Hierarchical Clock Synchronization in MPI

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

Alexandra Carpen-Amarie
Fraunhofer ITWM
Kaiserslautern, Germany
Email: alexandra.carpen-amarie@itwm.fraunhofer.de

***Abstract*—MPI benchmarks are used for analyzing or tuning the performance of MPI libraries. Generally, every MPI library should be adjusted to the given parallel machine, especially on supercomputers. System operators can define which algorithm should be selected for a specific MPI operation, and this decision which algorithm to select is usually made after analyzing benchmark results. The problem is that the latency of communication operations in MPI is very sensitive to the chosen data acquisition and data processing method. For that reason, depending on how the performance is measured, system operators may end up with a completely different MPI library setup.**

In the present work, we focus on the problem of precisely measuring the latency of collective operations, in particular, for small payloads, where external experimental factors play a significant role. We present a novel clock synchronization algorithm, which exploits the hierarchical architecture of compute clusters, and we show that it outperforms previous approaches, both in run-time and in precision. We also propose a different scheme to obtain precise MPI run-time measurements (called Round-Time), which is based on given, fixed time slices, as opposed to the traditional way of measuring for a predefined number of repetitions. We also highlight that the use of `MPI_Barrier` has a significant effect on experimentally determined latency values of MPI collectives. We argue that `MPI_Barrier` should be avoided if the average run-time of the barrier function is in the same order of magnitude as the run-time of the MPI function to be measured.

***Index Terms*—MPI, clock synchronization, collective communication operations, benchmarking, barrier algorithms**

I. Introduction

Measuring time correctly is one of the fundamentals of computer science, as time measurements help us to analyze the internal performance behavior of applications. Performance analysis techniques used in High Performance Computing (HPC), such as profiling or tracing, are often based on timestamps. The problem is that in many distributed systems, like the supercomputers used in HPC, the compute nodes have their own local clock, and commonly, these clocks are periodically corrected by using protocols such as NTP or RADclock [1]. The precision of such clocks is often good enough for coarse-grained time measurements, e.g., when a resolution of seconds or tens of seconds is required. However, for measuring very short events (several milli- or microseconds), these clocks fail to provide the desired level of precision, as distributed clocks are drifting apart over time. The drifting error may be tolerable or negligible for longer-lasting events (multiple seconds), but it has a tremendous impact on short observations (microseconds). In the context of MPI, precise clocks are required to study performance artifacts, e.g., when inspecting or analyzing event traces with tools like Tau [2] or Scalasca [3]. In addition, a precise global clock can help to benchmark MPI communication operations in a fair manner, which will be discussed below.

In the present paper, we primarily focus on the problem of obtaining a precise logical, global clock that can be used for benchmarking MPI collectives. Initially, our research on precise global clocks was driven by the need for accurate measurements when automatically tuning MPI functions in the context of PGMPITuneLib [4]. This library empirically evaluates the latency of a specific MPI operation and several semantically equal replacement algorithms, for which MPI performance guidelines have been defined [5], [6]. In many MPI applications, e.g., in the DOE Mini-apps [7], the message (buffer) size used in collectives like `MPI_Allreduce` and `MPI_Alltoall` is rather small, often in the range of $8\,\mathrm{B}$ to $1024\,\mathrm{B}$. A problem arises if one attempts to automatically tune collectives (find the shortest latency) for small message sizes, especially if the latency of the `MPI_Barrier` call is in the same order of magnitude as the latency of the MPI function to be optimized. In such cases, the use of `MPI_Barrier` for synchronizing processes—with the goal of starting to measure the latency on all processes simultaneously—can have a significant impact on the performance results, as participating processes may exit the barrier at different times [8], [9]. One approach to tackle this problem is to apply logical, global clocks to coordinate processes based on time slices (or windows) rather than by using `MPI_Barrier`. In previous work, we have shown how to obtain a logical, global clock in a scalable manner [10]. Now, we build upon previous works and make the following contributions:

1) We propose a new clock synchronization algorithm called HCA3, which can be used to obtain a logical, global clock in $\mathcal{O}(\log p)$ rounds and which improves precision compared to previous approaches.
2) We propose a hierarchical framework (or scheme) for clock synchronization called H^lHCA, where a different clock synchronization algorithm can be applied at each architectural level of the machine, e.g., compute nodes, sockets, or cores.
3) We introduce a novel benchmarking scheme called Round-Time, which solves two fundamental problems of the barrier- or the window-based measurement scheme: (1) the impact of imbalanced processes when benchmarking

(when using MPI_Barrier) and (2) the over- or under-estimation of window sizes, respectively (when using a window-based scheme).

The remainder of this article is organized as follows. In Section II, we discuss related work. In Section III, we introduce our novel clock synchronization algorithm called HCA3 and present experimental results to provide evidence that the new algorithm improves clock accuracy. We then show in Section IV how to combine different building blocks of clock synchronization algorithms in a hierarchical manner to form a new clock synchronization scheme. Finally, we demonstrate the usefulness of having a precise logical, global clock in several case studies in Section V, before we conclude and discuss our findings in Section VI.

II. Related Work and Background

We now discuss several approaches to two interrelated problems, which are (1) performing MPI benchmarking and (2) synchronizing distributed clocks in MPI. Before we continue, we need to introduce some terminology. The *clock offset* is the absolute difference between two clocks at a specific point in time. The *clock skew* denotes the difference in frequencies of time sources, and the *clock drift* is the difference between two clocks over a period of time.

MPI benchmarking aims at giving hints to developers or system operators on how well a given MPI library on a specific parallel machine performs. In a typical benchmark run, a particular MPI function, e.g., MPI_Reduce, is executed for a predefined number of times and the latency of each call is measured. Since the measurement can be done in various ways, we refer the reader to [11], [12] for a comprehensive overview. A large number of research articles show benchmark results that were obtained with either the Intel MPI Benchmarks [13] or the OSU Micro-Benchmarks [14]. Both benchmark suites use a barrier-based process synchronization scheme when measuring the latency of MPI collective operations, where processes are re-synchronized by calling MPI_Barrier after each call to an MPI function, whose latency should be measured. Worsch et al. [8] and Hoefler et al. [9] already pointed out that using MPI_Barrier for process synchronization may distort benchmarking results. Therefore, SKaMPI [8], [15] and NBCBench [9] can be configured to switch to a window-based measurement scheme. This scheme is based on a global clock, where processes negotiate common points in time when to start measuring the latency of a specific MPI function call. Thus, when one of these starting points has been reached (the beginning of a measurement window), all processes collectively start measuring. Unfortunately, many HPC machines do not own a global time source, and therefore, both SKaMPI and NBCBench provide implementations of different clock synchronization algorithms. The problem is that the clock models used in SKaMPI and NBCBench do not account for the clock drift, and thus, the precision of the logical, global clock quickly degrades over time.

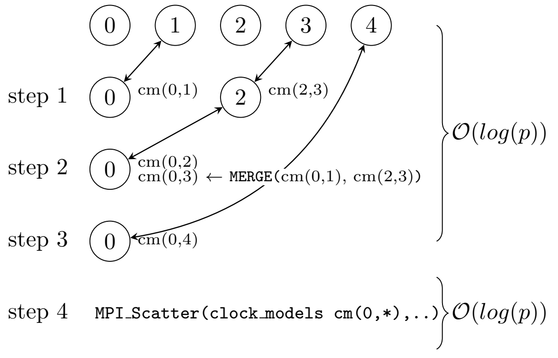
To solve that problem, we introduced the algorithms HCA and HCA2 [10] in the context of the ReproMPI [12] benchmarking suite. Both algorithms are extensions of concepts proposed by Hoefler et al. [9] and by Jones and Koenig [16]. The clock synchronization algorithm JK of Jones and Koenig [16] computes a clock drift model, but requires $\mathcal{O}(p)$ rounds to complete the clock synchronization, which makes it relatively slow compared to other approaches. To improve the speed, HCA and HCA2 use an inverted binomial tree when learning clock drift models, which is structurally similar to the approach of Hoefler et al. [9]. Both algorithms, HCA and HCA2, compute a global clock model by exchanging ping-pong messages that piggyback timestamps and by learning a linear regression model based on these timestamps. The difference between HCA and HCA2 lies in the last step, in which HCA performs an additional round of adjusting the clock offsets (the intercept in the linear model) between the time source and each of the other processes. Although this additional step makes this algorithm technically an $\mathcal{O}(p)$ algorithm, it is still often fast enough in practice [10]. In contrast, HCA2 does not re-compute the clock offset once the linear regression has been completed, and therefore works in $\mathcal{O}(\log p)$ steps.

