# On the Impact of Synchronizing Clocks and Processes on Benchmarking MPI Collectives [*]

Sascha Hunold
hunold@par.tuwien.ac.at

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

Vienna University of Technology
Faculty of Informatics, Institute of Information Systems
Research Group for Parallel Computing

## ABSTRACT

We consider the problem of accurately measuring the time to complete an MPI collective operation, as the result strongly depends on how the time is measured. Our goal is to develop an experimental method that allows for reproducible measurements of MPI collectives. When executing large parallel codes, MPI processes are often skewed in time when entering a collective operation. However, to obtain reproducible measurements, it is a common approach to synchronize all processes before they call the MPI collective operation. We therefore take a closer look at two commonly used process synchronization schemes: (1) relying on `MPI_Barrier` or (2) applying a window-based scheme using a common global time. We analyze both schemes experimentally and show the strengths and weaknesses of each approach. As window-based schemes require the notion of global time, we thoroughly evaluate different clock synchronization algorithms in various experiments. We also propose a novel clock synchronization algorithm that combines two advantages of known algorithms, which are (1) taking the inherent clock drift into account and (2) using a tree-based synchronization scheme to reduce the synchronization duration.

## CCS Concepts

•**Software and its engineering** → *Massively parallel systems;*

## Keywords

MPI, benchmarking, distributed clock synchronization

## 1. INTRODUCTION

The Message Passing Interface (MPI) is still one of the dominating programming models on today's supercomputers. Therefore, MPI libraries are major building blocks of virtually all High Performance Computing (HPC) applications. For that reason, evaluating the performance of MPI libraries is necessary for detecting possible optimization directions.

Several MPI benchmark suites have been developed over the last two decades. However, since the MPI standard has been continuously updated, there is still a need for benchmarking new features or functions of the libraries. In addition, modern machines and operating systems offer new features for developers and experimenters, such as pinning threads to CPUs, changing the Dynamic Voltage and Frequency Scaling (DVFS) levels, or accessing high resolution timers (e.g., `RDTSC`). In order to take advances of hardware and software layers into account and to assess their impact on MPI libraries, we need to revisit the problem of accurately benchmarking MPI functions.

We have already examined the reproducibility of run-times reported by different MPI benchmark suites [7]. In this work, we have shown how to fairly compare performance data of different MPI libraries. However, in our previous work, we have examined the run-times of MPI functions when synchronizing processes using the `MPI_Barrier` call. In the present article, we examine the applicability of the window-based process synchronization for MPI benchmarking.

The use of `MPI_Barrier` for process synchronization has advantages and disadvantages. One major advantage is its portability across different MPI implementations. The drawbacks of `MPI_Barrier` are that it is not guaranteed that processes leave the barrier synchronously and also that `MPI_Barrier` messages may influence the MPI operation currently being measured [5]. To overcome the problems introduced by `MPI_Barrier`, some MPI benchmark suites offer a *window-based* scheme for measuring performance, in which processes use a logical global time for synchronization.

We motivate the present article by showing the results of a single, simple MPI experiment with SKaMPI [14], in which we apply the window-based synchronization scheme. We configured SKaMPI to measure the run-time of `MPI_Bcast` with a payload of 8192 Bytes for a minimum of 100 000 times. Figure 1 shows the results of this experiment[1]. An *adaptive* window size means that we used the default method of SKaMPI to increase the window size dynamically if the

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

*EuroMPI '15, September 21 - 23, 2015, Bordeaux , France*
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3795-3/15/09…$15.00
DOI: http://dx.doi.org/10.1145/2802658.2802662

[1]Even though we configured SKaMPI to measure the run-time of `MPI_Bcast` only 100 000 times, roughly 150 000 measurements were recorded. The reason is that if some measurements cannot be completed within a time window, SKaMPI repeats the measurement until a predefined number of maximum repetitions is reached (in SKaMPI: `max_repetitions` × 1.5).

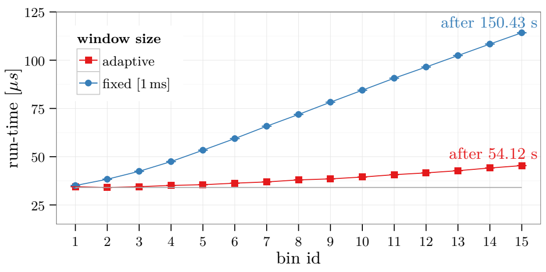


**Figure 1: Run-time of `MPI_Bcast` over time as reported by SKaMPI. The data is aggregated into bins of** $10\,000$ **repetitions** ($8192$ **Bytes,** $16 \times 1$ **processes, MVAPICH 2.1a,** ***TUWien*****,** $150\,000$ **measurements).**

MPI call cannot be finished within the given window. We also repeated the measurements with a *fixed* window size of 1 ms to improve the reproducibility of the experiment, as the dynamic adjustment of the window size may change the benchmark behavior from one run to another.

For reasons of clarity, we present the run-times accumulated into bins of size 10 000. The total run-time of the benchmark depends on the window size. In the case of a fixed window size, the total run-time was 150.43 s, and 54.12 s for the adaptive window method. We can observe that both lines in this figure are constantly increasing over time (bin id). The reason is that the distributed clocks are drifting over time. Thus, the processes that should synchronously start measuring the given MPI function will, in fact, be skewed in time. As a consequence, some processes enter the MPI call earlier than others, which is reflected in a larger overall run-time of the MPI function. The measurements using the *fixed* window size emphasize this effect, as the window is larger (on average) than in the *adaptive* case, and thus more time will have passed, which directly translates into a larger clock drift between processes.

One might assume that this is a problem of SKaMPI only. NBCBench is another well-known MPI benchmarking application that applies a window-based synchronization strategy [5]. Unfortunately, this benchmark also suffers from the problem of the clock drift as it only accounts for the clock offset.

The experimental results shown above highlight the problems of benchmarking MPI functions accurately. The first problem is that each `MPI_Barrier` implementation has an unpredictable behavior. The second problem is that the window-based schemes, implemented in SKaMPI and NBCBench, have an inherent error that grows linearly with the number of experiments conducted. To address these issues, we make the following contributions:

1. We empirically analyze the impact of different process synchronization methods on MPI benchmarking experiments.
2. We provide a detailed evaluation of several clock synchronization algorithms. We compare the time needed for clock synchronization to their accuracy.
3. We propose a novel clock synchronization algorithm called HCA, which provides a good trade-off between time and accuracy.

We start by giving an overview of the state of the art in the field of MPI benchmarking applications in Section 2. We briefly introduce our experimental setup in Section 3. We analyze the impact of different process synchronization methods such as the use of `MPI_Barrier` and the window-based schemes in Section 4 and Section 5, respectively. Section 6 presents our novel clock synchronization algorithm. We experimentally evaluate several clock synchronization algorithms in Section 7, before we conclude in Section 8.

## 2. PROCESS SYNCHRONIZATION USED IN MPI BENCHMARKS

We first summarize how process synchronization and time measurements in MPI benchmarking applications are usually done. Table 1 contains a list of well-known MPI benchmark suites, for which we add information about their respective process synchronization method. We also show the pseudocode of each measurement scheme applied in these benchmarks in Figure 2.

