MPI Process Synchronization in Space and Time

Joseph Schuchart
University of Tennesse, Knoxville
Innovative Computing Laboratory
Knoxville, Tennessee, USA
schuchart@icl.utk.edu

Sascha Hunold
TU Wien
Vienna, Austria
hunold@par.tuwien.ac.at

George Bosilca
University of Tennesse, Knoxville
Innovative Computing Laboratory
Knoxville, Tennessee, USA
bosilca@icl.utk.edu

ABSTRACT

Performance benchmarks are an integral part of the development and evaluation of parallel algorithms, both in distributed applications as well as MPI implementations themselves. The initial step of the benchmark process is to obtain a common timestamp to mark the start of an operation across all involved processes, and the state-of-the-art in many applications and widely used MPI benchmark suites is the use of MPI barriers. In this paper, we show that the synchronization in space provided by an `MPI_Barrier` is insufficient for proper benchmark results of parallel distributed algorithms, using MPI collective operations as examples. The resulting lack of a global start timestamp for an operation leads to skewed results, with a significant impact of the used barrier algorithm. In order to mitigate these issues, we propose and discuss the implementation of `MPIX_Harmonize`, which extends the synchronization in space provided by `MPI_Barrier` with a time synchronization to guarantee a common starting timestamp across all involved processes. By replacing the use of `MPI_Barrier` with `MPIX_Harmonize`, benchmark implementors can eliminate skews resulting from barrier algorithms and achieve stable performance benchmark results. We will show that the proper time synchronization can have significant impact on the benchmark results for various implementations of `MPI_Allreduce`, `MPI_Reduce`, and `MPI_Bcast`.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**.

KEYWORDS

MPI, collective communication, process synchronization, clock synchronization, OSU benchmarks, reduce, allreduce, broadcast, barrier

ACM Reference Format:
Joseph Schuchart, Sascha Hunold, and George Bosilca. 2023. MPI Process Synchronization in Space and Time. In *Proceedings of EuroMPI2023: the 30th European MPI Users' Group Meeting (EUROMPI '23), September 11–13, 2023, Bristol, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3615318.3615325

1 INTRODUCTION

Performance benchmarking is an integral part of the development of high-performance parallel applications and runtime systems. In order to understand the performance impact of code changes, it is imperative to properly measure the performance of the application or runtime system before and after a change. Execution time is a fundamental metric by which performance is evaluated.

The evaluation of performance impacts can either happen at the application level, measuring the execution time of a full application run, or at the level of individual components. In the latter case, the time measurement of a component happens as part of a benchmarking application measuring several invocations of the component under test. In any component that involves concurrent, interacting agents, the quality of the measurement depends on the quality of the synchronization of these agents. If the execution time of the component under test is only a fraction of the skew introduced by the synchronization then the measurement taken is dominated by the skew, not the performance of the component under test.

In the context of MPI [26], it is common practice to benchmark different components of an MPI implementation, including point-to-point, collective, and one-sided communication operations. Several MPI benchmark suites are available that cover these aspects and are widely used, including Intel MPI Benchmarks (IMB), OSU micro-benchmarks (OMB), and mpiBench by Lawrence Livermore National Lab [1, 2, 4]. In this work, we will focus on collective operations, which are most heavily impacted by the encompassing synchronization due to their tightly coupled nature. We will show that the synchronization mechanisms used in many of the benchmarking suites are insufficient for providing reliable performance measurements. We thus propose an extension to the MPI standard, `MPIX_Harmonize`, to enable the synchronization of processes both in time and space in order to provide a high quality synchronization primitive for use by benchmarks.

While our discussion focuses on MPI collective operations, we believe that the discussion applies to component-based application benchmarks as well, e.g., to determine load-imabalance at the application level [11].

The remainder of this paper is structured as follows: section 2 provides the detailed motivation for this work followed by an overview of related work in section 3. section 4 discusses strategies for synchronizing clocks in a distributed system. section 5 presents our proposal for an MPI function that synchronizes processes in space and time. We evaluate its use in section 6, and we draw conclusions in section 7.

2 MOTIVATION

The MPI community has a long history of developing new collective algorithms and tuning implementations [23–25, 28, 31, 36]. In many cases, the empirical evaluation of these novel algorithms and improved implementations relies on the above mentioned MPI

EUROMPI '23, September 11–13, 2023, Bristol, United Kingdom

ACM ISBN 979-8-4007-0913-5/23/09…$15.00
https://doi.org/10.1145/3615318.3615325

benchmark suites. Moreover, these benchmarks are often used in the evaluation of HPC systems and system features [34, 38].

Given the central role these benchmarks play in the development and evaluation of MPI implementations, it is important that these benchmarks provide reliable timing information to determine whether any given algorithm or implementation provides improved performance for a given configuration, factoring in process count, buffer size, and process topology.

The most commonly used metric for performance of collective operations is the mean of the time required for the operation on each process, i.e., each process takes timestamps before and after the call to the MPI function under test, summarizes the differences between these two timestamps, and computes the mean over all loop iterations. This final benchmark number is the mean of means over all processes, i.e.,

$$t_{avg} = \frac{1}{p}\sum_{p}\left(\frac{1}{N}\sum_{i=0}^{N} t_i\right)$$

for N repetitions on p processes. In addition, some benchmarks will report the maximum and minimum per-process average [14].

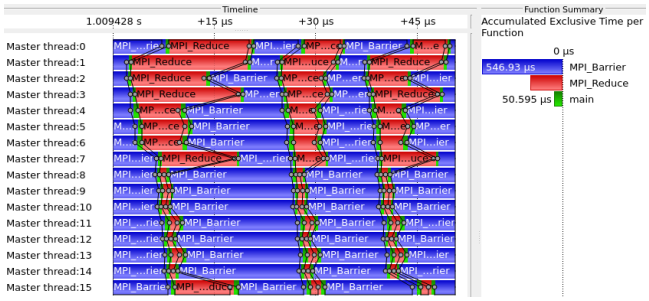
Due to their collective nature, the time spent by any given process in a collective operation may depend heavily on the arrival time of other processes. For example, in a binary tree-based reduction, the progress of any non-leaf process depends on the arrival of its child nodes in the tree. The late arrival of one node will delay its parent nodes. Thus, the used algorithm will determine the exit pattern of the collective operation (e.g., leaf nodes exiting earlier than the root node), which would lead to a non-uniform arrival to the next iteration.

In an attempt to mitigate this problem and to achieve a balanced arrival pattern, many benchmark suites rely on an MPI barrier to synchronize the processes involved before entering the collective operation under test, as shown in the listing below.

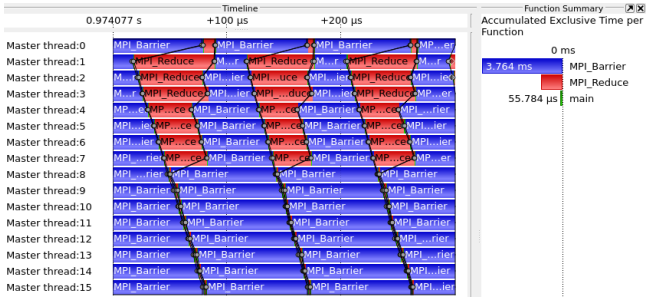
```
1  for (int i = 0; i < niter; ++j) {
2    MPI_Barrier(comm);
3    double begin = MPI_Wtime();
4    MPI_Reduce(..., comm);
5    acc_time += MPI_Wtime() - begin;
6  }
```

At first glance, this seems sufficient to make sure that no process is delayed by the `MPI_Reduce` of the previous iteration. In fact, `MPI_Barrier` is a *synchronizing* collective procedure, i.e., processes are guaranteed to only return from a call to `MPI_Barrier` after all other processes in the respective group have entered a call to `MPI_Barrier`. Thus, a call to `MPI_Barrier` is synchronizing in *space*, i.e., over all processes of the group.