With such a global clock, one can apply a window-based scheme, which allows for obtaining more accurate latency measurements of MPI operations than simply synchronizing using MPI_Barrier, in particular, if the latency of MPI_Barrier is of similar magnitude as the latency of the MPI operation under investigation. The problem with window-based measurements is twofold: first, one needs a relatively good estimate of the latency of an MPI operation, in order to determine the window size. Second, one outlier (a measurement that is much larger than the predefined window size) can cause a large number of subsequent measurements to be invalidated (as processes will miss the starting time of several subsequent windows). We will show how to overcome these disadvantages in Section V.

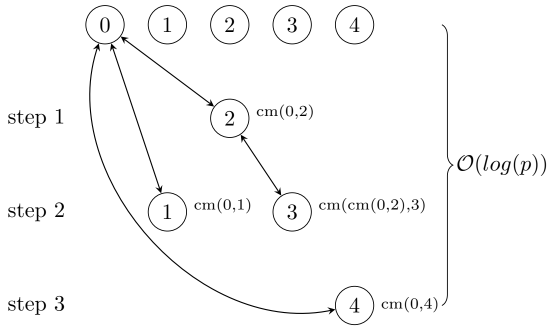
Logical, global clocks are not only important for MPI benchmarking, but also for MPI performance analysis. Trace analysis tools like Scalasca use linear interpolation to adjust timestamps. This is usually done by considering the clock drift measured between the initialization and the finalization phase of an MPI application. Here, the assumption is made that the clock drift is linear over time, which is not always true [16], [17]. Jones et al. [16] also point out that collocated applications may experience performance degradation if not scheduled coordinately, which can be avoided by using an accurate global clock.

III. Clock Synchronization Algorithm: HCA3

The idea to improve upon HCA2 was sparked by the PulseSync algorithm, which was proposed by Lenzen et al. [18] for the purpose of synchronizing clocks in wireless networks. PulseSync tries to spread the information about the reference clock as fast as possible on a network. To keep the number of hops small, it uses a breadth-first search to send "pulses" through the network. Each client node in the network estimates the offset and the drift of its clock relative to the reference clock after receiving a pulse and then forwards a timestamp as



(a) HCA2



(b) HCA3

Fig. 1: Illustration of the algorithmic structure of HCA2 in comparison to the novel HCA3 algorithm.

quickly as possible. The interesting fact is that each client node uses its global clock model when forwarding a new timestamp. It is important to note that timestamping in this approach is done at the MAC-layer to reduce jitter in clock timestamps. In our case, when trying to devise a clock synchronization algorithm on top of MPI, we do not have access to the MAC-layer. Thus, we will adapt ideas of PulseSync for our context.

A. Building Blocks: Clock Offset Algorithms

Let us first introduce algorithms to determine the current clock offset between a pair of processes, which will be used as building blocks in the new clock synchronization algorithm. The first algorithm that can be used as a building block is the clock offset algorithm called Mean-RTT-Offset, which was proposed by Jones and Koenig (cf. Algorithm 8 in Appendix A and [16, Figure 1]). This clock offset algorithm uses estimates of the round-trip time (RTT) between a client process and the master process owning the reference clock. The problem with this approach is that it needs an estimate of the RTT, which is another potential source of error. Another observation is that Mean-RTT-Offset relies on averages, as it determines the mean RTT between pairs of processes and computes the clock offset as the median of several clock offset measurements.

In contrast to the Mean-RTT-Offset algorithm, the clock synchronization algorithm implemented in SKaMPI uses a different method to estimate the clock offset, which is based on minima and does not rely on RTT estimates. The clock offset algorithm of SKaMPI (which we call SKaMPI-Offset) is sketched in Algorithm 7 in Appendix A (cf. [8, Algorithm 1]). In essence, it tries to minimize the time for sending and receiving timestamps between a reference and a client process, and it uses these minima to compute an estimate of the current clock offset. Using the minimum round-trip time seems to be a good choice, which was already pointed out by Ridoux and Veitch [1]: “If a timing packet is lucky enough to experience the minimum delay, then its timestamps have not been corrupted and can be used to set the absolute clock directly to the right value (asymmetry aside).”

In summary, each of our clock synchronization algorithms (that computes a global clock model using linear regression analysis) presented in this work is parameterized with a clock offset algorithm. We have implemented and tested the following two clock offset algorithms:

1) SKaMPI-Offset (trying to find minimum RTTs) and
2) Mean-RTT-Offset (relying on mean RTTs and medians of clock offsets).

B. Algorithm: HCA3

The pseudo code of HCA3 is shown in Algorithm 1. For more details about previous clock synchronization algorithms, we refer the reader to one of our technical reports [11]. HCA3 and its predecessor HCA2 rely on building and merging linear models of a reference clock in a logarithmic number of rounds. Sketches of both approaches are illustrated in Figure 1. On the left-hand side in Figure 1a, we can observe that HCA2 learns clock models towards the root (rank 0 in the example) using an inverted binomial tree. In the first step, two clock models are computed, one between ranks 0 and 1 denoted as $cm(0,1)$ and another between ranks 2 and 3 denoted as $cm(2,3)$. In step 2, a clock model between rank 0 and rank 2 is computed, and subsequently rank 0 can determine a clock model to rank 3, by merging the model $cm(0,2)$ and $cm(2,3)$.[1] After $\mathcal{O}(\log p)$ rounds, the root process (reference time source) has obtained a clock model to all other processes included in the given MPI communicator (the set of participating processes). Then, the root processes distributes the clock models to all processes by calling `MPI_Scatter`.

In contrast to the HCA2 scheme, HCA3 starts at the root and tries to push its reference time down the binomial tree. In our example shown in Figure 1b, process 2 computes its global clock model in the first step by exchanging messages with rank 0, and it will use this model to emulate the global reference clock in the upcoming, second step. Thus, in step 2 of the clock synchronization, ranks 2 and 3 exchange timestamps, but the process with rank 2 already employs its global clock model when timestamping messages.

We now take a brief look at Algorithm 1 and focus on the most important part, which is found between lines 5 and 15. In each round, a process can either be a client or a reference

[1]How to merge linear clock models was presented before [10].

Algorithm 1 HCA3 clock synchronization algorithm

```
nprocs - number of processes
r - current process rank (0 to nprocs − 1)
lm - linear model (defined by a slope and an intercept) of the clock drift of the
    current process to adjust the local clock relative to a reference clock
clk - base clock that provides a local time on each process

function SYNC_CLOCKS(comm, clk)
    nrounds ← $\lfloor log_2(nprocs) \rfloor$
    max_power ← $2^{nrounds}$
    my_clk ← new GLOBALCLOCKLM(clk, 0, 0) // default dummy clock
    // Step 1:compute linear models of the clock drifts for processes with
    // indices between 0 and (max_power − 1)
    for $i$ in nrounds to 1 do
        running_power ← $2^i$
        next_power ← $2^{i-1}$
        if $r \geq$ max_power then break
        if ($r$ mod running_power) == 0 then        // process with reference clock
            other_rank ← $r$ + next_power
            lm ← LEARN_CLOCK_MODEL(comm, r, other_rank, my_clk)
        else if ($r$ mod running_power) == next_power then        // client process
            other_rank ← $r$ − next_power
            lm ← LEARN_CLOCK_MODEL(comm, other_rank, r, my_clk)
            my_clk ← new GLOBALCLOCKLM(clk, lm)
    // Step 2: compute linear models of the clock drifts for processes with
    // indices between max_power and (nprocs − 1)
    if $r \geq$ max_power then        // client process
        other_rank ← $r$ − max_power
        lm ← LEARN_CLOCK_MODEL(comm, other_rank, r, my_clk)
        my_clk ← new GLOBALCLOCKLM(clk, lm)
    else if my_rank < (nprocs − max_power) then // process with reference clock
        other_rank ← $r$ + max_power
        lm ← LEARN_CLOCK_MODEL(comm, r, other_rank, my_clk)
    return my_clk
```

process. However, each process will be a client only once, but it can be a reference process for all upcoming rounds. Therefore, in line 9 of Algorithm 1, each process checks whether it will act as a client or a reference in this round. If a process takes part in the current algorithmic round, it executes LEARN_CLOCK_MODEL and obtains a clock model with respect to its communication partner. This procedure is repeated until all processes have obtained a global clock model.

