures would be misleading, e.g., a confidence interval for the mean could be too small [11, p. 47].

To answer the question whether measurements are independent, Le Boudec suggests to evaluate the autocorrelation of the experimental data [11]. Consequently, we computed the autocorrelation function (ACF) for all our experiments and show some of the results in the lag plots in Figure 5. Autocorrelation is typically used to test time-series data for randomness by estimating the correlation between two values of the same variable measured at different times as a function of the time lag between them. In particular, the

[2]Even though the selection of figures may seem unintuitive to the reader, we carefully chose the reported results to show the significance of our findings.

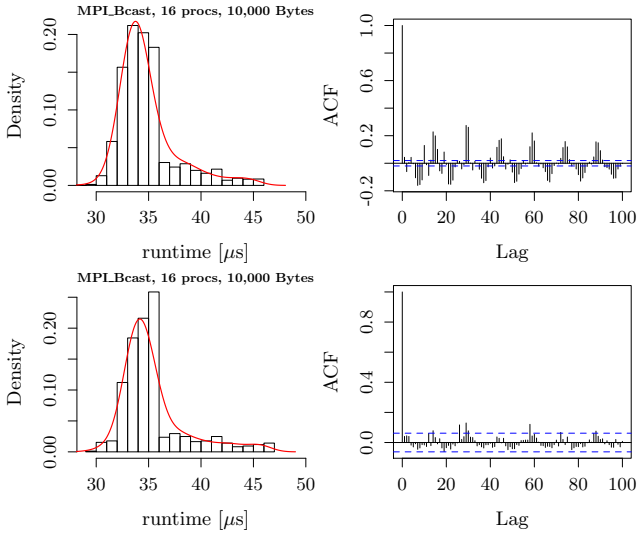


Figure 5: Distribution of runtimes for `MPI_Bcast` and the corresponding autocorrelation plot, for the experiments done *without* (top) and *with* random waiting (bottom) times on *TUWien*.

autocorrelation coefficient[3] at lag h is computed as the ratio of the autocovariance C_h and the variance C_0. A significant correlation of measurements in the data occurs if a line in the lag plot at a specific lag value exceeds the significance limit. If all values were chosen randomly, for example from a normal distribution, then no significant correlation can be found. As it can be seen in the graphs in the top row of Figure 5, the experimental data exhibit a significant correlation between measurements. The immediate consequence is that our assumptions for applying hypothesis tests do not necessarily hold true as many measurements are not independent.

This result also raises the question whether or not we can obtain statistically uncorrelated measurements. To answer that, we designed an experiment in which we introduce random waiting times after each measurement of an MPI function call. This experiment should also provide insight whether the correlation depends on the number of experi-

[3]`http://www.itl.nist.gov/div898/handbook/eda/section3/autocopl.htm`

ments (for example, it could be that the MPI implementation cleans up every 10th MPI call). Figure 5 compares two lag plots, where both plots show the results of 1,000 MPI runtimes obtained with `MPI_Bcast` on 16 processes and a message size of 10,000 Bytes. The histogram and the lag plot on top show the results without waiting times, and the graphics below show the results obtained when random waiting times after MPI calls were inserted. Interestingly, the lag plot on top exhibits a hidden sine function, which is dampened in the plot below. This suggests that MPI function timings experience a time correlation rather than a sequence correlation (e.g., every 10th run).

Inserting random waiting times needs to be carefully done as we measure in the microsecond range. We first experimented with POSIX functions like `usleep()` to implement waiting times, but this completely changed the resulting distribution (the mean times increased by 5-10 μs). For that reason, we implemented the waiting function in user space, where active waiting is realized by executing a loop in user space. The loop is executed k times, and the body of the loop performs simple bitshift operations. The number k is computed by first picking a random number r using the `rand()` function, and then setting $k = r \mod M$, where M denotes the maximum number of iterations to be performed.

Our experimental results show that we did not succeed in removing the correlation from all lag plots using random waiting times, and thus, the correlation between subsequent data points persisted. Originally, we had expected to see a change in the resulting histograms when the correlation between points is reduced. However, in the case in which we could remove the correlation, the histograms had a very similar shape. Thus, for the remaining experiments we used the measurements without waiting times, even if the confidence intervals could potentially be smaller.

Le Boudec [11] states that the correlation could be removed through sub-sampling of data. Indeed, if we draw sub-samples from our *npe* measurements and set the probability of selecting an element to $p = \frac{1}{32}$, the data points start to become uncorrelated. For now, we have not yet included sub-sampling in our data analysis method, but it is an option for future studies.