However, the MPI standard does not provide any guarantees about time synchronization, i.e., it does not guarantee that all processes must exit a call to `MPI_Barrier` at the same time. Indeed, MPI implementations typically provide several different barrier algorithms that are selected based on the current configuration. Some implementations may provide lower latency for small process counts while others may provide scalable synchronization at system-scale. While low-latency synchronization across the process space may be desirable for mere process synchronization (e.g., in an MPI RMA application), it is insufficient for the purpose of benchmarking of distributed algorithms [15, 37].



(a) Using barrier algorithm `linear`. Average reduce operation latency reported by the OSU benchmarks: 2.78 µs.



(b) Using barrier algorithm `double_ring`. Processes 1–7 leave the barrier earlier than process 0 (the root), causing significant wait times. Processes 8–15 leave the barrier late and thus spend almost no time in `MPI_Reduce`. Average reduce operation latency reported by the OSU benchmarks: 9.98 µs.

Figure 1: Vampir visualizaiton of execution trace for the OSU benchmark for `MPI_Reduce` (red) with 1 B data using different barrier implementations (red) on 16 processes using Open MPI.

2.1 Choice of Barrier Algorithm

Figure 1 demonstrates the impact of this problem on `MPI_Reduce`: the same reduction algorithm is used with two different barrier algorithms in Open MPI (`linear` in Figure 1a and `double-ring` in Figure 1b). The execution traces (recorded using Score-P [18] and visualized using Vampir [27]) show that the choice of barrier implementation heavily influences the arrival pattern for the reduction operation, causing a reported 3.5x slow-down of the same reduction algorithm when using the `double-ring` barrier over the `linear` barrier algorithm.

While these are two extreme examples, they demonstrate well how the barrier exit pattern become the arrival pattern for the operation under test. Figure 2 shows the measured latencies for several reduction algorithms paired with different barrier algorithms in Open MPI on Hawk (see section 6 for a system description).

Figure 2 demonstrates that the impact of the used barrier algorithm (horizontal) can be as significant as the choice of reduction algorithm (vertical). For example, the combination of `linear` reduction and `double-ring` barrier yield the lowest average latency

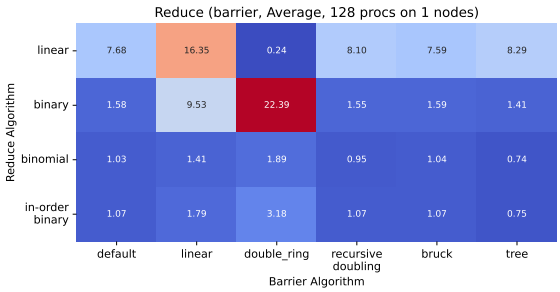


Figure 2: Average latency (microseconds) reported by the OSU benchmark for various `MPI_Reduce` implementations using different barrier implementations in Open MPI, with a message size of 4 B and 128 processes.

while the same reduction algorithm paired with the `linear` barrier yield the second highest average latency. Using any other barrier algorithm it appears that the `binomial` reduction algorithm yields the lowest average latency.

While the benchmark does not choose the used barrier algorithm, the measured performance for a collective algorithm should never depend on another collective operation issued by the benchmark.

2.2 Barrier Skew

Besides the differences between various barrier algorithms, we also highlight the fact that the same barrier algorithm may yield different synchronization quality between invocations of `MPI_Barrier`. Figure 3 shows the measured *synchronization skew* of processes leaving a call to `MPI_Barrier`. We define synchronization skew σ as the difference between the first and the last process exiting the call to `MPI_Barrier`, i.e.,

$$\sigma = \max_{t_0 \ldots t_p} - \min_{t_0 \ldots t_p} \quad ,$$

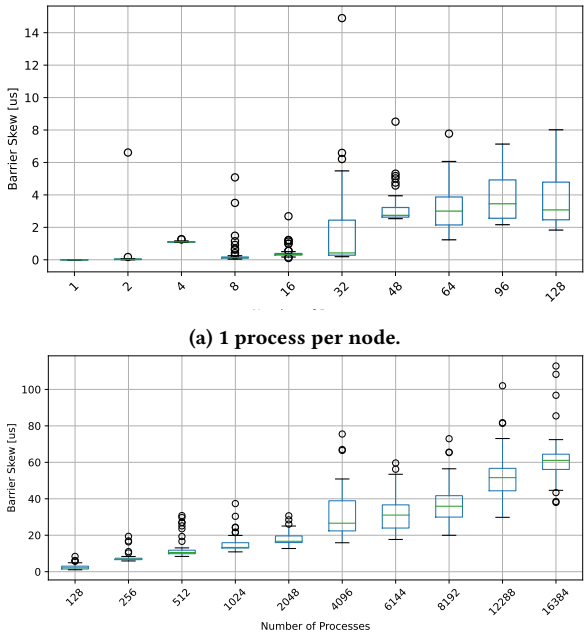
where t_i denotes the execution time of `MPI_Barrier` on process i. With a single process per node (Figure 3a), the skew is found to be mostly in the single digits but with a fairly wide range and some notable outliers up to 14 µs. However, such a skew is significant for reduction operations with an average latency reported in the single-digit microsecond range.

With full nodes (128 processes per node, Figure 3b), the skew increases significantly, with outliers above 100 µs and a mean of 60 µs at 16k processes (128 nodes). As we will show in section 6, even at that scale, the average latency of reduce is reported to be 30 µs for 4 B messages.

2.3 Unreliable Measurements

Given the above observations, we postulate that the state-of-the-art in synchronization and time measurement in MPI collective operation benchmarks is insufficient. It also stands to reason that some of the results on collective algorithm evaluation published in the past suffered from undue influence of the synchronization mechanism employed in the used benchmark suites.

One reason for these short-comings can likely be found in a lack of awareness, given that these benchmark suites have been under



(a) 1 process per node.

(b) 128 processes per node.

Figure 3: Barrier skew (time difference between last and first process exiting the barrier) on up to 128 nodes. Data from 50 repetitions for each configuration.

development for more than a decade. Even if the developers were aware of this problem, the perceived complexity of implementing a proper clock synchronization scheme might outweigh the perceived benefits. We are thus convinced that MPI itself should provide a routine that synchronizes both *in space* (similar to `MPI_Barrier`) and *in time* in order to provide a building block for reliable performance benchmarking in a distributed application.

3 RELATED WORK

The synchronization of various clocks in a distributed system is essential for its correct operation. Services like NFS or Slurm usually require a certain clock precision to work correctly. This level of accuracy can often be established by employing clock synchronization methods, such as the Network Time Protocol (NTP) [3]. However, the clock accuracy offered by NTP is often insufficient for analyzing events in supercomputers. Depending on the number of compute nodes involved, the duration of a broadcast is in the range of ten to hundred microseconds, which is several orders of magnitude smaller than the accuracy that NTP provides.

As pointed out by Jones et al. [17], the "time protocol in use, NTP, is not suitable for providing the level of time synchronization necessary for important system software tasks such as coordinated scheduling." Therefore, other clock synchronization methods are needed for these types of tasks on supercomputers. The Precision Time Protocol (PTP) provides an alternative way of synchronizing distributed clocks with a higher accuracy than what NTP can guarantee. Libri et al. [21] showed that a proper concurrent use of

NTP and PTP, the clock accuracy can be about 2 µs to 3 µs, while inducing a negligible overhead.