In line 4, we initialize the current clock with GLOBAL-CLOCKLM(*clk*, 0, 0), which simply represents the input clock without adjusting the drift and the offset. This is only done for symmetry reasons, as some processes may not take part in a round, and in this way they can return a clock of the same datatype. We also note that the current base clock (*clk*) is a parameter of SYNC_CLOCKS. This base clock can be a regular, local clock like `MPI_Wtime` or a logical, global clock, and this design enables HCA3 to be coupled with other synchronization algorithms (cf. Section IV).

The algorithm LEARN_CLOCK_MODEL is relatively straightforward. First, it obtains *nfitpoints* many fit points using one of the clock offset algorithms that were introduced as building blocks. A fit point is characterized by the client's clock offset to the reference clock and the timestamp when the clock offset was measured. Second, the algorithm computes an estimate of the clock skew (slope) and the clock offset (intercept) for a pair of processes using a linear regression analysis of these *nfitpoints* many data pairs. To improve the accuracy of the linear models, we also added the possibility to recompute the clock offset (intercept) between two clocks after completing each linear regression. This additional re-computation can be enabled or disabled using the *recompute_intercept* flag.

Algorithm 2 Clock drift model for a pair of processes

```
o_alg - clock offset algorithm (global variable)
recompute_intercept - boolean flags (global variable)

function LEARN_CLOCK_MODEL(comm, p_ref, other_rank, clk)
    slope ← 0, intercept ← 0
    if my_rank == p_ref then                    // process with reference clock
        for idx in 0 to nfitpoints − 1 do
            o_obj ← o_alg→MEASURE_OFFSET(comm, clk, p_ref, other_rank)
        if recompute_intercept == 1 then
            COMPUTE_AND_SET_INTERCEPT(comm, NULL, clk, p_ref, other_rank)
    else if my_rank == other_rank then                    // client process
        for idx in 0 to nfitpoints − 1 do
            o_obj ← o_alg→MEASURE_OFFSET(comm, clk, p_ref, other_rank)
            xfit[idx] ← o_obj→GET_TIMESTAMP()
            yfit[idx] ← o_obj→GET_OFFSET()
        lm ← FIT_LINEAR_MODEL(xfit, yfit, nfitpoints)
        if recompute_intercept == 1 then
            COMPUTE_AND_SET_INTERCEPT(comm, lm, clk, p_ref, other_rank)
    return lm

procedure COMPUTE_AND_SET_INTERCEPT(comm, lm, clk, p_ref, client)
    // compute the intercept by measuring the current offset to the reference clock
    // r is rank of process calling this function
    if r == client then
        o_obj ← o_alg→MEASURE_OFFSET(comm, clk, p_ref, client)
        timestamp ← o_obj→GET_TIMESTAMP()
        lm→intercept ← lm→slope · (−timestamp) + o_obj→GET_OFFSET()
    else if r == p_ref then
        o_obj ← o_alg→MEASURE_OFFSET(comm, clk, p_ref, client)
```

TABLE I: Parallel machines used in our experiments.

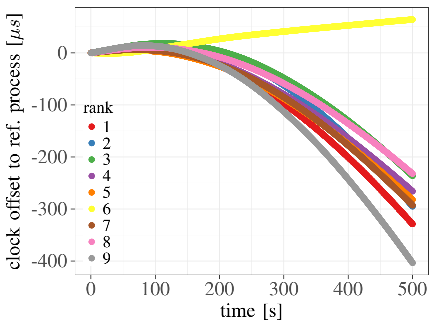
Name	Hardware	MPI Libraries	Compiler
Jupiter	36 × Dual Opteron 6134 @ 2.3 GHz, InfiniBand QDR	Open MPI 3.1.0	gcc 6.3.1
Hydra	36 × Dual Intel Xeon Gold 6130 @ 2.1 GHz, Intel OmniPath	Open MPI 3.1.0	gcc 6.3.0
Titan	Cray XK7, Opteron 6274 @ 2.2 GHz, Cray Gemini	cray-mpich/7.6.3	gcc 4.9.3

C. Experimental Evaluation of HCA3

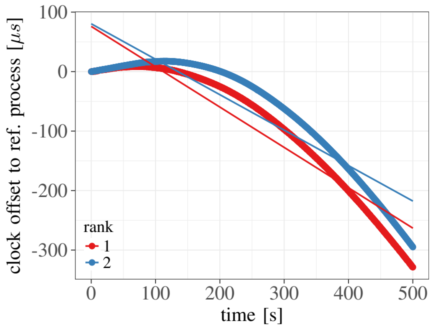
Before introducing other algorithmic clock synchronization schemes, we want to show that HCA3 indeed improves the quality of the logical, global clock.

1) Environment and Hardware: Our experimental environments are summarized in Table I. The compute nodes of *Jupiter* and *Titan* are similar, as they are equipped with two AMD processors, where each processor has 8 cores. The compute nodes of *Hydra* comprise two Intel Xeon Scalable processors, where each processor has 16 cores. We use three types of interconnects: InfiniBand, Intel OmniPath, and Cray Gemini.

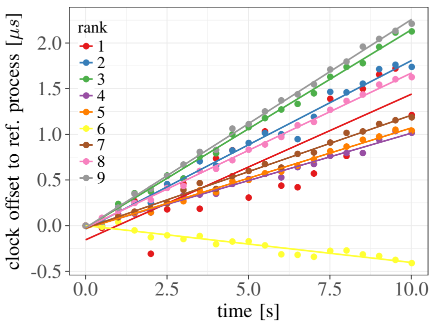
2) Validating Assumptions: A typical assumption underlying the clock synchronization algorithms accounting for the clock drift is that this clock drift is linear over time. Doleschal et al. [19] have already pinpointed that the clock drift is not constant over a longer period of time, and therefore, if MPI tracing tools want to exploit global timestamps then they have to re-synchronize clocks periodically. For that reason, we wanted to know the time period for which the linearity assumption is reasonable. A snapshot of the results is shown in Figure 2a, which demonstrates the clock drift of nine processes with respect to a single process acting as the time source over a period of 500 s. In this experiment, we only use one rank per compute node, which is pinned to the first core of a compute



(a) Clock drift over 500 s



(b) Clock drift over 500 s for 2 processes



(c) Clock drift over 10 s

Fig. 2: Clock offset between reference process and other MPI ranks over a fixed period of time; Open MPI 3.1.0, *Hydra*.

node of *Hydra*. We show the fitted linear regression models in Figure 2b for only two processes (for clarity reasons). It can be observed that the linearity assumption does not hold for such a long period of time. However, if we consider a shorter time frame, e.g., the first 10 seconds as in Figure 2c, then the drift can be considered linear (the R^2 values are usually higher than 0.9). We have seen a similar behavior of the clocks on the other machines.

This analysis gives us boundaries for which a global clock model will be accurate, which is a range of 0 s to 20 s. After one minute, the accuracy of the linear model goes down significantly. However, a time period of 0 s to 20 s is well suited for MPI benchmarking, where clocks are synchronized in the first few seconds and then the latency of MPI collectives is measured repeatedly, and one measurement only takes several micro- or milliseconds to complete if the payload is small.

3) Results: We have intensively tested various combinations of clock synchronization algorithms with different parameter settings. There are three main parameters to each clock synchronization algorithm:

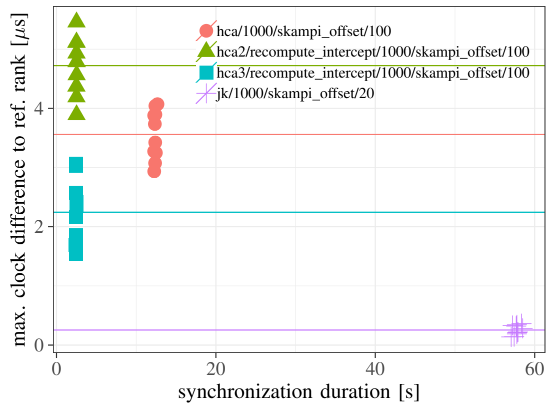
(a) the number of fit points (*nfitpoints*) that are used for the linear regression analysis;
(b) the clock offset algorithm (cf. Section III-A);
(c) and the number of ping-pongs to be done inside a clock offset algorithm.