5. STATISTICALLY RIGOROUS AND REPRODUCIBLE MPI BENCHMARKING

Now that we have examined several factors that influence results of MPI micro-benchmarks, we propose a new method to compare the performance of MPI functions based on statistical hypothesis testing. Our main goal is to establish an experimental methodology that allows the reproducibility of the test outcome between several experiments.

The need for such a novel experimental method is motivated by the variance between MPI experiments obtained with Intel MPI Benchmarks and SKaMPI as shown in Figure 6(a) and Figure 6(b). Here, we ran the standard configuration for measuring `MPI_Bcast` using either the Intel(R) MPI Benchmarks 4.0 Beta or SKaMPI 5. The benchmarks were started with 16 processes on *TUWien*, and we repeated the measurement $n = 30$ times[4]. Then, for each message size m, we compute the minimum time reported over all runs as follows: $t^*_m = \min_{1 \le i \le n} t_{m,i}$. We note that in both cases these minimum times are minimum mean times as reported by the respective benchmark. We compute the *normalized runtime* of each measurement for a specific message size as follows: $r_{m,i} = t_{m,i}/t^*_m$, for all $i, 1 \le i \le n = 30$. The normalized runtime gives us a relative performance metric to compare runtime variances while being independent of the actual runtime. We can see in Figure 6(a) and Figure 6(b) that the normalized runtimes of Intel MPI and SKaMPI exhibit a larger variance for smaller message sizes. The reason is that system noise influences the experiments with small message sizes more than those with larger message sizes. This observation also supports our claim that `mpirun` has a non-negligible influence on the overall runtime, hence, we need to choose a design of the experiment that includes multiple `mpirun` calls.

[4]SKaMPI reports also measurements for message sizes that are not multiples of two. But for better readability, we removed some results for particular message sizes.

Algorithm 2 Design of MPI experiment.

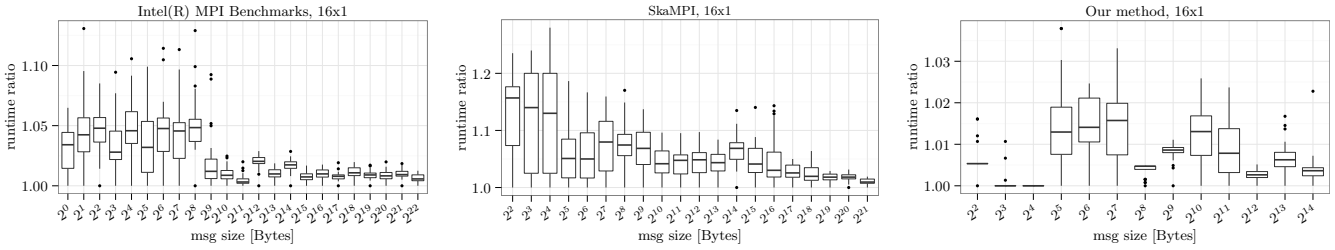
```
1: procedure DOE(lm, lf, n, npe, p)
   // lm - list of message sizes, lf - list of MPI functions, n -
   nb. of runs, npe - nb. of measurements per run, p - nb. of
   processors
2:     for i in 1 to n do
3:         mpirun -np p BENCHMARK lm lf npe
4: procedure BENCHMARK(lm, lf, npe)
5:     explist ← ()
6:     for all m in lm do
7:         for all f in lf do
8:             explist.add( TIME_MPI_FUNCTION(f, m, npe) )
9:     shuffle explist // create random permutation of calls
10:    for all exp in explist do
11:        call exp
```

5.1 Design of Reproducible Experiments

Our design of the experiment for a repeatable measurement of the performance of an MPI implementation is shown in Algorithm 2. The procedure to generate the experimental layout has five parameters, two of them being important for the statistical analysis: (1) n denotes the number of distinct calls to `mpirun` for each message size, and (2) *npe* the number of measurements taken for each message size in one of the n calls to `mpirun`. Hence, in total we measure the execution time of a specific MPI function for every message size $n \cdot npe$ times. In the procedure DOE in Algorithm 2, we issue n calls to `mpirun`, where the number of processors p stays fixed. To respect the principles of experimental design (randomization, replication, blocking) as stated by Montgomery [13], we randomize the experiment by randomly changing the order of experiments within a call to `mpirun`. Therefore, the procedure BENCHMARK received three parameters: the list of message sizes lm, the list of MPI functions to be tested lf, and the number of observations to be made for each message size *npe*. The procedure creates a list (*explist*) containing all *npe* experiments for all message sizes and MPI functions. We shuffle the order of elements in this list and then execute the items (experiments) of that random list.