The problem for the MPI layer is that it can usually not assume that a certain clock accuracy is provided by the underlying hardware or software. Similarly to the observation made by Jones et al. [17], many HPC centers still provide a clock accuracy similar to NTP.

Inaccuracies of distributed clocks may have a strong impact on the performance analysis of HPC applications. A common method to investigate inefficiencies of HPC codes is by recording and analyzing event traces. For each process-local event, e.g., a function call, the timestamps before and after the event are recorded. Afterwards, all events are displayed using some form of Gantt chart viewer, e.g., using Vampir [27] or Paraver [7]. Inaccurate, distributed clocks would render performance analyses meaningless, as events would be displayed out-of-place, due to the out-of-sync process-local clocks when the trace was recorded.

For that reason, several clock synchronization and trace correction methods for MPI have been proposed. Doleschal et al. [9] proposed a two-step solution to the problem of recording traces with inaccurate clocks. In their approach, clocks are periodically synchronized and the resulting clock offset information are used to interpolate the timestamps between synchronization points. It is important to note that the clock drift between hardware clocks is not constant over time, which requires such a piece-wise interpolation scheme. In a second step, time stamps are corrected in order to fulfill all logical event constraints, e.g., a send event must have started before the corresponding receive event has finished. Becker et al. [6] also tackle the problem of inaccurate trace events by employing the controlled logical clock (CLC) algorithm, which serves a similar purpose as Lamport's logical clock [19]. In order to repair the timestamps found in large traces, Becker et al. [6] proposed a parallel version of the CLC algorithm.

Another typical use case of highly synchronized clocks is micro-benching MPI operations, in particular collective operations. The problem is to precisely determine the exact makespan of a collective operation, which is defined as the time between the first process starts communicating until the last process finishes its communication. As mentioned in section 2, a commonly used benchmark pattern simply separates individual collective calls by using `MPI_Barrier`. The problem is that `MPI_Barrier` does not guarantee any type of synchronization in time.

To overcome this problem and to accurately measure MPI collectives, Worsch et al. [37] have proposed a clock synchronization scheme that is implemented in the micro-benchmarking suite SKaMPI. In this work, clocks are synchronized one by one with the master process once, resulting in a time complexity of $O(p)$. Hoefler et al. [12] used a binary tree synchronization scheme instead, reducing the time complexity to $O(\log p)$.

The disadvantage of both previous synchronization schemes is that they exchange the current clock offset only, i.e., the clocks were corrected once. However, the frequencies of quartz crystal oscillators, which are typically used in hardware clocks, are not identical and their difference is also not constant over time. For example, varying temperatures in a computer's enclosure may cause the clocks to drift by several microseconds each second [33]. Jones and Koenig [16] showed how to obtain an accurate clock model in MPI by estimating the clock drift using linear regression. As this scheme works linearly in the number of processes, Hunold and Carpen-Amarie extended that scheme to work in a tree-like fashion, to obtain a time complexity in $O(\log p)$ [13]. They also proposed a hierarchical clock synchronization scheme, where a different clock synchronization algorithm can be employed at different levels of the architectural hierarchy (e.g., on the level of a compute node or of a socket) [15]. Fundamentally, the proposed HCA3 algorithm in [15] works similar to the PulseSync algorithm proposed by Lenzen et al. [20], where one specific node sends timing information into the network, which is then used by the other compute nodes to estimate their offset to this master clock.

The issue of clock synchronization (or the lack thereof) has been raised before (including by one of the authors) [13, 14]. As pointed out above, awareness of the issue may not be enough for implementors to integrate time synchronization mechanisms into benchmark suites, which is why this work proposes an extension to MPI to provide a foundation for reliable benchmark results.

Recent studies have suggested that MPI collective operations are among the most time-consuming communication patterns in HPC applications, with `MPI_Allreduce` and `MPI_Reduce` being the most widely used collectives [8, 35].

Several studies of the influence of process arrival patterns on various collective operations have been conducted [10, 32], including `MPI_Barrier` itself [30], and collective algorithms that are agnostic of these patterns have been proposed [5, 29]. While real-world applications will typically exhibit arbitrary arrival patterns due to load imbalances, we argue that reliable micro-benchmarks should be conducted under predictable conditions that are independent of other aspects of a given MPI implementation, i.e., its choice of barrier algorithm. The inclusion of arrival patterns should be explicit and their impact reflected in the benchmark results.

4 CLOCK SYNCHRONIZATION IN MPI

We start by by introducing some of the key terminology in order to recall the anatomy of clock synchronization algorithms in MPI. At any point in time, the clocks in a distributed system show a specific magnitude of difference, which we call the *clock offset* (CO). We usually use one of the clocks as the reference clock, and all the other clocks have a specific clock offset to this reference clock.

In order to determine the CO between two clocks, a simple ping-pong scheme can be used. Process 1 sends its current timestamp to process 2, which returns its own timestamp. Repetitive execution of this scheme allows to estimate the latency of such a message exchange and eventually the difference between both clocks, as implemented in SKaMPI [37] and NBCBench [12].

As clocks are subject to drift, the CO has to be re-estimated in relatively short intervals, depending on the required accuracy. Therefore, it is worthwhile to reduce the number of clock offset estimations. This can be done by determining the clock drift to account for the non-constant CO over time. However, most clock drift schemes assume that the clock drift is linear in time, which is often only true for a relatively short period of time, e.g., few minutes. In order to estimate this clock drift, we can record several clock offsets over a short period of time and interpolate these clock offsets to determine the drift, as illustrated in Figure 4.

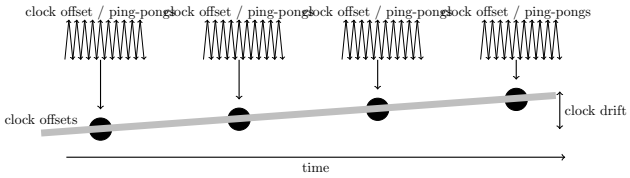


Figure 4: Clock drift (CD) estimation between two processes. Several clock offsets measurements are recorded over time and used to estimate the clock drift via linear regression.

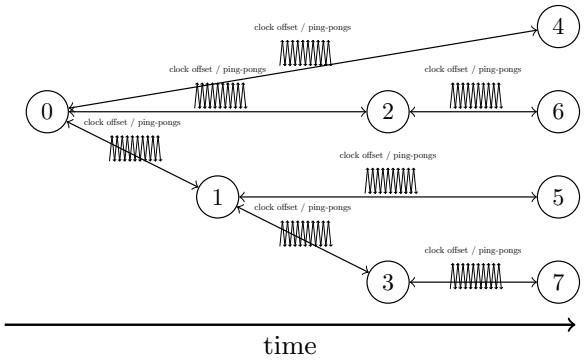


Figure 5: Hierarchical clock synchronization between 8 processes. In this scheme, only the clock offset to the reference clock (process 0) gets estimated.

The two previous schemes can be used to synchronize the clocks between two processes. However, on a large system, we need to synchronize thousands of processes. To that end, tree-like schemes are scalable methods for fast clock synchronization [12, 13, 15]. Figure 5 depicts one such method, where clocks are synchronized using a binomial tree of processes. In this example, only the clock offset between pairs of processes is determined by using the ping-pong method. This scheme is fast and highly efficient, but the clock accuracy will drop quicker than when using a drift-based model.

As mentioned in Section 3, the system topology and the software stack should be taken into account when synchronizing clocks. For example, if all processes on a compute node use the same time-source when calling `MPI_Wtime`, no node-level clock synchronization has to be performed. In these cases, one process per compute node acts as the reference clock for this node, and a distributed clock synchronization algorithm only synchronizes these reference clocks. After each reference clock has determined its clock offset to the main clock (e.g., the clock of process 0), the node-local clock is propagated to all processes residing on the same compute node.