In the comparison of the various clock synchronization algorithms, we use different labels to denote the algorithm and its parameter settings. For example, the label "hca/1000/SKaMPI-Offset/100" denotes that clock synchronization algorithm HCA was chosen, 1000 fit points were obtained with SKaMPI-Offset, and 100 ping-pongs were executed within SKaMPI-Offset to find each individual fit point.

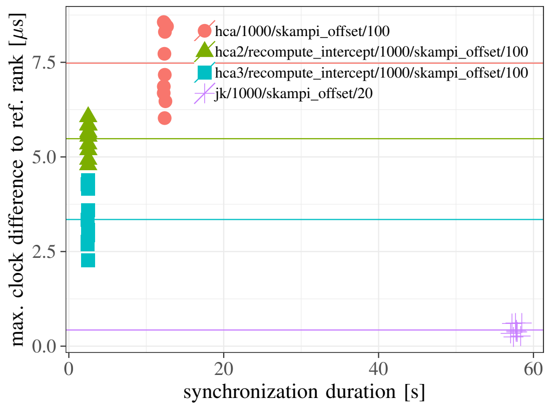
In our experiments, we first synchronize clocks using one of the clock synchronization algorithms and then perform a series of offset measurements between the root (reference) process and the other processes, as shown in Algorithm 6 (see appendix). These offset measurements give an indication of the absolute clock offset between the global clocks of two processes (which is an accuracy measure of the global clock model). After the clock offset to a reference clock has been computed for each client, we wait for a given amount of time, e.g., 10 s, and evaluate the accuracy of the global clock again. Usually clocks are very well synchronized directly after the synchronization algorithm has completed, but the global clocks drift off when time passes. This behavior is clearly visible in Figure 3 (here we only use the best parameter setting that was empirically found for each clock synchronization algorithm). Besides the accuracy of the logical, global clock, the time needed to perform the clock synchronization is often a conflicting goal. The algorithm JK, for example, synchronizes each client with the reference process, and thus requires $\mathcal{O}(p)$ synchronization rounds. We have shown before that HCA2 needs only $\mathcal{O}(\log p)$ rounds to complete the clock synchronization [10]. Similarly, also HCA3 works in $\mathcal{O}(\log p)$ steps. This smaller asymptotic complexity should be and indeed is empirically visible in Figure 3, in which we relate the synchronization duration to the synchronization accuracy. Each individual point represents the maximum clock offset after one call to `mpirun`. We repeated the experiment *nmpiruns* $= 10$ times, and thus, ten points per synchronization algorithm are plotted. Since the two goals, clock accuracy and synchronization duration, are conflicting, we want to find algorithms in the lower left corner, i.e., algorithms that are fast and produce accurate clock models.

Right after the clock synchronization has ended (0 s), the global clocks obtained from all algorithms are still very precise, as the maximum clock offset between each process and the time source was determined to be about 4 µs in the worst case. But as time passed, the global clock models created by HCA and HCA2 started to drift apart. When comparing the different HCA versions, we can observe that HCA3 indeed improves precision compared to its predecessors.

We also want to comment on the results of JK, which was found to be the most accurate for this configuration (512 processes) on *Jupiter*. However, JK suffers severely when the number of processes increases and the clock drift between processes changes rather quickly. On *Hydra*, for example, the JK algorithm was by far the worst for all considered combinations, as it simply suffered from its longer running time. We also noticed that it was often better to employ SKaMPI-Offset inside JK instead of the Mean-RTT-Offset algorithm, as this change boosted the global clock precision of JK. This finding could be considered as another contribution of the



(a) after 0 s



(b) after 10 s

Fig. 3: Maximum clock offset measured between any process and the ref. process X seconds after the clock synchronization. Horizontal bars represent the mean of the maximum clock offsets over the 10 `mpiruns`; *Jupiter*, 32×16 processes.

present paper, as it improves JK significantly. Interestingly, only 20 ping-pongs are required for JK to obtain these good results on *Jupiter*, which substantially reduces the time that the JK algorithm needs to complete, making it a valid option for practical clock synchronization purposes when the number of processes is relatively small.

Overall, we can identify the advantage of using HCA3 over JK in terms of synchronization duration. Even though it provides the most accurate results in this setting, JK needs almost 60 s on average to synchronize clocks. In sharp contrast, HCA3 only needs about 2 s for completing the same task. This is not surprising as we can expect a speedup of $\mathcal{O}(\frac{p}{\log p})$, which for $p = 512$ is approximately 56. However, since we use more ping-pongs for HCA3 than for JK, the experimentally observed speedup is only 15.

IV. Hierarchical Clock Synchronization

Although synchronizing distributed clocks is relatively fast using algorithms like HCA3, there is still room for improvement. Most MPI libraries contain a number of multi-level algorithms for collective MPI operations, which distinguish between intra- and inter-node communication. The same can be done for clock synchronization, which we will now look at.

A. General Structure

There are two main observations when examining the discussed clock synchronization algorithms and today's cluster architectures. Most clusters or supercomputers are composed of a large number of multi-socket and multi-core compute nodes. In many of these SMP compute nodes, the individual cores use the same time source, at least at socket level. Thus, it should be enough to learn only one clock drift model to other processes outside of my socket or compute node. Moreover, typically the jitter observed for intra- or inter-node communication is different, and in such scenarios, one may want to use a different synchronization algorithm or different parameter settings at each level. However, and more importantly,

Algorithm 3 ClockPropSync

```
comm - node level communicator
p_ref - rank that has been synchronized with global root

function SYNC_CLOCKS(comm, clk)
    if my_rank == p_ref then
        buffer ← FLATTEN_CLOCK(clk)
        MPI_BCAST(sizeof(buffer), root=p_ref, comm)
        MPI_BCAST(buffer, root=p_ref, comm)
    else
        MPI_BCAST(buf_size, root=p_ref, comm)
        buffer ← MALLOC(buf_size)
        MPI_BCAST(buffer, root=p_ref, comm)
        clock ← UNFLATTEN_CLOCK(buffer)
```

we can reduce the number of synchronization steps, since—on many architectures—only one process per compute node has to be synchronized with a reference node, and the other processes on the same compute node simply get a copy of the clock model from this already synchronized process.

We call the general multi-level clock synchronization method H^{l}HCA, where l determines the number of levels used. We introduce two concrete realizations in Sections IV-C and IV-D.

B. Clock Propagation Algorithm: ClockPropSync

If cores on a shared-memory compute node have a common time source, there is no need to synchronize clocks with all cores individually. In this scenario, we can employ the ClockPropSync algorithm sketched in Algorithm 3 at intra-node level. Although the idea is very simple, the concrete realization needs some care. The goal is to copy the clock model of the reference process to all other processes that are mapped to the same shared-memory domain. To do so, the reference process only needs to broadcast its clock model to all other processes in this shared-memory domain (implemented as an MPI communicator). Here, the problem is that we use nested clock models (internally, they are implemented using a decorator pattern). Let us give an example. Assume that we first synchronize the clock of process 2 with process 0. After doing so, we get a clock model $cm(0, 2)$ that process 2 will use to synchronize its children. Then, if process 4 is a child of process 2, we build another clock model $cm(cm(0,2),4)$, and this is why we get a nested structure of clock models. Moreover, clock models at each level can theoretically be of different type, e.g., one global clock type could use a linear extrapolation model, whereas another type may only use a static clock offset model. If all processes on the same node as process 4 should get a copy of its clock model, we pack this nested structure into a flat message buffer, broadcast the buffer to all processes on this node, and re-instantiate the data structure on the recipients' side. This is what we call `flatten_clock` and `unflatten_clock` in the pseudo code.

C. Concrete Realization 1: H^2HCA

The algorithm H^{2}HCA shown in Algorithm 4 is rather a synchronization scheme than a new clock synchronization algorithm. On most systems today, cores on a shared-memory node have the same time source, and thus, we create only two communicators, one for inter-node and one for intra-node communication.

We now assign a concrete clock synchronization algorithm to each of the two levels. Such a hierarchical composition of clock synchronization algorithms only works correctly if the algorithms are put together in a semantically correct fashion. For example, ClockPropSync can only be applied if all processes on this level indeed use the same time source, otherwise the resulting clock model will be incorrect. In order to ensure that cores have the same time source, we can check the result of `clock_getcpuclockid(0)` on Linux-based systems (cf. manpage of `time.h`). In contrast to ClockPropSync, all other clock synchronization algorithms (HCA2, HCA3, JK) can be mixed arbitrarily without restrictions.