5.2 Statistical Data Analysis

With the design of the experiment fixed, we now show how we compare the experimental results. First, we note that Algorithm 2 is executed for each MPI implementation (denoted A and B) that should be evaluated. Algorithm 3 presents the data analysis method applied for one implementation. Here, for each call i to `mpirun`, we extract the set of



(a) Intel(R) MPI Benchmarks 4.0 Beta (b) SKaMPI 5 (c) Our method

Figure 6: Distribution of normalized runtimes reported by Intel MPI (left), SkaMPI (center), and our method (right) for testing `MPI_Bcast` with various message sizes (16 processes, 1 process per node, *TUWien*).

Algorithm 3 Analyzing exp. for one implementation.

```
1: procedure ANALYZE_RESULTS(lm, lp, lf, n)
   // lm - list of message sizes, lp - list of processors, lf - list
   of MPI functions, n - nb. of runs
2:     for all m ∈ lm, p ∈ lp, f ∈ lf do
3:         for i in 1 to n do
4:             t_i = { time[m][p][f][i]   for all 1 ≤ j ≤ npe}
5:             t_i = remove_outlier(t_i)
6:             v[m][p][f][i] = median(t_i)
7:     print v // table with results
```

all measurements t_i for a particular combination of message size m, number of processors p and MPI function f. Then, we remove the outliers from t_i according to Section 4.1 and compute the median of all remaining measurements. Now, we have n medians for each message size. Not surprisingly, the distribution of these medians (not shown here) looks similar to the distribution of means shown in Figure 4. That means that the medians do not follow a normal distribution. As a consequence, if the number of medians is large, we could potentially use the t-test to compare the performance of different MPI implementations. But as we want to keep the number of required distinct calls to `mpirun` small, we apply the Wilcoxon–Mann–Whitney (Wilcoxon sum-of-ranks) test for comparing alternatives statistically. This test provides two advantages for us: (1) it is non-parametric and (2) it "does not require the assumption of normality" [18]. Altogether, we apply a Wilcoxon–Mann–Whitney test for each message size, i.e., we compare the n medians recorded for implementation A to n medians recorded for implementation B. The test reports whether the difference between the distributions of A and B is significant or not.

An example of our method is presented in Figure 7, in which we compare the runtime of `MPI_Bcast` for several message sizes and for two MPI implementations (A=MVAPICH, B=NEC MPI). We have used $n = 30$ distinct calls to `mpirun` (leading to 30 medians each) and $npe = 1,000$ observations for each message size. We apply the Wilcoxon–Mann–Whitney test on the distribution of $n = 30$ medians of both NEC MPI and MVAPICH. If the test reports that the result is significant, we visualize the computed p-values by plotting asterisks on top of the graph. One asterisk (*) would represent a p-value of $p \leq 0.05$, two asterisks denote $p \leq 0.01$, and three asterisks stand for $p \leq 0.001$. Figure 7 shows only three asterisks on top of each message size, i.e., the time to complete `MPI_Bcast` using NEC MPI is significantly shorter than using MVAPICH ($p \leq 0.001$).

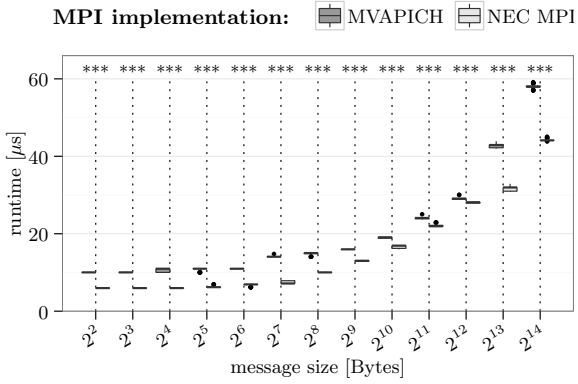


Figure 7: Comparison of the implementations of `MPI_Bcast` found in NEC MPI and MVAPICH on *TUWien* by applying the Wilcoxon–Mann–Whitney test for each message size.