Clock synchronization in MPI does not modify the system clock but uses the determined CO to correct the time provided by the system clock.

5 HARMONIZING MPI PROCESSES

In order for HPC applications and benchmarks to attain reliable time measurements that are not impacted by arbitrary arrival patterns, a synchronization scheme is required that provides a best effort synchronization. MPI could provide a synchronized clock source that is then used by the upper layers to synchronization the beginning of a timed code region. However, given the clock drift mentioned earlier, the application would be responsible for resynchronizing to ensure sufficient accuracy.

Instead of providing a synchronization clock source, we propose a synchronization mechanism that synchronizes the group of processes in space and time, i.e., a system that extends the semantics of `MPI_Barrier` with a requirement for a time-synchronized exit pattern. Upon return, the processes are *in harmony*, i.e., they resume execution at the same time.

5.1 Function Overview

The proposed function `MPIX_Harmonize` has the following signature:

```
int MPIX_Harmonize(MPI_Comm comm, int *flag);
```

As a first input argument, it takes a communicator that represents the group of processes whose execution should be harmonized. This procedure is collective over this group and blocks until all processes have arrived.

The second parameter is an output flag that is set to `1` (or `true`) if the operation was successful. Otherwise it is set to `0` (or `false`). A synchronization operation may fail if, for example, a sudden burst of communication in the network increases the internal communication latency and causes processes to miss the synchronization deadline, as described in subsection 5.2.

Success or failure of this operation is a local property, i.e., some processes may return `true` while other processes return `false`. Synchronizing this flag inside the call would introduce additional communication, which induces skew and distorts the time synchronization. It is thus left to the application to check the value of the returned flag and handle a failed synchronization according to its needs. An example for how to handle failed time synchronization will be provided in subsection 5.3.

While a synchronization may fail, we decided to not treat it as a condition that warrants raising an error. We reserve such drastic measures for more pathological errors, e.g., the use of an invalid communicator or the loss of a process in the group of the communicator. We believe that applications will be able to gracefully handle a failed synchronization but that they may opt to raise an error themselves, e.g., using ULFM to revoke a communicator on which processes may be waiting for communication [22].

5.2 Implementation

1 provides the algorithm used in our implementation of `MPIX_Harmonize`.[1] In essence, the algorithm checks whether clock synchronization is required and if so performs that synchronization. At the end, each process waits until a deadline occurs by which all processes *should* have completed the previous communication. A process that missed the deadline will set the output `flag` to `false`.

In more detail, each process checks two criteria that would trigger a resynchronization of clocks: i) whether in a previous call to `MPIX_Harmonize` the deadline was missed (Line 4), i.e., whether the determined deadline was too early; and ii) whether the elapsed

[1] A proof-of-concept implementation is available at https://github.com/devreal/mpix-harmonize/.

Algorithm 1 Algorithm for `MPIX_Harmonize`.

```
Require: comm                                          ▷ Input communicator
Require: outflag                                       ▷ Output flag
 1: slack ← sync_slack(comm)
 2: flag ← 0                                           ▷ Local and global state
 3: root ← 0
 4: if last_sync_failed(comm) then
 5:     flag ← SYNC_FAILED                             ▷ Locally failed
 6: end if
 7: if elapsed_since_last_sync(comm) > 1.0s then
 8:     flag ← flag | SYNC_EXPIRED                     ▷ Locally expired
 9: end if
10: flag ← MPI_Reduce(flag, MPI_BOR, root, comm)
11: if root = comm_rank(comm) then
12:     if flag then                                   ▷ Global State
13:         sync_time ← −1.0                           ▷ Trigger clock sync
14:         if flag | SYNC_FAILED then
15:             slack ← slack × 1.5                    ▷ Increase slack
16:             sync_slack(comm) ← slack
17:         end if
18:     else
19:         sync_time ← global_time() + slack
20:     end if
21: end if
22: sync_time ← MPI_Bcast(sync_time, root, comm)
23: if sync_time < 0.0 then                            ▷ Negative time triggers sync
24:     sync_clocks();
25:     MPI_Reduce(0, root, comm)                      ▷ Reduce clock-sync jitter
26:     last_sync_time(comm) ← global_time()
27:     sync_time ← global_time() + slack              ▷ New sync time
28:     sync_time ← MPI_Bcast(sync_time, root, comm)
29: end if
30: if sync_time < global_time() then
31:     outflag ← 0                                    ▷ Missed deadline
32:     last_sync_failed(comm) ← TRUE                  ▷ Store for next call
33: else
34:     outflag ← 1                                    ▷ Success, wait for sync time
35:     while sync_time > global_time() do
36:     end while
37: end if
```

time since the last synchronization exceeds 1s (Line 7). This is a conservative estimate based on the observation that clocks tend to drift by several microseconds per second [33].

The resulting information is aggregated at a root process using `MPI_Reduce`, which then determines whether a resynchronization is necessary (Line 13). In that case, a negative synchronization deadline is chosen to be distributed. The algorithm also checks whether the *synchronization slack* should be increased in case a process missed a deadline (Lines 14–17), which will give processes more time to meet the synchronization deadline. If no resynchronization is required, a synchronization deadline is determined based on the previous slack (Line 19). A starting value for the synchronization slack is picked based on the latency of `MPI_Bcast`. The resulting value is distributed to all processes using `MPI_Bcast` (Line 22).

If necessary (i.e., a negative value for the synchronization deadline was broadcast), clocks are synchronized and a new is deadline determined and broadcast to be used instead (Lines 23–29). Eventually, the deadline is checked for feasibility. If the deadline was missed (Lines 31–32), the output flag is set to `0` and the failed synchronization state is set on the communicator (to be checked on the next call, see above). If the deadline is feasible, the output flag is set to `1` and the process waits for the deadline to pass (Lines 34–36).

Overall, the algorithm requires a minimum of one reduce and one broadcast. Using two distinct operations (instead of a single `MPI_Allreduce`) simplifies the task of picking a deadline based on the previously determined duration of the barrier. In the presence of stragglers, `MPI_Allreduce` might not provide the same well-defined behavior as the combination of `MPI_Reduce` and `MPI_Bcast`.

5.2.1 Synchronization Quality. The correctness of process synchronization in space is a binary property, which can be observed using external signalling mechanisms (e.g., files, sockets).

In contrast, synchronization in time is neither binary nor easily observable. As discussed earlier, clock synchronization is a process with limited accuracy. While benchmarks attempting to judge the accuracy of an MPI implementation's time synchronization could implement a custom clock synchronization scheme and compare the results against the synchronization by MPI, the two synchronization schemes will never be 100% accurate. However, we hope that by providing a reference implementation of `MPIX_Harmonize` and the necessary clock synchronization algorithms, we enable other implementations to adopt these implementations and to provide a qualitatively similar synchronization. As we have shown in section 2, even a slightly inaccurate time-based synchronization is significantly better than relying on `MPI_Barrier`.

5.3 Usage in Sample Benchmark

Listing 1 below shows how `MPIX_Harmonize` can be used in a benchmark code. Instead of using `MPI_Barrier` for process synchronization (as shown in section 2), the code calls `MPIX_Harmonize` and stores the returned flag for later examination. After `CHECK_EVERY` iterations—which could be 16 or smaller—the previous iterations are checked for validity (i.e., whether every process' time synchronization was successful) and measurements that were not properly time-synchronized are discarded.

For simplicity, this code assumes at least `MIN_VALID_ITERATIONS` successful (Line 28) iterations. A more robust benchmark implementation would opt to repeat failed benchmarks and abort if a maximum number of retries has been reached.