D. Concrete Realization 2: H^3HCA

The clock hierarchy shown above can be extended if needed. For example, if each socket on a compute node has a different time source, an additional level can be introduced to the clock synchronization scheme. This new scheme called H^{3}HCA uses three different communicators (groups of processes) to which (possibly different) clock synchronization algorithms can be attached: (1) one communicator comprising one process (the first one) on each compute node for the inter-node communication, (2) on each node, one communicator containing one process from each socket, and (3) at the level of each socket, one communicator comprising all processes on that socket.

In such a case, information about the hierarchical level of each process can easily be retrieved using MPI and `hwloc` [20]. MPI-3 provides functionality to obtain communicators comprising all processes on a shared-memory node (`MPI_COMM_TYPE_SHARED`) and `hwloc` can be used to detect the socket identifier that each process is mapped to.

H^{3}HCA illustrates how the number of levels of the H^{l}HCA scheme can be adjusted to match the hardware architecture.

E. Experimental Analysis of H^2HCA

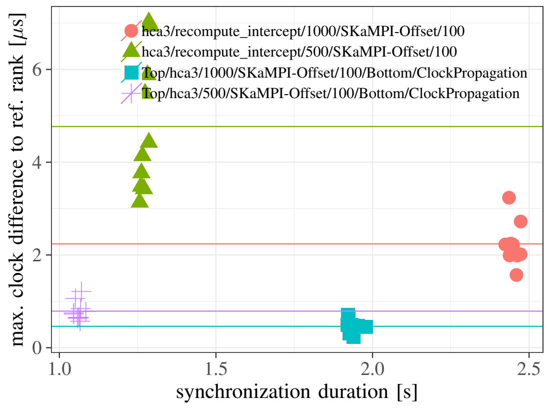
We show experimental results for the three different machines listed in Table I. In each experiment, we created processes on all available cores and pinned processes to cores. We also note that different calls to `mpirun` were done within the same node allocation (which is important for *Titan*). We do not show experimental results for H^{3}HCA, as they were found to be almost identical to the ones produced by H^{2}HCA. Since the compute nodes in our experiments have a common time source, we can treat all cores on a particular node equally.

Algorithm 4 H^{2}HCA

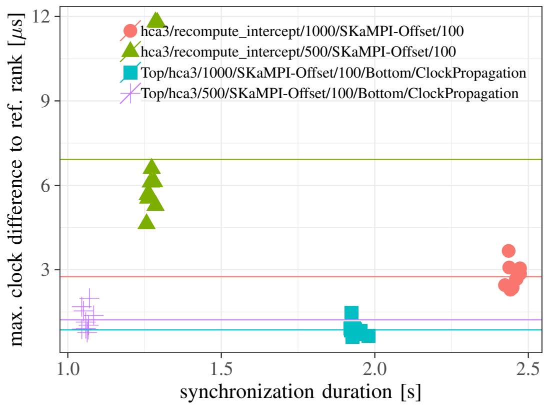
```
clk - base clock that provides a local time on each process
sync_internode - algorithm for inter-node clock synchronization
sync_intranode - algorithm for intra-node clock synchronization

1:  function SYNC_CLOCKS(comm, clk)
    // the creation of the communicators is only done once
    // e.g., split with MPI_COMM_TYPE_SHARED
2:      comm_internode ← CREATE_COMM_INTERNODE(comm)
3:      comm_intranode ← CREATE_COMM_INTRANODE(comm)
    // Step 1: synchronization between nodes
4:      global_clk1 ← new GLOBALCLOCKLM(clk, 0, 0)              // dummy clock
5:      if comm_internode $\neq$ MPI_COMM_NULL then
6:          size ← MPI_COMM_SIZE(comm_internode)
7:          if size > 1 then
8:              global_clk1 ← sync_internode→SYNC_CLOCKS(comm_internode, clk)
    // Step 2: synchronization within compute node
9:      size ← MPI_COMM_SIZE(comm_intranode)
10:     global_clk2 ← global_clk1
11:     if size > 1 then
12:         global_clk2 ← sync_intranode→SYNC_CLOCKS(comm_intranode,
                                                    global_clk2)
13:     return global_clk2
```

In the experiments, we compare two configurations of HCA3, which were found to perform well in isolation, with two configurations of H^{2}HCA. As concrete realization for H^{2}HCA, we employed HCA3 at top level (inter-node) and ClockPropSync at bottom level (intra-node). We do not show experimental results of other combinations of algorithms, as this combination led to the best results in terms of accuracy and duration. Of course, one could tune parameter settings or choose different combinations of clock synchronization algorithms for special cases, but our primary focus here is to provide a general proof-of-concept.

In Figure 4 we compare similar configurations of HCA3 (in isolation) and H^{2}HCA, where the number of fit points and the number of ping-pongs used in H^{2}HCA are simply taken from HCA3. We can see that the hierarchical composition can indeed reduce the synchronization time and at the same time improve the accuracy of the global clock, as the number of required fitted linear models is lower, which in turn reduces the overall error of the clock model. In this particular case with 512 (32×16) processes, we can obtain a very accurate global clock model in about $1\,\mathrm{s}$. One may ask why the time to synchronize the clocks was not reduced by a factor of approximately 2 $(9/5)$, as the hierarchical method only needs $\log 32 = 5$ rounds, while the flat case (HCA3) needs $\log 512 = 9$ rounds. The reason is that we need to create communicators to apply the hierarchical synchronization framework, which is a collective operation with non negligible costs. Although we only need to create these communicators once, we decided to include them in the measurements, as this allows for a more realistic and fairer assessment of the different algorithms. We can also observe in Figure 4b that the global clock model is still very

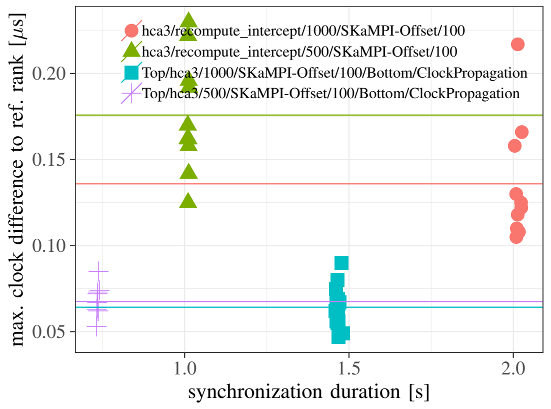


(a) after 0 s

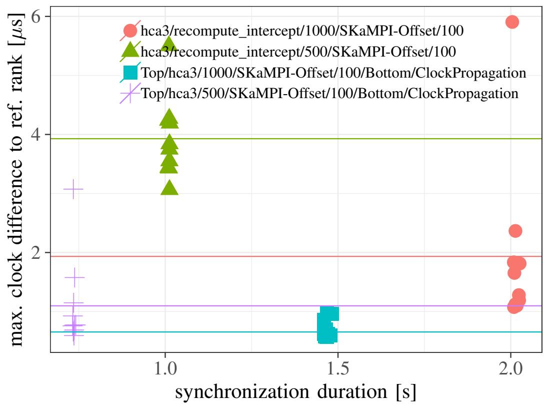


(b) after 10 s

Fig. 4: Maximum clock offset by seconds after clock synchronization, *Jupiter*, *nmpiruns* $= 10$, 32×16 processes.



(a) after 0 s



(b) after 10 s

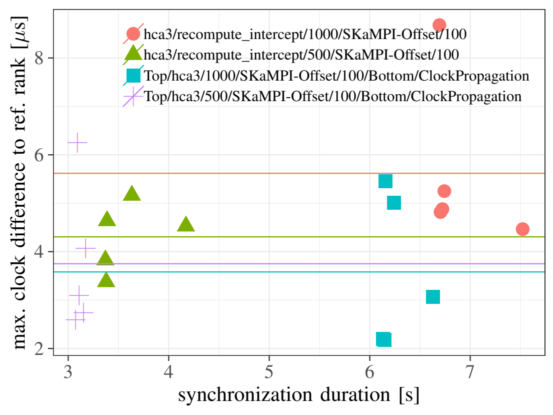
Fig. 5: Maximum clock offset by seconds after clock synchronization, *Hydra*, *nmpiruns* $= 10$, 36×32 processes.

precise after 10 s, since the maximum clock offset of both variants of H^{2}HCA was found to be less than 1.5 µs (this is noteworthy considering that the ping-pong latency on this network is 3 µs to 4 µs).