The majority of the listed MPI benchmark suites employ measurement scheme (1), which synchronizes processes by calling `MPI_Barrier` after each individual measurement has been completed (MPIBench [2], `mpicroscope` [16], MPIBlib [11], OSU Micro-Benchmarks [12]).

The Phloem MPI benchmarks apply scheme (3), which can optionally synchronize individual experiments (line 5), although only the overall time of all MPI calls is measured.

`mpptest` [1] and Intel MPI Benchmarks [8] use measurement scheme (2), where processes are synchronized only once, before starting a pre-defined number of subsequent MPI calls. Only one time measurement is taken for an entire batch of MPI calls.

SKaMPI [14, 17] and NBCBench [3, 5] support measurement schemes (1) and (4). The latter scheme synchronizes processes between MPI calls by letting processes wait for a globally-known start time of the next measurement window.

This window-based synchronization method works as follows: (1) The distributed clocks of all participating MPI processes are synchronized relative to a reference clock. To this end, each MPI process computes its clock offset relative to a master process (e.g., process 0) to be able to normalize its time to the reference clock. (2) The master process selects a start time, a point in time that lies in the future, and broadcasts this start time to all participating processes. (3) Since each process knows the time difference to the master process, all processes are now able to wait for this start time before executing the respective MPI function synchronously. 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 is called a “window”.

Hoefler et al. showed how blocking and non-blocking collective MPI operations could be measured scalably and accurately [4]. They pointed out that interleaving calls to `MPI_Barrier` and to the MPI function to be timed can lead to pipelining effects, which could distort the results. To overcome this problem, they applied a window-based synchronization scheme, which was inspired by SKaMPI. However, Hoefler et al. highlighted the fact that the SKaMPI synchronization method is less scalable, since it requires $\mathcal{O}(p)$ rounds. Hence, they developed a time synchronization method that only needs $\mathcal{O}(\log p)$ rounds to complete. Both SKaMPI and NBCBench perform a periodical re-adjustment of the window size to cope with run-times that are too long

**Table 1: Comparison of process synchronization methods used by MPI benchmark suites.**

| benchmark name | ref. | version | synchronization between indiv. MPI calls | sync. scheme |
|---|---|---|---|---|
| Intel MPI Benchmarks | [8] | 4.0.0 | none | 2 |
| MPIBench | [2] | 1.0beta | `MPI_Barrier` | 1 |
| MPIBlib | [11] | 1.2.0 | `MPI_Barrier` | 1 |
| `mpicroscope` | [16] | 1.0 | `MPI_Barrier` | 1 |
| `mpptest` | [1] | 1.5 | none | 2 |
| NBCBench | [3] | 1.1 | `MPI_Barrier` or window-based | 1, 4 |
| OSU Micro-Benchmarks | [12] | 4.4.1 | `MPI_Barrier` | 1 |
| Phloem MPI Benchmarks | [13] | 1.0.0 | `MPI_Barrier` or none (only initial `MPI_Barrier`) | 3 |
| SKaMPI | [14] | 5.0.4 | `MPI_Barrier` or window-based | 1, 4 |

```
1: for $obs$ in 1 to $nrep$ do
2:     MPI_Barrier
3:     $s\_time$ = MPI_Wtime
4:     execute MPI function
5:     $e\_time$ = MPI_Wtime
```

(1)

```
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
```

(2)

```
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
```

(3)

```
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:     $s\_time$ = GET_TIME()
6:     execute MPI function
7:     $e\_time$ = GET_TIME()
```

(4)

**Figure 2: Measurement schemes found in MPI benchmarks. In scheme (4), depending on the implementation, `Get_Time` returns the local time (measured with `MPI_Wtime` or `RDTSC`) or a logical global time.**

to fit into the synchronization window.

Jones and Koenig proposed a time synchronization method that considers the clock drift between distributed processes [9]. Their method is based on the assumption that different clocks drift linearly in time. Each process learns a linear model of the clock drift by exchanging ping-pong messages with a single reference process. After the linear model of the clock drift has been computed (using a linear regression), each process can determine a logical global time by adjusting its local time relative to the time of the reference process. In the ping-pong phase, local times are exchanged between the reference process and the other processes. When processes receive the local time from the reference process, some time has already passed, which is the time for sending the message containing the timestamp. Therefore, received timestamps are corrected by half of the mean round-trip time (RTT).

For a more detailed description of SKaMPI, NBCBench, and the algorithm of Jones and Koenig, we refer the reader to our technical report [6], which also provides pseudocodes for each algorithm.

## 3. EXPERIMENTAL SETUP

### 3.1 Parallel Machines

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. On *VSC-3* and *Cartesius*, we made sure that our allocations are composed of dedicated nodes only. However, we have no dedicated access to the switches on these machines.

### 3.2 Measuring Time

Hoefler et al. discussed the problem of the `MPI_Wtime` resolution, which is typically not high enough for measuring short time intervals [5]. They therefore use the CPU's clock register to count the number of processor cycles since reset. More specifically, Netgauge implements 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.

For all experiments presented in this article, we performed our measurements using the equivalent `RDTSCP` call, which guarantees instruction serialization, i.e., `RDTSCP` makes sure that all previous instructions have been executed when the timestamp counter is read. 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 Notation

The benchmarks NBCBench and Netgauge are related. For example, Hoefler et al. state: "We used our new findings to implement a new benchmark scheme in the benchmark suite Netgauge. The implementation bases on NBCBench [..]" [5]. For that reason, we use NBCBench to refer to the MPI benchmark and Netgauge to the algorithm to synchronize clocks hierarchically.

We use the following notation in the remainder of the article, which we borrowed from Kshemkalyani and Singhal [10]. The *clock offset* is the difference between the time reported by two clocks. The *clock skew* is the difference in the frequencies of two clocks, and the *clock drift* is the difference between two clocks over a period of time.

## 4. PROBLEMS WITH MPI_BARRIER SYNCHRONIZATION

We now investigate how a call to `MPI_Barrier` for the purpose of process synchronization affects the measurements of MPI functions. The advantage of synchronizing processes with `MPI_Barrier` is that this method is independent of a logical global clock, and thus, subsequently measured run-times will not experience a drift. Further, processes will typically require a shorter waiting time compared to a window-based scheme, which makes the `MPI_Barrier`-benchmarks usually faster to complete a set of experiments. However, we need to examine how well the synchronization using `MPI_Barrier` really works in practice.