6 EVALUATION

We evaluated our implementation on Hawk, an HPE Apollo 9000 system installed at HLRS in Stuttgart, Germany.[2] The system comprises dual-socket nodes with 64-core AMD EPYC Rome (7702) CPUs and a dual-rail InfiniBand HDR200 fabric.

All codes were compiled using GCC 10.2.0 and used MVAPICH2 and Open MPI in version 4.1.4 as MPI implementations. The OSU

[2] https://www.hlrs.de/solutions/systems/hpe-apollo-hawk. Last accessed May 14, 2023.

Listing 1: Simplified `MPI_Reduce` micro-benchmark using `MPIX_Harmonize` for process synchronization.

```
1  int check_flags[CHECK_EVERY];
2  int valid_iterations = 0;
3  double check_time[CHECK_EVERY];
4  double wtime = 0.0;
5  for (int i = 0; i < NITER; ++j) {
6    int flag, check_idx;
7    /* synchronize in space and time */
8    MPIX_Harmonize(comm, &flag);
9    /* take measurement */
10   double begin = MPI_Wtime();
11   MPI_Reduce(..., comm);
12   check_idx = i%CHECK_EVERY;
13   check_time[check_idx] = MPI_Wtime() - begin;
14   check_flags[check_idx] = flag;
15   if (i == NITER-1 ||
16       check_idx == CHECK_EVERY-1)) {
17     /* discard invalid timings */
18     MPI_Allreduce(MPI_IN_PLACE, check_flags,
19                   check_idx,
20                   MPI_INT, MPI_LAND, comm);
21     for (int k = 0; k < check_idx; ++k) {
22       if (check_flags[k]) { /* valid */
23         wtime += check_time[k];
24         valid_iterations++;
25       }
26     }
27   }
28   assert(valid_iterations >
         MIN_VALID_ITERATIONS);
29   double avg = wtime / valid_iterations;
30 }
```

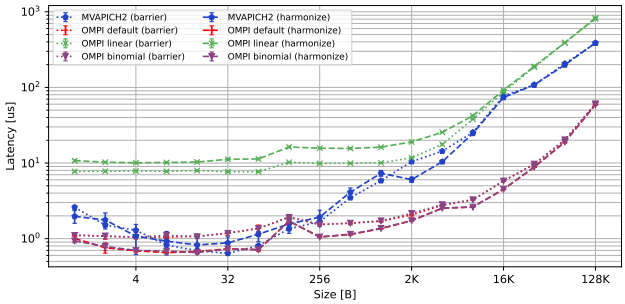
micro-benchmarks were used in version 7.0.1. We ran each benchmark ten times and present the mean and standard deviation as error bars. Each repetition included warm-up iterations.

For obtaining a virtual, global clock in MPI, we leverage the clock synchronization algorithms provided in ReproMPI [14]. In our experiments, we use a variant of the HCA3 clock synchronization algorithm which determines a process' current clock offset to the reference clock but which does not estimate clock drifts (as this would entail a larger overhead) [15].

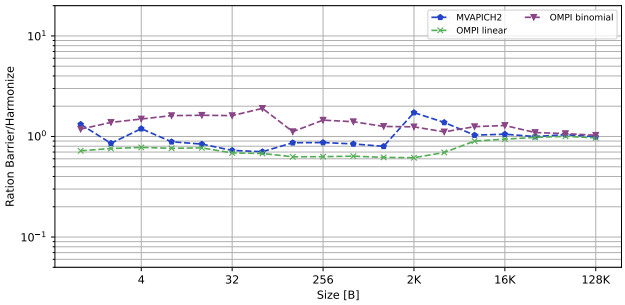
Open MPI provides many different implementations for the collective operations discussed below. However, in the interest of readability, we will focus only on a small, representative selection. We present data for MVAPICH2 (blue), the default selection in Open MPI (red), and a few specific implementations in Open MPI (other colors). We show measurements using `MPI_Barrier` as dotted lines and those using `MPIX_Harmonize` as dashed lines.

6.1 MPI_Reduce

Figure 6 shows the latency of different algorithms for `MPI_Reduce` in combination with different barrier algorithms and `MPIX_Harmonize` on 128 processes (single node). It becomes apparent that the use of



(a) Absolute values



(b) Ratio between `MPI_Barrier` and `MPIX_Harmonize`

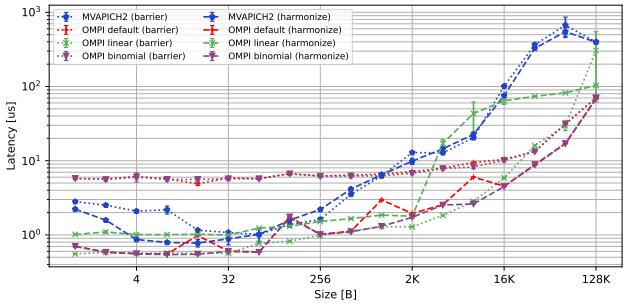
Figure 6: Message scaling of different implementations for `MPI_Reduce` on 128 processes.

`MPIX_Harmonize` alters the reported average latency slightly: while the `linear` implementation shows a higher average latency when synchronized with `MPIX_Harmonize` than when synchronized with a barrier, the `binomial` implementation shows a consistently lower latency using `MPIX_Harmonize`. We compute the ratio between the two synchronization methods as $\delta = \frac{t_{barrier}}{t_{harmonize}}$. Thus, for $\delta > 1$ using `MPIX_Harmonize` yielded a lower latency, while for $\delta < 1$, we saw a lower latency when measured with `MPI_Barrier`. As shown in Figure 6b for 128 processes, that range is 0.6–1.8.

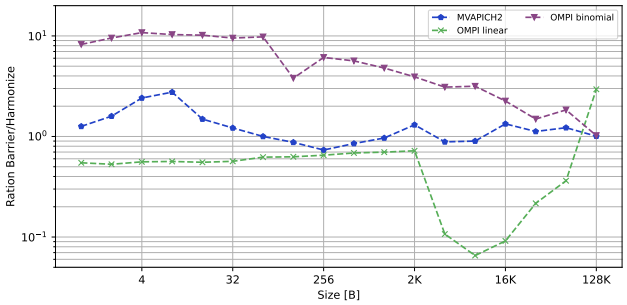
A more dramatic picture emerges on 16k processes (128 nodes; Figure 7): the `binomial` and `linear` implementations yield up to 10x lower and more then 10x higher average latency, respectively, when measured with `MPIX_Harmonize`. The implementation in MVAPICH2 seems to be somewhat less affected by the synchronization (2x lower latency for small messages).

Scaling the number of processes from 128 (single node) to 16k (128 nodes) in Figure 8 shows that using `MPIX_Harmonize` yields more stable measurements. The average and maximum latencies (Figure 8a and Figure 8b, respectively) show pronounced oscillations under barrier synchronization. Such an effect is not present using `MPIX_Harmonize`, for which the scaling curves appear more stable.