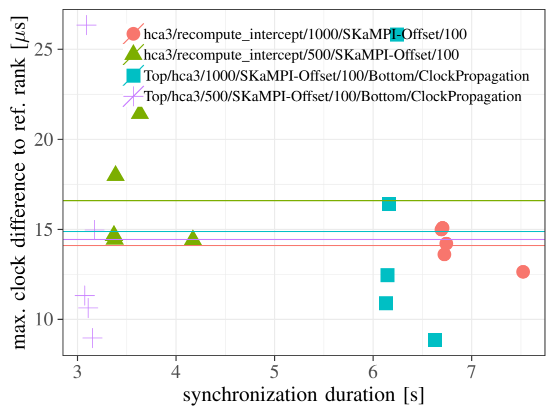
Figure 5 shows a similar set of experiments, but this time the cluster architecture has changed significantly. First, the newer OmniPath network has a smaller latency (which allows for more ping-pongs in a comparable amount of time). Second, the compute nodes have 32 instead of 16 cores, which could be a considerable factor. From Figure 5a, we can observe that all clock synchronization algorithms are able to create a very accurate global clock model, where the clock offset error is less than 0.2 µs on average. However, we can also see that the models lose precision when time passes. The global clocks are still very accurate for the purpose of MPI benchmarking, as the clock offset error is only 1 µs for H^{2}HCA. Nonetheless, the changing clock drift discussed in Section III-C2 is noticeable in the experimental results.

The last set of experimental results for H^{2}HCA is shown in Figure 6, in which we used 16k cores of *Titan*. When checking the accuracy of the global clock in this case, we take a random sample containing 10% of all processes, otherwise the measurement procedure would take too long. This experiment should reveal whether H^{2}HCA can also be used to synchronize clocks over a larger number of cores. Unsurprisingly, the maximum clock offset shown for individual experiments (different `mpiruns`) is larger compared to the previous experiments. Directly after the clock synchronization with H^{2}HCA had finished, we measured an error of less than 4 µs, and after 10 s, the error was found to be roughly 15 µs.

It is also noticeable that the variance between the maximum clock offsets for different calls to `mpirun` has increased. For example, after 10 s, algorithm H^{2}HCA (with 500 fit points) showed one case where the maximum clock offset was in the order of 27 µs. In all the other cases, the maximum clock offset was below 15 µs. Thus, we analyzed the maximum clock offsets of individual processes, and it turned out that only a few processes showed these larger offsets. It could be that the clock drift was changing rapidly in this time period or that the processes experienced temporary network congestion. We have tried to run a larger set of experiments with only 128 nodes on *Titan*, but were not able to reproduce this behavior. Although



(a) after 0 s



(b) after 10 s

Fig. 6: Maximum clock offset by seconds after clock synchronization, *Titan*, $nmpiruns = 5$, 1024×16 processes.

a clock error of about 27 µs can still be considered very small for 16k processes, further investigations are needed to find the root cause of this variance on *Titan*.

V. Further Impact of Global MPI Clocks

Now, we present case studies where a fast clock synchronization algorithm improves accuracy of the studied phenomena.

A. New Measurement Scheme: Round-Time

Before we discuss the case studies, let us introduce a novel measuring scheme for benchmarking MPI collectives, which we call Round-Time. This scheme is mainly driven by the need of taking fair and sound measurements of MPI collectives. In the usual barrier- or window-based approaches, one has to specify the number of repetitions that the benchmarking tool will perform. This number is usually predefined in most benchmarking suites like OSU Micro-Benchmarks and Intel MPI Benchmarks. Although, the user is free to change these values in the code or on the command line, the question of how to choose this number of repetitions remains. In addition, the barrier-based measurement scheme impacts the results, as we will show later. The window-based scheme, where each process waits until a common starting time, requires a predefined window size, which is also hard to estimate.

For that reason, we use a time-based measurement scheme. The idea is that each MPI collective gets an equal amount of time and the benchmarking tool then performs as many measurements as possible until it runs out of time. The pseudo code of the Round-Time scheme is shown in Algorithm 5. After synchronizing the clocks, the reference process will set the new starting time for the next measurement. To do so, it selects the new starting time by taking into account the estimated latency of `MPI_Bcast`, and then it broadcasts the next starting time to all other processes. When all processes arrive at this starting time, they start measuring the MPI collective. If one of the processes is late (it receives the new starting time too late), it sets a flag, which invalidates this particular measurement. After every round of measurements, all processes check whether their

Algorithm 5 Measurement Scheme: Round-Time

```
1:  lat^{MPI_Bcast} ← ESTIMATE_LATENCY(MPI_Bcast)
2:  g_clk ← SYNC_CLOCKS(comm, clk)
3:  t_start ← g_clk→GET_TIME()
4:  nrep ← 0
5:  do
6:      if my_rank == p_ref then
7:          // B ∈ [1, ∞) predefined global constant to allow some slack time
8:          start_time ← g_clk→GET_TIME() + B × lat^{MPI_Bcast}
9:          MPI_BCAST(start_time, root=p_ref)
10:     else
11:         MPI_BCAST(start_time, root=p_ref)
12:     do  // now that all processes know when to start wait for start_time
13:         if g_clk→GET_TIME() ≥ start_time then
14:             set invalid flag if first iteration
15:             break
16:     while TRUE
17:     t_run [nrep] -= g_clk→GET_TIME()
18:     call MPI collective
19:     t_run [nrep] += g_clk→GET_TIME()
20:     if g_clk→GET_TIME() - t_start ≥ MAX_TIME_SLICE then
21:         out_of_time ← 1
22:     MPI_ALLREDUCE( invalid, out_of_time )
23:     if invalid == 0 then
24:         nrep ← nrep + 1
25:     if out_of_time or nrep == max_nrep then
26:         break
27: while TRUE
```

timer is up, and if so, they set the `out_of_time` flag. These two flags (`invalid` and `out_of_time`) are “allreduced” after each round, such that each process knows what to do. The additional calls `MPI_Bcast` and `MPI_Allreduce` can be seen as an undesired overhead, but the advantages outweigh the disadvantages in this case. With this scheme, we can get as many measurements as we desire (unless our time slice is up) and additionally we also get precise measurements, since it does not rely on `MPI_Barrier`. Of course, this scheme is only effective if the latency of `MPI_Barrier` and the latency of the MPI call under investigation are close to each other.

We implemented this scheme in our ReproMPI benchmarking tool and applied it when measuring MPI collectives on *Titan*. In the particular experiment shown later, we use time slices of 5 s for each message size, which is also much safer in terms of expected (expensive) compute hours on *Titan* than hoping that a predefined number of iterations will not take too long.

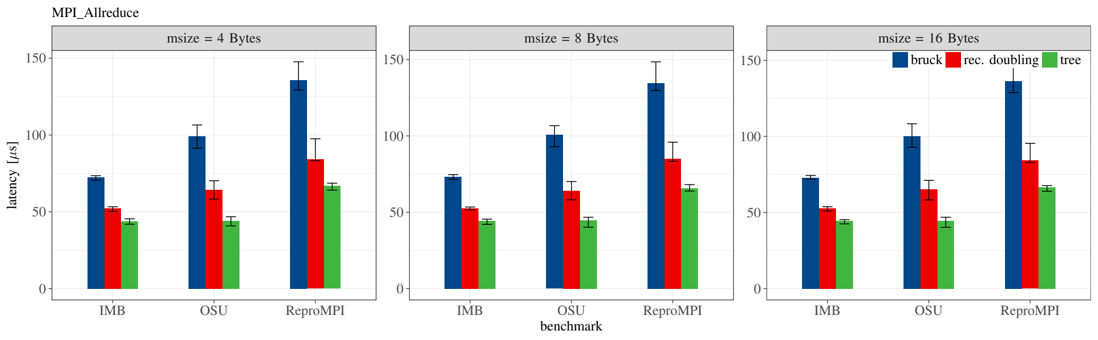


Fig. 7: Average latency (the mean with Intel MPI Benchmarks and OSU Micro-Benchmarks and the median with ReproMPI) of `MPI_Allreduce` using different implementations of `MPI_Barrier`, Open MPI 3.0.0, 32×16 processes, *Jupiter*.