Typically, MPI benchmarks that use `MPI_Barrier` to syn-

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

| name | nodes | interconnect | MPI libraries |
|---|---|---|---|
| *TUWien* | 36 Dual Opteron 6134 @ 2.3 GHz | IB QDR MT4036 | NEC MPI/LX 1.2.11<br>MVAPICH 2.1a |
| *VSC-3* | 2000 Dual Xeon E5-2650V2 @ 2.6 GHz | IB QDR-80 | Intel MPI 5<br>MVAPICH 2.0a-qlc |
| *Cartesius* | 64 Dual Xeon E5-2450V2 @ 2.5 GHz | IB Mellanox ConnectX-3 FDR | Intel MPI 4.1 |

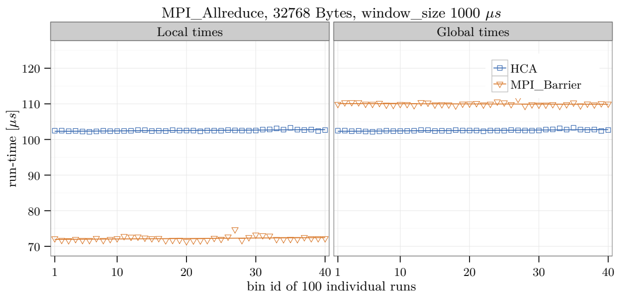


**Figure 3: Run-time of `MPI_Allreduce` obtained when using window-based synchronization and `MPI_Barrier`-based synchronization and two different approaches for computing the run-time ($32$ KiBytes, $16 \times 1$ processes, $4000$ runs, bin size: $100$, MVAPICH 2.0a-qlc, *VSC-3*).**

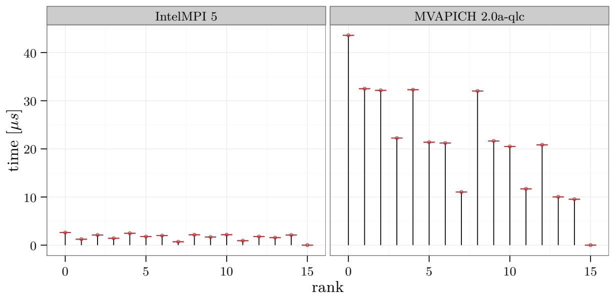


**Figure 4: Synchronization imbalance of `MPI_Barrier` implementations. Exit times of each process relative to the first process that leaves `MPI_Barrier` (mean of $1000$ measurements and $95\,\%$ confidence intervals, $16 \times 1$ processes, one `mpirun`, HCA synchronization, window size: $100\,\mu s$, *VSC-3*).**

chronize processes between measurements define the run-time of an MPI function as the maximum local run-time measured on each process. The problem with this way of estimating the run-time is that it is assumed that all processes leave `MPI_Barrier` and enter the MPI call to be benchmarked almost synchronously.

When we compared measurements obtained with window-based and `MPI_Barrier`-based schemes, we encountered cases for which we initially had no explanation. The graph on the left-hand side of Figure 3 shows one of these experiments, where we compare the run-time of `MPI_Allreduce` obtained with a window-based scheme (in which clocks were synchronized using HCA) to the run-time obtained when synchronizing with `MPI_Barrier`. The HCA synchronization method will be introduced in Section 6, but for now it is only important that it gives us accurate logical global clocks. The mean run-time of `MPI_Allreduce` when applying the `MPI_Barrier` synchronization was about $70\,\mu s$ (computed by using local run-times), while the same call took approximately $100\,\mu s$ using the window-based scheme. As this difference seemed very large, it needed further investigation.

We repeated the experiment for `MPI_Allreduce`, but this time, while we still synchronized processes using `MPI_Barrier`, we measured global times on each process using our HCA method to normalize local times to the *root*'s reference clock. Thus, instead of taking the maximum run-time over the local run-times of $p$ processes, we computed the run-time as follows: $\max_{0 \leq r<p}(\text{global end time}_r) - \min_{0 \leq r<p}(\text{global start time}_r)$. The chart on the right-hand side in Figure 3 shows the resulting run-times of `MPI_Allreduce` for both synchronization methods (`MPI_Barrier` and window-based with HCA), where all times were obtained using globally synchronized clocks. Now, the resulting run-times are much closer and their difference can reasonably be explained by the way the two synchronization schemes work.

Nevertheless, we still need to explain the gap between the observed run-times, whether we rely on local or global times to determine the overall run-time. Ideally, both run-time computation methods should lead to similar results. Therefore, we investigated the skew of MPI processes when they exit the `MPI_Barrier` function. For this purpose, we applied the HCA method to synchronize clocks and recorded the global timestamp of each process at the end of the `MPI_Barrier` call. The results of this experiment are shown in Figure 4. The graphs compare the process skew after completing `MPI_Barrier`, measured with Intel MPI 5 (left) and MVAPICH 2.0a-qlc (right). Surprisingly, a call to `MPI_Barrier` using MVAPICH 2.0a-qlc resulted in a large process skew. In particular, the mean exit times between process 0 and process 15 differed by more than $40\,\mu s$. This finding directly explains why the measurements in the previous experiment (cf. Figure 3) showed such a large difference in run-time.

The experiments discussed previously only indicate the potential consequences of using `MPI_Barrier` to synchronize processes for MPI benchmarking results. We show results obtained with MVAPICH 2.0a-qlc, even though it is not the latest version of MVAPICH, but the one that was preinstalled on the system and for which we experienced this significant process skew. However, these results are not meant to evaluate the performance of MVAPICH, but rather to point out potential pitfalls when relying on `MPI_Barrier` for synchronization.

Last, we would like to demonstrate how misleading the run-time measurements can be when the experimenter relies on an `MPI_Barrier` synchronization. Figure 5 compares the normalized run-times of `MPI_Bcast` obtained with ei-

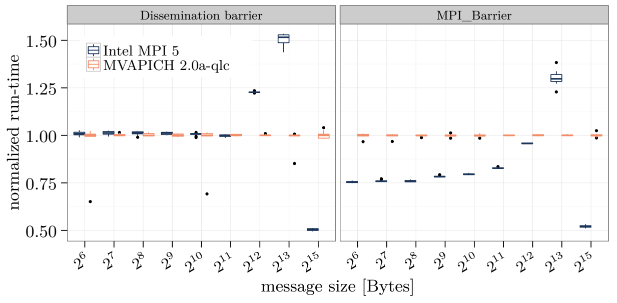


**Figure 5: Distribution of normalized median run-times of MPI_Bcast for Intel MPI 5 and MVAPICH 2.0a-qlc, synchronization based on an external dissemination barrier or MPI_Barrier ($16 \times 1$ processes, 1000 measurements, 10 mpiruns, *VSC-3*).**

ther an external benchmark-provided dissemination barrier (cf. [15]) or the barrier implementation provided by each library. We have executed 10 distinct calls to `mpirun`, in each of which 1000 measurements were recorded. We compute the median of each sample and normalize the run-times for one message size to the median run-time of these 10 medians for MVAPICH 2.0a-qlc. We observe, especially for the smaller message sizes ($2^6$ Bytes to $2^{11}$ Bytes), that there is no clear winner between Intel MPI 5 and MVAPICH 2.0a-qlc when our own barrier implementation is used (left-hand side). However, when we employ the library-provided `MPI_Barrier` implementation for synchronizing processes, we see a significant performance difference between the libraries. In this case, one could easily draw wrong conclusions.

*As a result, we contend that an MPI benchmark should provide its own barrier implementation for meaningful and fair comparisons.*

## 5. PROBLEMS WITH WINDOW-BASED PROCESS SYNCHRONIZATION

The window-based measurement scheme, introduced in Section 2, requires accurately synchronized, distributed clocks. We now look at some pitfalls when applying this scheme.