Choice of Metric. As a side-note, the scaling behavior of the `linear` algorithm demonstrates that the average latency is a rather questionable metric for benchmarking collective operations. An explanation for the average latency *decreasing* under increasing process counts is that at any scale, most processes are able to leave the collective procedure call immediately (due to eagerly sent messages),

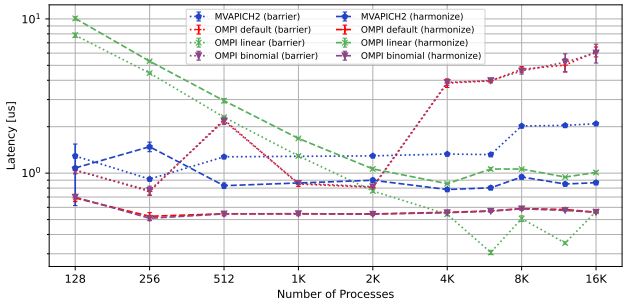


(a) Absolute values

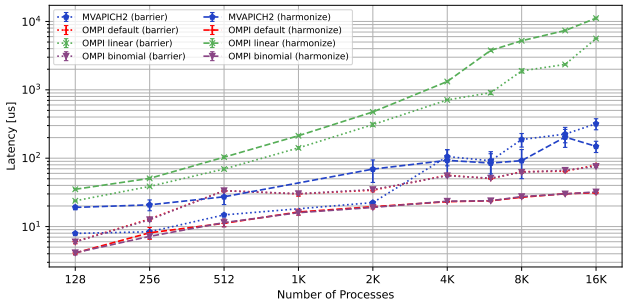


(b) Ratio between `MPI_Barrier` and `MPIX_Harmonize`

Figure 7: `MPI_Reduce` message scaling on 16k processes.

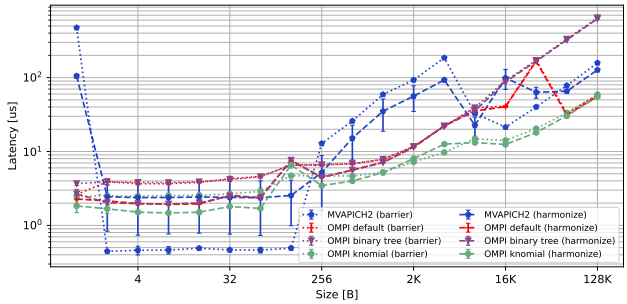


(a) Average latency

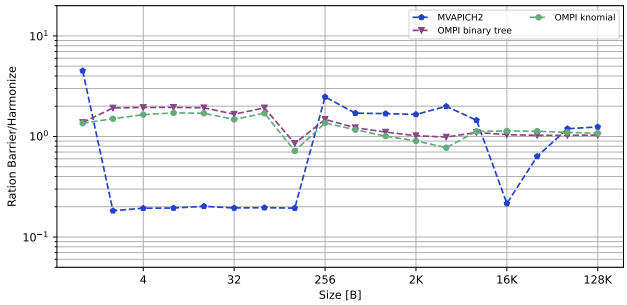


(b) Maximum latency

Figure 8: Process scaling for `MPI_Reduce` with 4B messages.



(a) Absolute values



(b) Relative differences

Figure 9: `MPI_Bcast` message scaling on 128 processes.

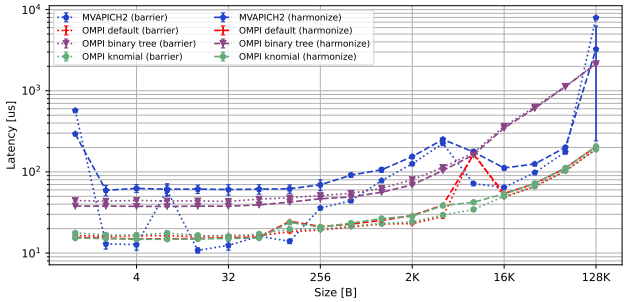
reducing the overall mean latency with their small contribution. The maximum latency, on the other hand, appears to be a better representative of the potential impact of the impact of a collective operation on a parallel application. Given the tightly coupled nature of MPI applications, the delay introduced by the process with the highest latency in an MPI collective operation will impact subsequent collective or point-to-point operations. This fact, however, is not reflected in any of the benchmarks due to the immediate and unaccounted synchronization between each operation.

6.2 `MPI_Bcast`

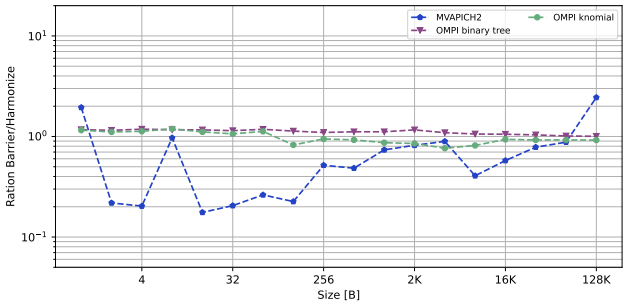
For `MPI_Bcast`, the data in Figure 9 shows a similar picture as for `MPI_Reduce`. Most notable, however, the MVAPICH2 implementation for small messages seems to be geared towards `MPI_Barrier`, yielding a 9x higher latency when using `MPIX_Harmonize` compared to `MPI_Barrier`. On the other hand, Open MPI (and MVAPICH2 in the mid-range sizes) generally show lower latencies with `MPIX_Harmonize`.

A similar picture for MVAPICH2 emerges at 16k processes (Figure 10). For Open MPI, the normalized differences are generally smaller at this scale and the relative order of two algorithms shown does not change. Thus, the use of `MPIX_Harmonize` would not change the selection heuristic in Open MPI on this system.

However, when scaling the number of processes (Figure 11) most implementations show significant variations across the process range when synchronized using `MPI_Barrier`, most notably the `knomial` implementation in Open MPI. Using `MPIX_Harmonize` yields stable results across the range.



(a) Absolute values



(b) Ratio between `MPI_Barrier` and `MPIX_Harmonize`

Figure 10: `MPI_Bcast` message scaling on 16k processes.

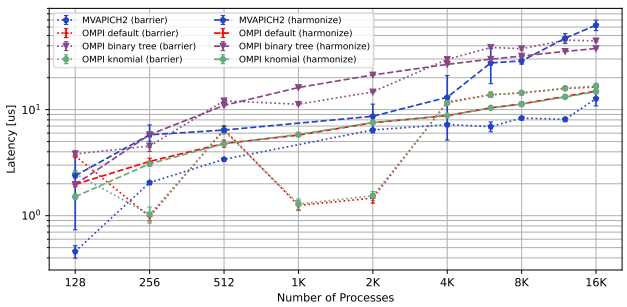
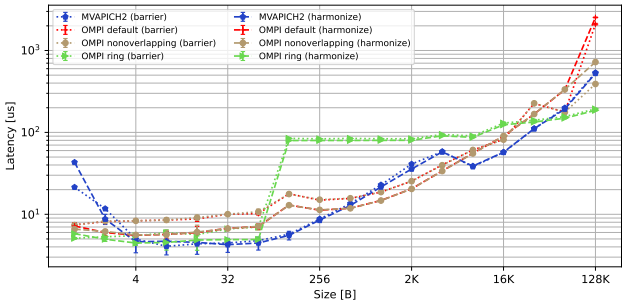


Figure 11: Process scaling for `MPI_Bcast` with 4B messages.

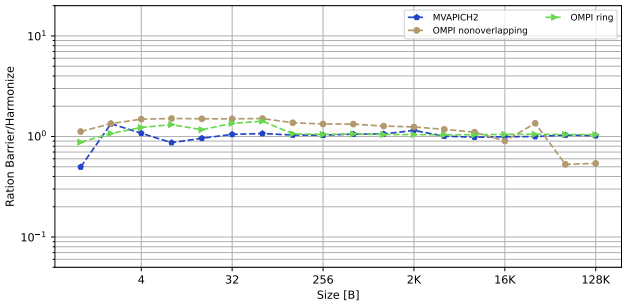
6.3 `MPI_Allreduce`

In contrast to the rooted `MPI_Reduce` and `MPI_Bcast` operations discussed above, the distortions introduced by the barrier synchronization on the results for `MPI_Allreduce` appear to be less significant at 128 processes (Figure 12). The most impacted implementation shown is `nonoverlapping` in Open MPI, which is a combination of `MPI_Reduce` and `MPI_Bcast`.