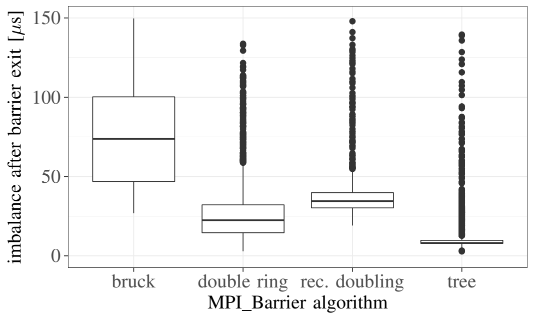


Fig. 8: Imbalance introduced by `MPI_Barrier` algorithms, 32×16 processes, *nmpiruns* $= 10$, Open MPI 3.1.0, *Jupiter*.

B. Impact of MPI_Barrier on Benchmarking Results

We discovered one important issue during our work on tuning MPI collectives using MPI performance guidelines (cf. PGMPI [5]). The problem was to find the best implementation of an MPI collective for a given number of processes and a given message size. In general, MPI tuning experiments can be done with various benchmarking tools, such as OSU Micro-Benchmarks, Intel MPI Benchmarks, or ReproMPI. While performing these tuning experiments, we noticed that the tuning results do not only depend on the MPI benchmark suite, but also on the `MPI_Barrier` algorithm used. In previous work, we had already pointed out that the same `MPI_Barrier` implementation is needed to get comparable results [12].

Figure 7 shows the dilemma when tuning MPI collectives. The graphs show the latency of `MPI_Allreduce` measured with different MPI benchmark suites for small message sizes (4 B to 16 B) and different barrier implementations found in Open MPI. We omitted results for the "double ring" barrier algorithm, as this algorithm had an even larger influence on the results. We can observe a significant impact of the barrier algorithm used internally. Interestingly, the "tree" algorithm gives the smallest latency in all cases. One could have expected that the Bruck or the recursive doubling algorithm have less impact on the results, as they are working more synchronously.

However, thanks to the accurate global clock of HCA3, we can examine the imbalance introduced by different barrier algorithms. The results for 32×16 processes are shown in

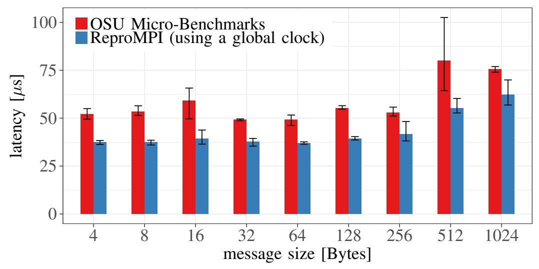
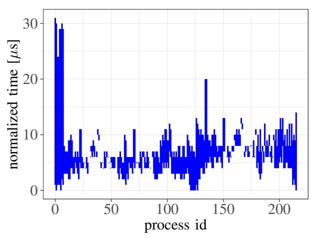


Fig. 9: Latency of `MPI_Allreduce` measured with OSU Micro-Benchmarks and ReproMPI (using the Round-Time scheme), 64×16 processes, *nmpiruns* $= 3$, *Titan*; (error bars denote the min and max of the average latency measured).

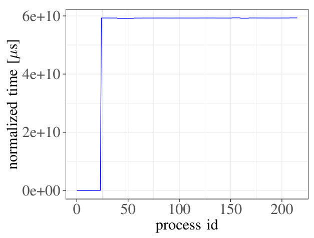
Figure 8, which compares distributions of process imbalance for 500 calls to `MPI_Barrier` over five calls to `mpirun` (in total 2500 data points each). To measure this imbalance, we synchronize the barrier with a common start time (Round-Time) and record the timestamp when each process exits the barrier. We compute the maximum skew between the first and the last process that leave the barrier, and this duration is called imbalance. A barrier-based measurement scheme suffers less from barrier effects if this imbalance is small. We see in Figure 8 that the tree algorithm is by far the best in terms of average imbalance. The take-away message is twofold: first, these phenomena can now be studied thanks to our precise global clock and second, we should resort to time-based process synchronization when measuring with small message sizes. Supporting evidence is provided in Figure 9, which compares the average latency of `MPI_Allreduce` measured with the OSU Micro-Benchmarks 5.4.2 and with ReproMPI (using the Round-Time scheme) on *Titan* with 64×16 processes. We can clearly observe that the latencies reported by the OSU Micro-Benchmarks are affected by the barrier synchronization, and so will be the results reported by Intel MPI Benchmarks or ReproMPI (if the barrier-based scheme were applied).

C. Impact of Timing Issues on MPI Tracing

Last, we want to look at one use case of global clocks when analyzing the performance of an MPI application. As mentioned in the introduction, MPI profiling and tracing tools help to

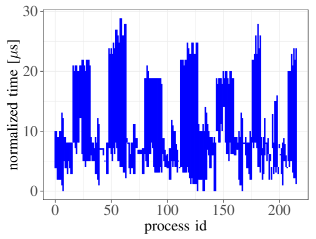


(a) global clock

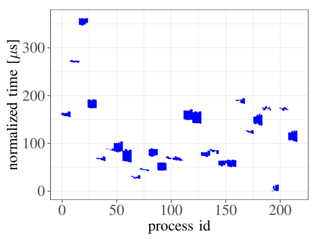


(b) local clock

time source: `clock_gettime`



(c) global clock



(d) local clock

time source: `gettimeofday`

Fig. 10: Gantt charts (traces) marking the start time and the duration (for each process) of the 10th iteration of `MPI_Allreduce` in AMG2013 using a global clock (left) or a local clock (right), Open MPI 3.1.0, 27×8 procs., *Jupiter*.

discover performance bottlenecks in applications. However, precise timings are required if one wants to fine tune applications. Event recording tools like Score-P [21] rely on accurate clocks when profiling or tracing MPI applications. These tools usually interpolate timestamps between two different points in time (e.g., epochs) and this is often done post-mortem.

Here, we will show that having an accurate global clock could be beneficial. For that purpose, we use one of the DOE Mini-apps called AMG2013 that can be found on the NERSC website [7]. For this application, MPI profiling data has been made available [22]. In this specific case (AMG2013 with inputs $N = 40$, $P = 6$, cf. [22]), the application spends about 80% of the time in `MPI_Allreduce` with a buffer size of 8 B. If one would like to tune this particular MPI function, one will need very accurate timestamps.

Figure 10 shows Gantt charts that we obtained by tracing `MPI_Allreduce` of AMG2013 (using our own tailor-made MPI tracing library that first executes H^2HCA to provide a global clock while tracing). We configured Open MPI to use `clock_gettime` as the time source in a first case and to use `gettimeofday` in a second case. The problem with `clock_gettime` for tracing is that the timer offsets between cores may be different, as can be seen in Figure 10b. If we now trace and plot an individual event like `MPI_Allreduce` with timestamps coming from `clock_gettime`, individual start and end times of processes are not visible. When using `gettimeofday`, the local timestamps are now closer to each other and the situation improves. Now the start and the end times of all events are visible in Figure 10d. Yet, for fine grained events like one call to `MPI_Allreduce` these timers are still too imprecise. When using our logical, global clock, we get much more accurate results with both time sources, and it is now possible to see that processes spend roughly 30 µs in `MPI_Allreduce`, independently of the time source.[2]

VI. Conclusions

MPI benchmarking and MPI performance analysis are complicated tasks, especially when the events under investigation are relatively short. A precise logical, global clock can improve the accuracy of both tasks. When benchmarking MPI functions, a global clock can be used to synchronize processes based on timestamps, which reduces the measurement error compared to the use of `MPI_Barrier`, if the latencies of the MPI function under investigation and `MPI_Barrier` are in the same order of magnitude. For the purpose of tracing, global clocks can help to obtain more accurate timestamps for a more realistic view of the trace.

We have presented a new clock synchronization algorithm called HCA3 and a novel clock synchronization scheme called H^lHCA, where the latter exploits the architectural levels of today's compute clusters and supercomputers. HCA3 computes a logical, global clock model in a tree-wise fashion, thereby trying to minimize the model error. This algorithm can be used as a building block in our new hierarchical synchronization scheme H^lHCA, where a different clock synchronization algorithm can be assigned to different architectural levels of the machine, e.g., an algorithm that should be used for inter-node clock synchronization and another one that should be applied at compute node level. We have proposed two realizations of this hierarchical scheme, one with two (H^2HCA) and one with three logical levels (H^3HCA). In one possible implementation of H^2HCA, HCA3 was used for the inter-node clock synchronization and a simple cloning step was used for the intra-node clock synchronization. This hierarchical clock synchronization method has been empirically found to be not only faster than previous approaches but also more accurate.