In the context of MPI benchmarks, Hoefler et al. have shown that two processor clocks are drifting over time [5], and also that the clock drift is linear in time. We re-conducted their experiment to examine the clock drift on our current machines, but using a finer resolution than what was done by Hoefler et al. [5] (we only measure in the range of seconds instead of hours). Figure 6 shows that the maximum clock drift between two hosts of our cluster is about 700 µs (|−400 µs|+300 µs) after 50 s. Thus, not accounting for such a clock drift will lead to highly inaccurate window-based measurements, in the range of microseconds after only a few seconds of conducting measurements. Hence, a window-based scheme must precisely deal with both the clock offset and the clock drift.

Now, we examine how accurately the window-based synchronization schemes of Netgauge and SKaMPI work in practice. We designed an experiment that measures the individual run-times of 4000 consecutive calls to `MPI_Bcast` using 512 processes (distributed over 32 compute nodes). The process synchronization between calls to `MPI_Bcast` is

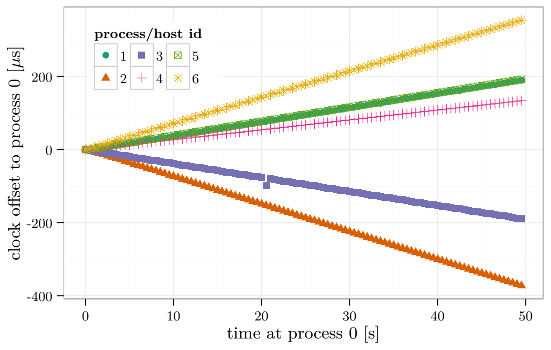


**Figure 6: Clock drift between a reference host and six other hosts on *TUWien*.**

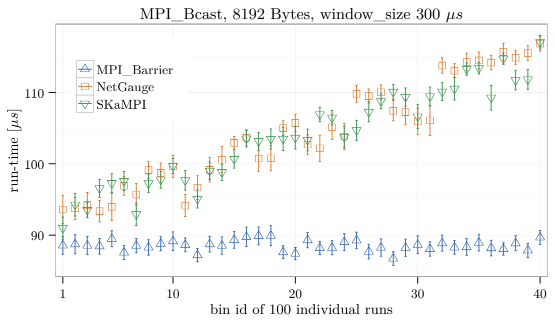


**Figure 7: Run-time of MPI_Bcast (mean and 95 % confidence interval) with the synchronization methods of Netgauge, SKaMPI, and MPI_Barrier (8192 Bytes, $32 \times 16$ processes, 4000 measurements, 10 calls to mpirun, MVAPICH 2.0rc1, *TUWien*).**

either done by using an `MPI_Barrier` call or by applying the window-based scheme implemented in Netgauge and SKaMPI with a fixed window size. Figure 7 shows the development of the mean run-time of `MPI_Bcast` over time. For presentation purposes, we binned every 100 consecutive, individual measurements and only plotted the bin means and their confidence intervals in this figure. As expected, the run-time of `MPI_Bcast` stays relatively stable when we synchronize using `MPI_Barrier` (as this process synchronization method is independent of the clock). In contrast, the mean binned run-times increase over time when a window-based scheme is applied.

The underlying problem is that neither Netgauge nor SKaMPI consider the clock drift when synchronizing processes. Instead, both benchmarks "only" determine the clock offset between processes. To cope with this problem, they could perform a periodic clock re-synchronization. However, neither SKaMPI nor NBCBench implement a re-synchronization of the distributed clocks to counterbalance the clock drift.

*In conclusion, we contend that window-based process synchronization schemes need to consider the clock drift when computing the logical global clock.*

**Algorithm 1** HCA clock synchronization.

```
    $p$ - number of processes
    $r$ - current process rank (0 to $p-1$)
    $lm$ - linear model of the current process (defined by a *slope*
      and an *intercept*) to adjust the local clock to the reference
      time of *root*
    $l^{model}$ - array of $p$ linear models
    $l^{model}[0] = (0, 0)$ // *reference clock*
    *hierarchical_intercepts* - if defined, compute intercepts
      hierarchically (instead of directly between each $r$ and *root*)
    *start_time* - next window start time, updated after each sync
    *initial_time* - local timestamp used to adjust the local clock to
      the time 0 of the synchronization start
    maxpower = $2^{\lfloor log_2 p \rfloor}$

procedure SYNC_CLOCKS(N_FITPTS, N_EXCHANGES)
    *initial_time* = GET_TIME()
    // *compute linear models of each clock's drift relative to root*
    SYNC_CLOCKS_POW2(N_FITPTS, N_EXCHANGES)
    SYNC_CLOCKS_REMAINING(N_FITPTS, N_EXCHANGES)
    // *send final linear models from root to each process*
    MPI_SCATTER($l^{model}$, 1, T_PAIR_DOUBLE,
                    *lm*, 1, T_PAIR_DOUBLE, *root*)
    #ifndef *hierarchical_intercepts*
    COMPUTE_AND_SET_ALL_INTERCEPTS(*lm*)
    #endif
    MPI_BARRIER()
    *start_time* = GET_ADJUSTED_TIME() + *win_size*
    MPI_BCAST(*start_time*, 1, MPI_DOUBLE, *root*)
```