5.3 Experimental Evaluation

The last question we answer is the question of repeatability. In this experiment, we perform the experiment and data analysis described previously, repeating the measurements 30 times. Similarly to the experiment with the Intel MPI Benchmarks and SKaMPI, we wanted to measure the variance between reported runtimes. The only problem is that our method is based on comparing a distribution of medians instead of a single value. Thus, for this experiment we combine all medians for a given message size into a single value by computing the mean of these medians. We then compute the variance (the normalized runtime) of these means as done for obtaining Figure 6. The resulting distribution of normalized runtimes is displayed in Figure 6(c) for *TUWien* and in Figure 8 for *G5k (Edel)*, which show that our method of benchmarking MPI functions has only little variance when executed multiple times. The relative error is mostly less than 2%, and only in some cases about 3%, which is clearly an improvement to the competing measurement methods.

6. DISCUSSION AND CONCLUSIONS

We have revisited the problem of micro-benchmarking MPI functions. The work was motivated by the need (1) to compare MPI implementation alternatives using a sound statistical analysis and (2) to allow the reproducibility of our experimental results. We have pointed out experimen-

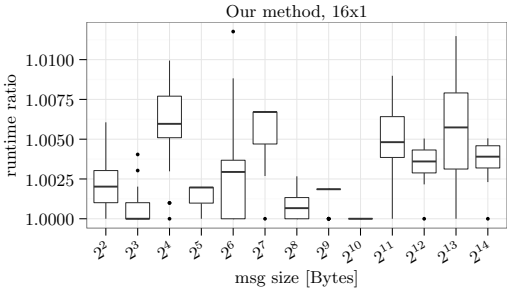


Figure 8: Distribution of the normalized runtimes reported for `MPI_Bcast` with various message sizes when running one MPI benchmark 30 times on *G5k (Edel)*.

tal factors that significantly influence the outcome of measurements, the influence of the `mpirun` call being the most prominent one (which was not previously mentioned in literature). Some experiments have also shown that subsequent measurements might be subject to periodic system noise, which could lead to dependent measurements and therefore to an underestimation of experimental variance. We have also shown how to design the experiment for measuring MPI functions such that the statistical analysis is sound and the results are reproducible. For the statistical comparison of two MPI implementations, we have demonstrated how to use the non-parametric Wilcoxon–Mann–Whitney test for hypothesis testing.

This research is far from being finished. In the future, it will be interesting to examine the time dependence of measurements in detail, e.g., why some autocorrelation plots show these sine functions. We will also try to reduce the number of experiments required to make a sound statistical analysis. So far, we have used 30 calls to `mpirun` and 1,000 observations, leading to 30,000 measurements for only one message size. There is surely a trade-off between time (number of measurements) and precision (statistical significance). Nonetheless, we need to find the smaller number of measurements required for a statistically significant outcome.

Acknowledgments

We thank Maciej Drozdowski (Poznan University of Technology) and Peter Filzmoser (Vienna University of Technology) for discussions and comments. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

7. REFERENCES

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

[2] T. Chen, Y. Chen, Q. Guo, O. Temam, Y. Wu, and W. Hu. Statistical performance comparisons of computers. In *HPCA*, pages 399–410. IEEE, 2012.

[3] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA*, pages 57–76, 2007.

[4] J. Gil, K. Lenz, and Y. Shimron. A microbenchmark case study and lessons learned. In *SPLASH Workshops*, pages 297–308, 2011.

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

[6] D. Grove. *Performance modelling of message-passing parallel programs.* PhD thesis, University of Adelaide, 2003.

[7] D. A. Grove and P. D. Coddington. Communication benchmarking and performance modelling of MPI programs on cluster computers. *The Journal of Supercomputing*, 34(2):201–217, 2005.

[8] 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*, 25(4):241–258, 2010.

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

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

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

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

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

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

[15] OSU MPI benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[16] Phloem MPI Benchmarks. https://asc.llnl.gov/sequoia/benchmarks/.

[17] R. Reussner, P. Sanders, and J. L. Träff. SKaMPI: a comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.

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

[19] 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*, 25(10):1410–1426, 2013.

[20] J. L. Träff. mpicroscope: Towards an MPI benchmark tool for performance guideline verification. In *EuroMPI*, pages 100–109, 2012.

[21] J. Vitek and T. Kalibera. R3: Repeatability, reproducibility and rigor. *SIGPLAN Not.*, 47(4a):30–36, Mar. 2012.