We have also demonstrated that logical, global clocks can be used to avoid the influence of `MPI_Barrier` on MPI benchmarking results. In addition, we have introduced a novel benchmarking scheme to measure MPI collectives called Round-Time, which uses the logical, global clock to synchronize processes and also ensures that the latency of an MPI operation for different message sizes is measured for a comparable amount of time. Last, we have shown that MPI tracing tools can benefit from our clock synchronization algorithm.

Acknowledgments

This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

[2]We note that there was no particular reason to show the 10th execution of `MPI_Allreduce`, but since all other iterations led to a similar picture, we decided to display results of the 10th execution.

Appendix A
Additional Algorithms

Algorithm 6 Check-Global-Clock

```
wait_time - time in seconds that is waited before global clock is checked again
1:  function CHECK_CLOCK_ACCURACY(comm, clk, wait_time)
2:      g_clk ← SYNC_CLOCKS(comm, clk)
3:      if my_rank == p_ref then
4:          timestamp ← g_clk→GET_TIME()
            // measure clock offset directly after synchronization
5:          for $p$ in $0$ to $nprocs - 1$ do
6:              if $p$ != p_ref then
7:                  clockoff[$p$,0] ← MEASURE_OFFSET(comm, g_clk, p_ref, $p$)
8:          while g_clk→GET_TIME() $\leq$ timestamp + wait_time do // busy wait
            // measure clock offset wait_time seconds after the synchronization
9:          for $p$ in $0$ to $nprocs - 1$ do
10:             if $p$ != p_ref then
11:                 clockoff[$p$,1] ← MEASURE_OFFSET(comm, g_clk, p_ref, $p$)
12:     else
13:         MEASURE_OFFSET(comm, g_clk, p_ref, $p$)
14:         MEASURE_OFFSET(comm, g_clk, p_ref, $p$)
15:     return clockoff
```

Algorithm 7 SKaMPI-Offset

```
nexchanges - number of ping-pongs used to determine the clock offset
1:  function MEASURE_OFFSET(comm, clk, p_ref, client)
2:      td_min ← $-\infty$
3:      td_max ← $\infty$
4:      o_obj ← NULL
5:      if my_rank == p_ref then                     // process with reference clock
6:          for $i$ in $0$ to $nexchanges - 1$ do
7:              MPI_RECV(dummy_time, 1, MPI_DOUBLE, client)
8:              t_last ← clk→GET_TIME()
9:              MPI_SEND(t_last, 1, MPI_DOUBLE, client)
10:     else if my_rank == client then                     // client process
11:         for $i$ in $0$ to $nexchanges - 1$ do
12:             s_slast ← clk→GET_TIME()
13:             MPI_SEND(s_slast, 1, MPI_DOUBLE, p_ref)
14:             MPI_RECV(t_last, 1, MPI_DOUBLE, p_ref)
15:             s_now ← clk→GET_TIME()
16:             td_min ← MAX(td_min, $t\_last - s\_now$)
17:             td_max ← MIN(td_max, $t\_last - s\_slast$)
18:         diff ← $(td\_min + td\_max)/2$
19:         o_obj ← new CLOCK_OFFSET(clk→GET_TIME(), diff)
20:     return o_obj
```

Algorithm 8 Mean-RTT-Offset

```
nexchanges - number of ping-pongs used to determine the clock offset
1:  function MEASURE_OFFSET(comm, clk, p_ref, client)
2:      o_obj ← NULL
3:      if have_rtt == 0 then  // RTT between these two processes is not yet measured
4:          rtt ← measure_rtt(clk, p_ref, client)
5:          have_rtt ← 1
6:      if my_rank == p_ref then                     // process with reference clock
7:          for idx in $0$ to $nexchanges - 1$ do
8:              MPI_RECV(dummy_time, 1, MPI_DOUBLE, client)
9:              tlocal ← clk→GET_TIME()
10:             MPI_SSEND(tlocal, 1, MPI_DOUBLE, client)
11:     else if my_rank == client then                     // client process
12:         for idx in $0$ to $nexchanges - 1$ do
13:             MPI_SSEND(dummy_time, 1, MPI_DOUBLE, p_ref)
14:             MPI_RECV(ref_time, 1, MPI_DOUBLE, p_ref)
15:             local_time[$i$] ← clk→GET_TIME()                // local timestamp
16:             time_var[$i$] ← local_time[$i$] $-$ ref_time $-$ rtt$/2$      // current offset
17:         med_idx ← $i$ s.t. $0 \leq i <$ nexchanges  &
                         time_var[$i$] == MEDIAN(time_var)
18:         o_obj ← new CLOCK_OFFSET(local_time[med_idx], time_var[med_idx])
19:     return o_obj
```

References

[1] J. Ridoux and D. Veitch, "Principles of robust timing over the internet," *Communications of the ACM*, vol. 53, no. 5, pp. 54–61, 2010.

[2] S. Shende and A. D. Malony, "The Tau parallel performance system," *IJHPCA*, vol. 20, no. 2, pp. 287–311, 2006.

[3] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *CCPE*, vol. 22, no. 6, pp. 702–719, 2010.

[4] S. Hunold and A. Carpen-Amarie, "Autotuning MPI collectives using performance guidelines," in *HPC Asia*. ACM, 2018, pp. 64–74.

[5] S. Hunold, A. Carpen-Amarie, F. D. Lübbe, and J. L. Träff, "Automatic verification of self-consistent MPI performance guidelines," in *Euro-Par*, ser. LNCS, vol. 9833. Springer, 2016, pp. 433–446.

[6] J. L. Träff, W. D. Gropp, and R. Thakur, "Self-consistent MPI performance guidelines," *IEEE TPDS*, vol. 21, no. 5, pp. 698–709, 2010.

[7] "Characterization of the DOE Mini-apps." [Online]. Available: http://portal.nersc.gov/project/CAL/overview.htm

[8] T. Worsch, R. H. Reussner, and W. Augustin, "On benchmarking collective MPI operations," in *EuroMPI/PVM*, ser. LNCS, vol. 2474. Springer, 2002, pp. 271–279.

[9] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale," *IJPEDS*, vol. 25, no. 4, pp. 241–258, 2010.

[10] S. Hunold and A. Carpen-Amarie, "On the impact of synchronizing clocks and processes on benchmarking MPI collectives," in *EuroMPI*. ACM, 2015, pp. 8:1–8:10.

[11] ——, "MPI benchmarking revisited: Experimental design and reproducibility," *CoRR*, vol. abs/1505.07734, 2015. [Online]. Available: http://arxiv.org/abs/1505.07734

[12] ——, "Reproducible MPI benchmarking is still not as easy as you think," *IEEE TPDS*, vol. 27, no. 12, pp. 3617–3630, 2016. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2016.2539167

[13] "Intel MPI benchmarks user guide." [Online]. Available: https://software.intel.com/en-us/imb-user-guide

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

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

[16] T. Jones and G. A. Koenig, "Clock synchronization in high-end computing environments: a strategy for minimizing clock variance at runtime," *CCPE*, vol. 25, no. 6, pp. 881–897, 2013.

[17] D. Becker, R. Rabenseifner, and F. Wolf, "Implications of non-constant clock drifts for the timestamps of concurrent events," in *IEEE CLUSTER*. IEEE Computer Society, 2008, pp. 59–68.

[18] C. Lenzen, P. Sommer, and R. Wattenhofer, "PulseSync: An efficient and scalable clock synchronization protocol," *IEEE/ACM Trans. Netw.*, vol. 23, no. 3, pp. 717–727, 2015.

[19] J. Doleschal, A. Knüpfer, M. S. Müller, and W. E. Nagel, "Internal timer synchronization for parallel event tracing," in *EuroPVM/MPI*, ser. LNCS, vol. 5205. Springer, 2008, pp. 202–209.

[20] B. Goglin, "Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc)," in *HPCS*. IEEE, 2014, pp. 74–81.

[21] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler *et al.*, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing*, 2011, pp. 79–91.

[22] "AMG2013 IPM profile." [Online]. Available: http://portal.nersc.gov/project/CAL/doe-miniapps-ipm-files/amg/p6x6x6/ipm_html/index.html