At 16k processes, however, we see a signification impact on both MVAPICH2 and the `nonoverlapping` implementation in Open MPI. Both implementations see a 5x higher latency with `MPIX_Harmonize` compared to `MPI_Barrier` at different points in the size range (ratio of 0.2 in Figure 13b). With `MPI_Barrier`, MVAPICH2 appears to yield lower average latencies, while with `MPIX_Harmonize` the default implementation in Open MPI yield lower latencies.

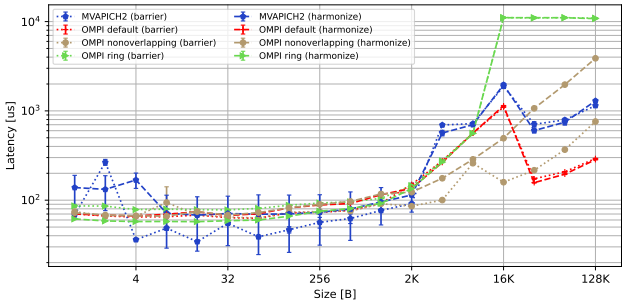


(a) Absolute values

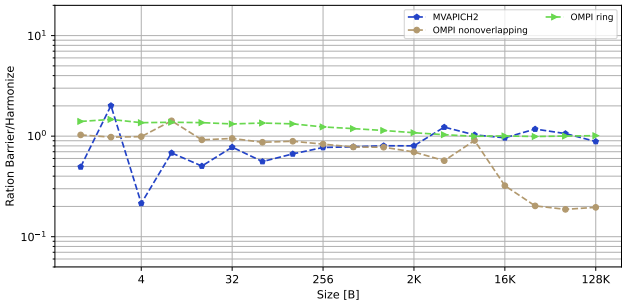


(b) Ratio between `MPI_Barrier` and `MPIX_Harmonize`

Figure 12: `MPI_Allreduce` message scaling on 128 processes.



(a) Absolute values



(b) Ratio between `MPI_Barrier` and `MPIX_Harmonize`

Figure 13: `MPI_Allreduce` message scaling on 16k processes.

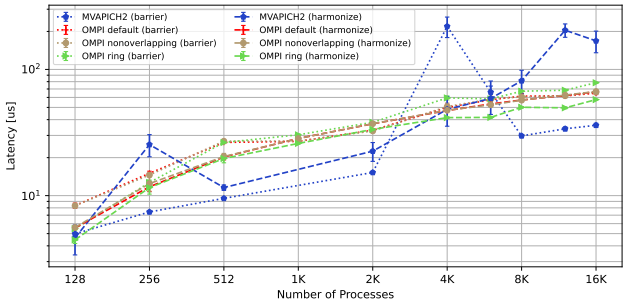


Figure 14: Process scaling for `MPI_Allreduce` with 4B messages.

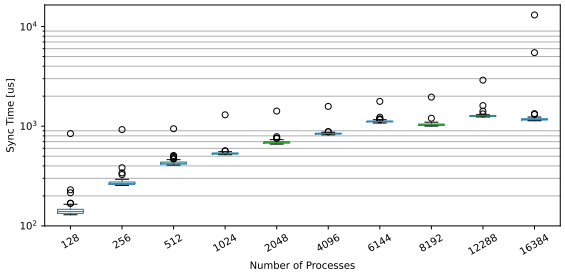


Figure 15: Duration of clock synchronization over several process counts with 128 processes per node.

This is also reflected in Figure 14, where at the lower and the higher end of the process range the comparison between Open MPI and MVAPICH2 tips in favor of Open MPI when using `MPIX_Harmonize` instead of `MPI_Barrier` for synchronization.

6.4 Cost of Synchronization

We measured the cost of clock synchronization at different process counts, i.e., the time the clock synchronization algorithm described in section 4 takes. Figure 15 shows that at small scales that cost is in the order of a few hundred microseconds and increases into the range of milliseconds. As described in subsection 5.2, we currently perform a clock synchronization if the last synchronization has been longer than 1 s ago. At 16k processes, the synchronization would thus incur less than 2 % overhead.

As can be seen in Figure 15, most of the 50 repetitions fall into a narrow band, with only a few outliers. A closer look at the data revealed that the outliers at each point appear at the first repetition, suggesting that they are caused by the MPI-internal connection setup. These outliers will happen during the warm-up phase and would not be of concern throughout the rest of the benchmark run.

6.5 Quality of Clock Synchronization

We have attempted to quantify the quality of the clock synchronization. Our benchmark performs a clock synchronization among all processes. We then sample 1 % of the processes and check their clock offset relative to the root process. This process is repeated 20 times. From the description in section 4, it is clear that processes with higher rank numbers will yield lower accuracy, given their distance in the graph from the root process, so we purposefully sample processes in the middle and high end of the process range.

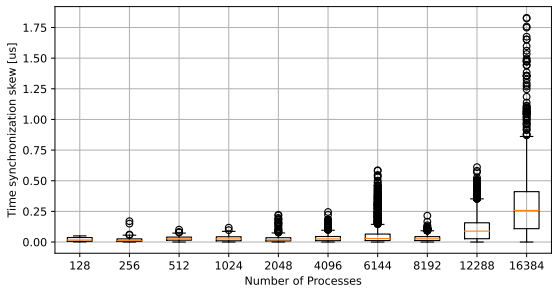


Figure 16: Synchronization accuracy over several process counts with 128 processes per node.

As can be seen, the mean synchronization accuracy stays well below 1 µs across all process numbers. While we found a significant number of outliers at larger scales, their accuracy is still below 2 µs, an order of magnitude lower than the collective operations and than the synchronization skew of a barrier (Figure 3).

6.6 Other Collective Algorithms

We have also conducted experiments on `MPI_Scatter` and `MPI_Gather` but have not found similar levels of differences between the two synchronization methods so we refrain from presenting the results.

7 CONCLUSIONS

In this paper, we have shown that the process synchronization in space typically employed by MPI and application benchmarks is insufficient and introduces artificial arrival patterns into the operation under test, introducing significant skew into measurements. In order to mitigate this issue and encourage wide-spread adoption, we propose an addition to the MPI standard—`MPIX_Harmonize`—that performs both synchronization in space and time, making sure that all processes in a group resume execution as close as possible in time. Our analysis shows that a synchronization accuracy around 1 µs can be achieved, which is a significant improvement over the use of `MPI_Barrier`. We have shown that benchmarks using this new synchronization primitive do not suffer from arrival pattern artifacts introduced by barriers, leading to more reliable benchmark results of MPI collective operations.

ACKNOWLEDGMENTS

This research was supported partly by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We gratefully acknowledge the provision of computational resources by the High Performance Computing Center (HLRS) at the University of Stuttgart, Germany. This work was partially supported by the Austrian Science Fund (FWF): project P 33884-N.

REFERENCES

[1] [n. d.]. Intel(R) MPI Benchmarks User Guide. https://software.intel.com/en-us/imb-user-guide Last accessed May 11, 2023.

[2] [n. d.]. mpiBench. https://github.com/LLNL/mpiBench Last accessed May 11, 2023.

[3] [n. d.]. Network Time Protocol (NTP). https://www.rfc-editor.org/rfc/rfc958.

[4] [n. d.]. OSU Micro Benchmarks. https://mvapich.cse.ohio-state.edu/benchmarks/ Last accessed May 11, 2023.