**Algorithm 2** Hierarchical linear models of the clock drift.

```
function GET_ADJUSTED_TIME
    return GET_TIME() - *initial_time*

procedure SYNC_CLOCKS_POW2(N_FITPTS, N_EXCHANGES)
    // *compute linear models of the clock drifts for processes with*
    // *indices between 0 and (maxpower − 1)*
    *round* = 1
    if $r \geq$ maxpower then return
    while $2^{round} \leq$ maxpower do
        if ($r$ mod $2^{round}$) == 0 then // *process with reference clock*
            *p_client* = $r + 2^{round-1}$
            *rtt* = COMPUTE_RTT($r$, *p_client*)
            LEARN_MODEL_HCA(N_FITPTS, N_EXCHANGES,
                                    *rtt*, $r$, *p_client*)
    #ifdef *hierarchical_intercepts*
            COMPUTE_AND_SET_INTERCEPT(NULL, *p_client*, $r$)
    #endif
    // *receive linear models collected by the client*
            MPI_RECV($l^{model}_{client}$, $2^{round-1}$, T_PAIR_DOUBLE, *p_client*)
            $l^{model}$[*p_client*] = $l^{model}_{client}[0]$ // *save client model*
            for $i$ in 1 to ($2^{round-1}-1$) do // *compute resulting models*
                $l^{model}$[*p_client* + $i$] = MERGE_LMS($l^{model}$[*p_client*],
                                                        $l^{model}_{client}[i]$)
        else if ($r$ mod $2^{round}$) == $2^{round-1}$ then // *client*
            *p_ref* = $r - 2^{round-1}$
            *rtt* = COMPUTE_RTT(*p_ref*, $r$)
            $l^{model}[r]$ = LEARN_MODEL_HCA(N_FITPTS, N_EXCHANGES,
                                                    *rtt*, *p_ref*, $r$)
    #ifdef *hierarchical_intercepts*
            COMPUTE_AND_SET_INTERCEPT($l^{model}[r]$, $r$, *p_ref*)
    #endif
    // *send all new linear models to the reference process*
            MPI_SEND($l^{model}[r]$, $2^{round-1}$, T_PAIR_DOUBLE, *p_ref*)
        *round* = *round* + 1

procedure SYNC_CLOCKS_REMAINING(N_FITPTS, N_EXCHANGES)
    // *compute linear models of the clock drifts for processes with*
    // *indices between maxpower and (p − 1)*
    if maxpower == $p$ then return
    if $r < p -$ maxpower then          // *process with reference clock*
        *p_client* = $r$ + maxpower
        *rtt* = COMPUTE_RTT($r$, *p_client*)
        LEARN_MODEL_HCA(N_FITPTS, N_EXCHANGES,
                                *rtt*, $r$, *p_client*)
    #ifdef *hierarchical_intercepts*
        COMPUTE_AND_SET_INTERCEPT(NULL, *p_client*, $r$)
    #endif
    else if $r \geq$ maxpower then          // *client*
        *p_ref* = $r$ − maxpower
        *rtt* = COMPUTE_RTT(*p_ref*, $r$)
        *lm* = LEARN_MODEL_HCA(N_FITPTS, N_EXCHANGES,
                                        *rtt*, *p_ref*, $r$)
    #ifdef *hierarchical_intercepts*
        COMPUTE_AND_SET_INTERCEPT(*lm*, $r$, *p_ref*)
    #endif
    *sub_comm* = create communicator comprising process ranks
                    (0, maxpower, maxpower + 1, . . . , $p-1$)
    if $r$ == *root* then
        MPI_GATHER(*lm*, 1, T_PAIR_DOUBLE,
                        *tmp_lm*, 1, T_PAIR_DOUBLE, *root*, *sub_comm*)
        for $j$ in 0 to ($p$ − maxpower − 1) do
            $q$ = maxpower + $j$
            $l^{model}[q]$ = MERGE_LMS($l^{model}[j]$, *tmp_lm*[$j+1$])
    else if $r \geq$ maxpower then
        MPI_GATHER(*lm*, 1, T_PAIR_DOUBLE,
                        *tmp_lm*, 1, T_PAIR_DOUBLE, *root*, *sub_comm*)

procedure COMPUTE_AND_SET_ALL_INTERCEPTS(*lm*)
    // *compute intercepts for the model lm of the current process r*
    if $r \neq$ *root* then
        COMPUTE_AND_SET_INTERCEPT(*lm*, $r$, *root*)
    else
        for $i$ in 0 to ($p-1$) s.t. $i \neq$ *root* do
            COMPUTE_AND_SET_INTERCEPT(*lm*, $i$, *root*)
```

## 6. A HIERARCHICAL CLOCK SYNCHRONIZATION METHOD

We want to combine the advantages of the synchronization algorithm developed by Jones and Koenig (JK) with the synchronization scheme applied by Netgauge. We propose a novel algorithm that synchronizes distributed clocks in a hierarchical way, but also takes the clock drift into account. Jones and Koenig already noted in their article that they had applied an $\mathcal{O}(p)$ scheme for better accuracy, "whereas a balanced $\mathcal{O}(\log p)$ scheme may complete in milliseconds with higher variance (owing to the multiple reference stratums)" [9]. We still want to explore the possibility of applying such an $\mathcal{O}(\log p)$ scheme to improve the scalability of the algorithm of Jones and Koenig.

Algorithm 1 shows the pseudocode of our novel HCA algorithm[2]. The computational structure of the HCA algorithm works similarly to the algorithm described by Hoefler et al. [5]. The difference, however, is that instead of only determining the clock offset of each process relative to the *root*, HCA computes a linear model of the clock drift of each process.

The algorithm synchronizes clocks in two steps. In the first step, the clock drifts of processes with ranks smaller than the largest power of two $(0, \ldots, 2^{\lfloor \log_2 p \rfloor} - 1)$ are estimated in function `Sync_Clocks_Pow2` of Algorithm 2. Then, in the second step, the remaining processes with larger ranks ($\geq 2^{\lfloor \log_2 p \rfloor}$) compute their linear models of the clock drift with respect to the already synchronized processes in one additional round (cf. `Sync_Clocks_Remaining` function).

The major difference to the synchronization method found in Netgauge is the call to `Learn_Model_HCA`, which determines the model of the clock drift between two processes (Algorithm 4). The parameters N_FITPTS and N_EXCHANGES were introduced by Jones and Koenig. N_FITPTS specifies how many points (a *fitpoint* is a tuple containing a reference clock timestamp and a clock offset) will be recorded as input for the linear regression analysis. However, we only select a

[2]HCA stands for **H**unold and **C**arpen-**A**marie

**Algorithm 3** Measurement of the RTT between two nodes.

```
 1: function COMPUTE_RTT($p1$, $p2$)
 2:   $mean\_rtt = 0$
 3:   WARMUP_ROUNDS() // send dummy ping-pong messages
 4:   if $my\_rank == p1$ then
 5:     for $i$ in 0 to N_PINGPONGS $- 1$ do
 6:       MPI_RECV($tdummy$, 1, MPI_DOUBLE, $p2$)
 7:       $tremote$ = GET_ADJUSTED_TIME()
 8:       MPI_SEND($tremote$, 1, MPI_DOUBLE, $p2$)
 9:   else if $my\_rank == p2$ then
10:     for $i$ in 0 to N_PINGPONGS $- 1$ do
11:       $s\_time$ = GET_ADJUSTED_TIME()
12:       MPI_SEND($s\_time$, 1, MPI_DOUBLE, $p1$)
13:       MPI_RECV($tremote$, 1, MPI_DOUBLE, $p1$)
14:       $e\_time$ = GET_ADJUSTED_TIME()
15:       $l^{rtt}[i] = e\_time - s\_time$
16:     $l^{rtt}$ = REMOVE_OUTLIERS($l^{rtt}$)
17:     $mean\_rtt$ = MEAN($l^{rtt}$)
18:   return $mean\_rtt$
```

**Algorithm 4** Clock drift model for a pair of processes.

```
 1: function LEARN_MODEL_HCA(N_FITPTS, N_EXCHANGES,
                             $rtt$, $p1$, $p2$)
 2:   $slope = 0$, $intercept = 0$
 3:   if $my\_rank == p1$ then // process with reference clock
 4:     for $idx$ in 0 to N_FITPTS $- 1$ do
 5:       for $i$ in 0 to N_EXCHANGES $- 1$ do
 6:         MPI_RECV($tdummy$, 1, MPI_DOUBLE, $p2$)
 7:         $tremote$ = GET_ADJUSTED_TIME()
 8:         MPI_SEND($tremote$, 1, MPI_DOUBLE, $p2$)
 9:   else if $my\_rank == p2$ then // client process
10:     for $idx$ in 0 to N_FITPTS $- 1$ do
11:       for $i$ in 0 to N_EXCHANGES $- 1$ do
12:         MPI_SEND($tdummy$, 1, MPI_DOUBLE, $p1$)
13:         MPI_RECV($tremote$, 1, MPI_DOUBLE, $p1$)
14:         $local\_times[i]$ = GET_ADJUSTED_TIME()
15:         $l^{diff}[i] = local\_times[i] - tremote - rtt/2$
16:       $l^{diff}$ = SORT($l^{diff}$)
17:       $yfit[idx]$ = COMPUTE_MEDIAN($l^{diff}$)
18:       $idx\_median = i$ s.t. $(0 \leq i <$ N_EXCHANGES &
                              $l^{diff}[i] == yfit[idx])$
19:       $xfit[idx] = local\_times[idx\_median]$
20:     $slope$, $intercept$ = LINEAR_FIT($xfit$, $yfit$, N_FITPTS)
21:   return NEW_LM($slope$, $intercept$)

22: procedure COMPUTE_AND_SET_INTERCEPT($lm$, $p\_client$, $p\_ref$)
    // compute the intercept using the SKaMPI method
23:   if $r == p\_client$ then
24:     $diff$ = SKAMPI_PINGPONG($p\_client$, $p\_ref$)
25:     $diff\_timestamp$ = GET_ADJUSTED_TIME()
26:     $lm.intercept = lm.slope \cdot (-diff\_timestamp) + diff$
27:   else if $r == p\_ref$ then
28:     $diff$ = SKAMPI_PINGPONG($p\_client$, $p\_ref$)

29: function MERGE_LMS($lm1$, $lm2$)
30:   $new\_lm.intercept = lm1.intercept + lm2.intercept$
                         $- lm2.intercept \cdot lm1.slope$
31:   $new\_lm.slope = lm1.slope + lm2.slope - lm1.slope \cdot lm2.slope$
32:   return $new\_lm$
```

subset of all measurements (in total N_FITPTS many) for the linear regression, due to the high variance of the measured clock offsets. Thus, to determine each *fitpoint*, we perform a set of N_EXCHANGES ping-pongs between the two processes, and we select the median of these measurements. In addition, since the measured clock offsets need to be corrected by the RTT, we present our method for estimating the RTT in Algorithm 3. This RTT estimation will also be used to benchmark the algorithm of Jones and Koenig.

In the clock synchronization method found in Netgauge, intermediate clock offsets are summed up in a tree-like fashion to compute the offset of each process relative to the reference *root* node. Let us assume that we have three processes located on different hosts called $p_1$, $p_2$, and $p_3$, and each process has its own clock. If the clocks of hosts $p_1$ and $p_2$ have an offset of $diff_{p_1,p_2}$, and the clocks of $p_2$ and $p_3$ have an offset of $diff_{p_2,p_3}$, the clock offset between $p_1$ and $p_3$ can be computed as $diff_{p_1,p_3} = diff_{p_1,p_2} + diff_{p_2,p_3}$.

Therefore, we apply a similar transitive computation to combine linear regression models to obtain the clock drift between different processes. If the clock drifts are computed in one round for process pairs $(p_1, p_2)$ and $(p_2, p_3)$, the question becomes: how should these two linear models be combined such that $p_3$ can obtain its clock drift with respect to $p_1$?

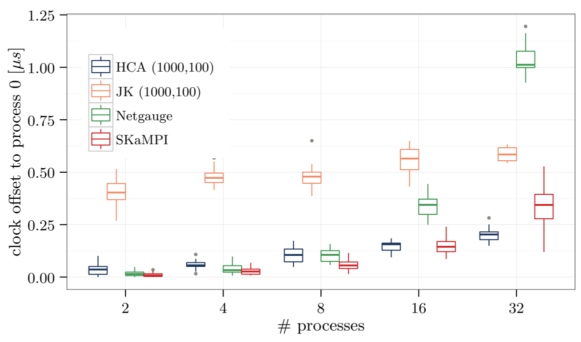
Let us denote the model of the clock drift of $p_2$ relative to $p_1$ as $t^{2\to1}(t_1) = t_1 - t_2 = s^{2\to1}t_1 + i^{2\to1}$, where the *slope* and the *intercept* of the model are defined as $s^{2\to1}$ and $i^{2\to1}$, respectively. Similarly, the clock drift of $p_3$ relative to $p_2$ is given as $t^{3\to2}(t_2) = t_2 - t_3 = s^{3\to2}t_2 + i^{3\to2}$. The computation of the clock drift between $p_3$ and $p_1$ is shown in Equation 1 and implemented in `Merge_LMs(lm1, lm2)` (cf. line 29 of Algorithm 4).

$$\begin{aligned} t^{3\to1}(t_1) &= s^{3\to1}t_1 + i^{3\to1} \\ &= t_1 - t_3 \\ &= s^{2\to1}t_1 + i^{2\to1} \\ &\quad + s^{3\to2}t_2 + i^{3\to2} \\ &= s^{2\to1}t_1 + i^{2\to1} \\ &\quad + s^{3\to2}(t_1 - s^{2\to1}t_1 - i^{2\to1}) + i^{3\to2} \\ &= t_1(s^{2\to1} + s^{3\to2} - s^{2\to1}s^{3\to2}) + i^{2\to1} \\ &\quad - s^{3\to2}i^{2\to1} + i^{3\to2} \end{aligned} \tag{1}$$

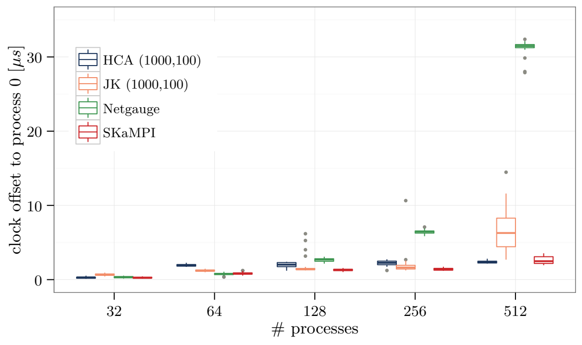
To estimate the error of the computed linear model of the clock offset as a function of time, we conducted a statistical analysis for each pair of processes. We performed an experiment to estimate the confidence intervals of both the slope and the intercept of the clock drift models. To do so, we measured 1000 fitpoints for each of 15 pairs of processes, running on different nodes, and computed the confidence intervals. For each pair, the length of the confidence interval of the slope was at most $2 \times 10^{-8}$, whereas the intercept computation revealed much wider confidence intervals, in the order of 100 ms. The consequence of these larger intervals is that the intercept computed with a linear regression analysis will decrease the accuracy of the initial clock offset with a high probability. As a consequence, the global clock error will increase over time.

To minimize the impact of the intercept error, we do not use the intercepts computed with a linear regression analysis. Instead, we have explored two approaches that appeared to be promising for computing the intercepts of the clock drift. Both approaches rely on SKaMPI's method for determining the clock offset between two processes at a given point in time (`SKaMPI_PingPong`). The intercepts can be obtained by measuring the clock offset between two processes and then using the already computed slope to find the intercept of the linear clock model (`Compute_And_Set_Intercept` of Algorithm 4). The reason why we have selected the SKaMPI method for computing the offset is that it provided us with the lowest initial clock offset values, as it will be shown in the first experiment in Section 7.

The *first approach* is to compute the intercepts in $\mathcal{O}(p)$ rounds after completing the hierarchical computation of the



(a) 1 process per node



(b) 16 processes per node

**Figure 8: Clock offset directly after synchronizing the processes (30 `mpiruns`, MVAPICH 2.1a, *TUWien*).**

clock models, which only requires $\mathcal{O}(\log p)$ rounds. We employ SKaMPI's clock synchronization to measure the clock offset between the *root* and each of the other $p-1$ processes as shown in function `Compute_And_Set_All_Intercepts`. The advantage is that the intercept is measured for each clock model separately. Thus, the intercept error only depends on the accuracy of a single SKaMPI synchronization and on the error of the slope, which was found to be very small ($10^{-8}$).

The *second approach* is to compute the intercepts during the hierarchical computation of the clock model in $\mathcal{O}(\log p)$ rounds. This algorithmic option is enabled by defining the global variable *hierarchical_intercepts*. In this case, we measure the clock offset and compute the new intercept by adjusting the offset using the clock model. Then, the intercept obtained from the linear regression is replaced with this new intercept. Here, the SKaMPI method is used to measure the clock offset for a pair of processes in each round. In order to compute the clock model between each process and the *root*, the linear models are combined hierarchically using Equation (1). The advantage of this method compared to the first approach is its better scalability. The downside is that relying on a combined intercept for the linear model increases the error of the logical global clock.

In addition, as large timestamps may affect the accuracy of the computed linear models, we adjusted all local timestamps of each process to an initial timestamp measured when the synchronization procedure is initiated.

We would like to point out that HCA should be considered as a general framework to synchronize clocks. In the present paper, we have used the method of Jones and Koenig to compute the clock drift model and SKaMPI's method to improve the accuracy of the model intercept. However, the concrete implementations of (1) how to obtain the linear model or (2) how to measure the clock offsets can easily be modified by substituting the functions `Learn_Model_HCA` and `Compute_And_Set_Intercept`, respectively.

## 7. EXPERIMENTAL EVALUATION

Now, we evaluate the different clock synchronization methods experimentally. The pseudocode of the experiments can be found in our technical report [6]. To compare the synchronization schemes of SKaMPI and Netgauge (NBCBench) to the competitors, we have extracted the relevant clock synchronization algorithms from their respective benchmarking frameworks. In particular, we use a fixed window size and disable the dynamic window adaptation as done by SKaMPI and NBCBench. This allows for a fairer analysis of the clock drifts. Furthermore, we rely on scheme (4) of Figure 2 for process synchronization, and we compute the run-times of MPI calls by applying the global time method described in Section 4. The tuple (N_FITPTS, N_EXCHANGES) used by HCA and JK is specified in each figure.

In the following experiments, we show results obtained with the *first approach* of HCA, i.e., we use the hierarchical way of computing the slopes and the linear way of obtaining the intercepts ($\mathcal{O}(\log p) + \mathcal{O}(p)$ rounds). In practice, the estimation of the drift slope using linear regression typically requires many more ping-pong messages than the offset computation with SKaMPI for a pair of processes. Thus, $p-1$ SKaMPI rounds can be much shorter than $\mathcal{O}(\log p)$ rounds of the hierarchical slope computation. In our experimental setting (e.g., number of processes), a simple analytic model revealed that the *first approach* of HCA does not incur a significant run-time overhead compared to the *second approach*, since the time for obtaining the values for the linear regression is dominating.

In our first experiment we apply each of the previously described synchronization methods to obtain a global clock for every process. Then, we measure the clock offset between the *root* process and each of the other processes *directly after* the synchronization phase has been completed. To that end, the *root* process exchanges a number of ping-pong messages with all other processes and estimates its clock offset relative to the global time computed on each process.

Figure 8(a) presents the maximum clock offset measured between any process and the reference process directly after finishing the clock synchronization. We used one MPI process per compute node in this experiment. Let $diff^{j}_{r,root}$ be the clock offset between processes $r$ and *root* in ping-pong round $j$ (in total $nrounds = 10$). The maximum clock offset is computed as $\text{MAX}_{0\leq r<p}(\text{MIN}_{0\leq j<nrounds}(diff^{j}_{r,root}))$ for each synchronization algorithm. Each experiment was repeated 30 times (different calls to `mpirun`). The graph shows that the clock offset measured directly after synchronizing the clocks with SKaMPI or Netgauge is very small, i.e., we measured an offset of at most 0.2 µs for up to 8 different compute nodes. However, the method of Netgauge leads to significantly larger offsets as the number of processes (and therefore

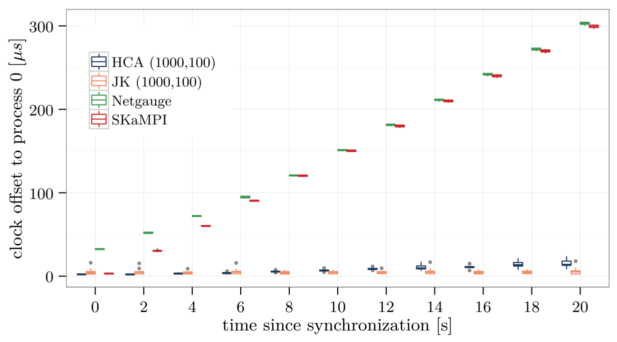


**Figure 9: Clock drift for** 512 ($32 \times 16$) **processes after** 0, 2, 4, . . ., 20 s **(distribution of maximum offsets over** 10 **calls to** `mpirun`**, MVAPICH 2.1a,** ***TUWien*****).**

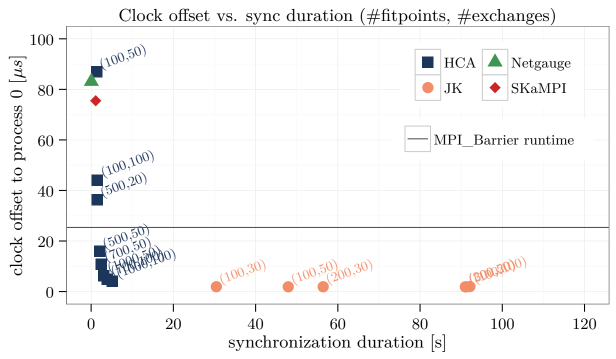


**Figure 10: Median clock offset (after** 5 s**) vs. synchronization duration for** 512 ($32 \times 16$) **processes (**10 **calls to** `mpirun`**, MVAPICH 2.1a,** ***TUWien*****).**

the number of synchronization rounds) increases. The JK synchronization method produces slightly larger clock offsets for a small number of processes (2–16) compared to SKaMPI and Netgauge, due to the inaccuracy of their approach for computing linear models.

We also checked how the HCA algorithm compares to the other clock synchronization methods. Figure 8(a) shows that for up to 32 processes (1 process per node), HCA results in a maximum clock offset that is similar to the offset yielded by the SKaMPI method. Furthermore, in the particular case of 32 processes, the maximum clock offset lies around 0.25 µs, which represents an improvement over all other methods. However, HCA-based clock offsets show an increasing trend with the number of processes, which is a consequence of the need to combine linear models hierarchically.

The picture does not change for larger numbers of processes, as shown in Figure 8(b). Here, SKaMPI still synchronizes the distributed clocks with the highest precision, but the relative difference to JK is smaller. Netgauge, in contrast, will lead to the least synchronized clocks among its competitors for 256 or more processes on our machine, due to its hierarchical way of combining the computed offsets. The HCA method appears to be a viable alternative to SKaMPI, as it results in clock offsets in the same order of magnitude.

However, it is important to recall that real clocks are drifting apart, as shown in Figure 6. To evaluate the synchronization methods in this scenario, we performed another experiment, in which we measured the clock offset over time (clock drift). The *root* process waits in a loop for a given amount of time (e.g., 1 s) and then measures its clock offset to all other processes. In this way, we can determine how much the logical global time is drifting on each process. Figure 9 presents the clock drift measured for 512 processes on our 36 node cluster (*TUWien*). We see that the clock synchronization methods that account for the clock drift (JK and HCA) are largely superior to the ones that only compute the initial clock offset to the reference clock (Netgauge and SKaMPI). While these results suggest that the method of Jones and Koenig leads to the most precise measurements for long execution times, its synchronization mechanism is slow, as it serializes the computation of linear models. We are therefore interested in understanding the trade-off between the most accurate clock offset that is obtainable and the

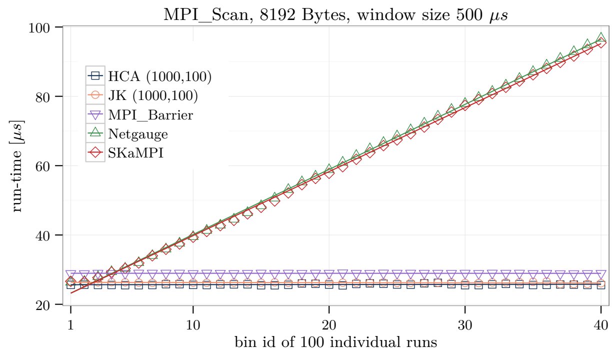


**Figure 11: Run-time of** `MPI_Scan` **(median of** 3 **calls to** `mpirun`**,** 8192 Bytes, $16 \times 1$ **processes,** 4000 **runs, bin size:** 100**, Intel MPI 4.1,** ***Cartesius*****).**

time it takes to synchronize the processes.

Figure 10 shows the Pareto frontier of the clock offset versus the synchronization time, which visualizes the possible configuration choices. We also added the mean time to complete a call to `MPI_Barrier` as a baseline. It provides an insight on the magnitude of process imbalance when synchronizing measurements through `MPI_Barrier` calls and a limit to the clock offset that is acceptable for window-based synchronization methods to prove useful in benchmarking contexts. The figure plots the clock offsets that were measured five seconds after completing each clock synchronization. We see that the clock offsets of Netgauge and SKaMPI are relatively large (≈ 80 µs), but both need less than one second to complete. In contrast, the time to complete the clock synchronization phases of JK and HCA depends on the number of ping-pong messages needed to compute the regression models. Thus, the parameters *number of fitpoints* and *number of exchanges* have a strong influence on the quality of the clock synchronization. Figure 10 indicates that HCA is able to synchronize the clocks with a higher precision than what `MPI_Barrier` can provide, while only requiring approximately 5 s to finish the synchronization process. The method of Jones and Koenig, on the other hand, produces even smaller clock offsets, but requires at least 30 s (in the (100, 30) case) to complete.

Now, we would like to know how the different clock synchronization algorithms influence the resulting MPI measurements. Figure 11 compares the resulting run-times of `MPI_Scan` with a message size of 8192 Bytes obtained with different synchronization methods. The run-times are computed as medians over three experiments, in which, for each experiment, we computed the mean of bins of 100 consecutive measurements. Even though we only show the measurement results on *Cartesius*, we obtained similar results for experiments conducted with other MPI functions, message sizes, and on different machines. We can observe that a window-based approach might improve the accuracy of the execution time of MPI functions compared to synchronizing using `MPI_Barrier`. For example, in Figure 11, the run-times measured with Netgauge, JK, or HCA are initially smaller than the ones measured when `MPI_Barrier` is used to synchronize processes. However, as explained before, the run-times obtained with SKaMPI and Netgauge are quickly drifting in time. In contrast, the HCA synchronization algorithm leads to stable measured run-times, suggesting it can be a reliable tool for benchmarking MPI functions.

## 8. CONCLUSIONS

We have shown that the choice of the clock and process synchronization methods used for MPI benchmarking has tremendous effects on the outcome. The clock synchronization method implemented by SKaMPI can achieve very accurate timings, but since the logical global clocks are drifting quickly, only a small number of MPI operations can be measured precisely, which of course depends on the duration of an MPI function call. In case the experimenter wants to measure the run-time of MPI functions over a longer period of time (e.g., several milliseconds or even seconds), the approaches used in SKaMPI and Netgauge will most likely lead to inaccurate measurements. To overcome this problem, one could start re-synchronizing the clocks after a given amount of time has passed or use a clock synchronization algorithm that accounts for the clock drift.

The clock synchronization method of Jones and Koenig accurately synchronizes a set of distributed clocks and also considers the clock drift between processes. This approach could be used if very accurate window-based measurements are required and if the relatively long time for completing the clock synchronization phase can be tolerated.

Our novel clock synchronization method, called HCA, can be seen as a trade-off between achieving accurate results for longer measurements (like the JK method) and being scalable (like Netgauge). Yet, it suffers from the same problem as Netgauge, as it combines models with an inherent experimental error. Nevertheless, in our MPI benchmarking setup the HCA algorithm emerged as the best option for process synchronization compared to `MPI_Barrier`, SKaMPI, or Netgauge when measurements over longer periods of time (we have tested for up to 20 s) for many processes are needed.

Last, we note that using a library-provided implementation of `MPI_Barrier` may lead to unforeseeable results, as processes can become significantly skewed when they leave `MPI_Barrier`. The decision whether to rely on `MPI_Barrier` should therefore be done after investigating the behavior of the implementation on the given network. Nevertheless, an MPI benchmark should provide its own `MPI_Barrier` implementation for fairly comparing two MPI libraries.

## Acknowledgments

We thank Alf Gerisch (TU Darmstadt) for discussions and comments, and Thomas Geenen (SURFsara) for the access to the Dutch national supercomputer Cartesius.

## 9. REFERENCES

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

[2] 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.

[3] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of Supercomputing 2007*, pages 1–10, 2007.

[4] T. Hoefler, T. Schneider, and A. Lumsdaine. Accurately measuring collective operations at massive scale. In *Proceedings of the IPDPS Workshops*, 2008.

[5] T. Hoefler, T. Schneider, and A. Lumsdaine. Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale. *Int. J. of Parallel, Emergent and Distributed Systems*, 25(4):241–258, 2010.

[6] S. Hunold and A. Carpen-Amarie. MPI benchmarking revisited: Experimental design and reproducibility. *CoRR*, abs/1505.07734, 2015.

[7] S. Hunold, A. Carpen-Amarie, and J. L. Träff. Reproducible MPI micro-benchmarking isn't as easy as you think. In *EuroMPI/ASIA '14*, pages 69–76, 2014.

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

[9] T. Jones and G. A. Koenig. Clock synchronization in high-end computing environments: a strategy for minimizing clock variance at runtime. *Concurr. and Comput.: Practice and Experience*, 25(6):881–897, 2012.

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

[11] 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.

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

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

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

[15] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Prentice-Hall, USA, 2006.

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

[17] T. Worsch, R. Reussner, and W. Augustin. On benchmarking collective MPI operations. In *Euro PVM/MPI 2002*, page 271–279, 2002.