[5] Pedram Alizadeh, Amirhossein Sojoodi, Yiltan Hassan Temucin, and Ahmad Afsahi. 2022. Efficient Process Arrival Pattern Aware Collective Communication for Deep Learning. In *Proceedings of the 29th European MPI Users' Group Meeting* (Chattanooga, TN, USA) *(EuroMPI/USA'22)*. Association for Computing Machinery, New York, NY, USA, 68–78. https://doi.org/10.1145/3555819.3555857

[6] Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John C. Linford. 2009. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Comput.* 35, 12 (2009), 595–607. https://doi.org/10.1016/j.parco.2008.12.012

[7] BSC Performance Tools. 2022. Paraver. https://tools.bsc.es/paraver

[8] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. 2018. Characterization of MPI Usage on a Production Supercomputer. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 386–400. https://doi.org/10.1109/SC.2018.00033

[9] Jens Doleschal, Andreas Knüpfer, Matthias S. Müller, and Wolfgang E. Nagel. 2008. Internal Timer Synchronization for Parallel Event Tracing. In *EuroPVM/MPI (LNCS, Vol. 5205)*. Springer, 202–209.

[10] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. 2007. A Study of Process Arrival Patterns for MPI Collective Operations. In *Proceedings of the 21st Annual International Conference on Supercomputing* (Seattle, Washington) *(ICS '07)*. Association for Computing Machinery, New York, NY, USA, 168–179. https://doi.org/10.1145/1274971.1274996

[11] Hewlett Packard Enterprise. [n. d.]. Cray Performance Measurement and Analysis Tools User Guide. https://support.hpe.com/hpesc/public/docDisplay?docId=a00113915en_us&docLocale=en_US&page=Measure_MPI_Load_Imbalance.html Last accessed August 16, 2023.

[12] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Accurately measuring overhead, communication time and progression of blocking and non-blocking collective operations at massive scale. *IJPEDS* 25, 4 (2010), 241–258. https://doi.org/10.1080/17445760902894688

[13] Sascha Hunold and Alexandra Carpen-Amarie. 2015. On the Impact of Synchronizing Clocks and Processes on Benchmarking MPI Collectives. In *EuroMPI*. ACM, 8:1–8:10. https://doi.org/10.1145/2802658.2802662

[14] Sascha Hunold and Alexandra Carpen-Amarie. 2016. Reproducible MPI Benchmarking is Still Not as Easy as You Think. *IEEE Transactions on Parallel and Distributed Systems* 27, 12 (2016), 3617–3630. https://doi.org/10.1109/TPDS.2016.2539167

[15] Sascha Hunold and Alexandra Carpen-Amarie. 2018. Hierarchical Clock Synchronization in MPI. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 325–336. https://doi.org/10.1109/CLUSTER.2018.00050

[16] Terry Jones and Gregory A. Koenig. 2013. Clock synchronization in high-end computing environments: a strategy for minimizing clock variance at runtime. *CCPE* 25, 6 (2013), 881–897. https://doi.org/10.1002/cpe.2868

[17] Terry R. Jones, George Ostrouchov, Gregory A. Koenig, Oscar H. Mondragon, and Patrick G. Bridges. 2018. An evaluation of the state of time synchronization on leadership class supercomputers. *Concurr. Comput. Pract. Exp.* 30, 4 (2018). https://doi.org/10.1002/cpe.4341

[18] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91.

[19] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[20] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. 2015. PulseSync: An Efficient and Scalable Clock Synchronization Protocol. *IEEE/ACM Trans. Netw.* 23, 3 (2015), 717–727. https://doi.org/10.1109/TNET.2014.2309805

[21] Antonio Libri, Andrea Bartolini, Daniele Cesarini, and Luca Benini. 2018. Evaluation of NTP/PTP fine-grain synchronization performance in HPC clusters. In *Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems, ANDARE@PACT 2018, Limassol, Cyprus, November 4, 2018*, Andrea Bartolini, João M. P. Cardoso, and Cristina Silvano (Eds.). ACM, 3:1–3:6. https://doi.org/10.1145/3295816.3295819

[22] Nuria Losada, Patricia González, María J Martín, George Bosilca, Aurélien Bouteiller, and Keita Teranishi. 2020. Fault tolerance of MPI applications in exascale systems: The ULFM solution. *Future Generation Computer Systems* 106 (2020), 467–481.

[23] Xi Luo, Wei Wu, George Bosilca, Thananon Patinyasakdikul, Linnan Wang, and Jack Dongarra. 2018. ADAPT: An Event-Based Adaptive Collective Communication Framework. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. Association for Computing Machinery, New York, NY, USA, 118–130. https://doi.org/10.1145/3208040.3208054

[24] Xi Luo, Wei Wu, George Bosilca, Yu Pei, Qinglei Cao, Thananon Patinyasakdikul, Dong Zhong, and Jack Dongarra. 2020. HAN: a Hierarchical AutotuNed Collective Communication Framework. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 23–34. https://doi.org/10.1109/CLUSTER49012.2020.00013

[25] Teng Ma, George Bosilca, Aurelien Bouteiller, Brice Goglin, Jeffrey M. Squyres, and Jack J. Dongarra. 2011. Kernel Assisted Collective Intra-node MPI Communication among Multi-Core and Many-Core CPUs. In *2011 International Conference on Parallel Processing*. 532–541. https://doi.org/10.1109/ICPP.2011.29

[26] MPI v4.0 2021. *MPI: A Message-Passing Interface Standard, Version 4.0*. Technical Report. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[27] Wolfgang E Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. 1996. VAMPIR: Visualization and analysis of MPI resources. (1996).

[28] Emin Nuriyev, Juan-Antonio Rico-Gallego, and Alexey L. Lastovetsky. 2022. Model-based selection of optimal MPI broadcast algorithms for multi-core clusters. *J. Parallel Distributed Comput.* 165 (2022), 1–16. https://doi.org/10.1016/j.jpdc.2022.03.012

[29] Pitch Patarasuk and Xin Yuan. 2008. Efficient MPI Bcast across different process arrival patterns. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–11. https://doi.org/10.1109/IPDPS.2008.4536308

[30] Ivy Bo Peng, Stefano Markidis, and Erwin Laure. 2015. The Cost of Synchronizing Imbalanced Processes in Message Passing Systems. In *2015 IEEE International Conference on Cluster Computing*. 408–417. https://doi.org/10.1109/CLUSTER.2015.63

[31] Jelena Pješivac-Grbović, George Bosilca, Graham E Fagg, Thara Angskun, and Jack J Dongarra. 2007. MPI collective algorithm selection and quadtree encoding. *Parallel Comput.* 33, 9 (2007), 613–623.

[32] Jerzy Proficz, Piotr Sumionka, Jarosław Skomiał, Marcin Semeniuk, Karol Niedzielewski, and Maciej Walczak. 2020. Investigation into MPI All-Reduce Performance in a Distributed Cluster with Consideration of Imbalanced Process Arrival Patterns. In *Advanced Information Networking and Applications*, Leonard Barolli, Flora Amato, Francesco Moscato, Tomoya Enokido, and Makoto Takizawa (Eds.). Springer International Publishing, Cham, 817–829.

[33] Thomas Schmid, Zainul Charbiwala, Jonathan Friedman, Young H. Cho, and Mani B. Srivastava. 2008. Exploiting Manufacturing Variations for Compensating Environment-Induced Clock Drift in Time Synchronization. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Annapolis, MD, USA) *(SIGMETRICS '08)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/1375457.1375469

[34] Kawthar Shafie Khorassani, Chen Chun Chen, Bharath Ramesh, Aamir Shafi, Hari Subramoni, and Dhabaleswar Panda. 2022. High Performance MPI over the Slingshot Interconnect: Early Experiences. In *Practice and Experience in Advanced Research Computing* (Boston, MA, USA) *(PEARC '22)*. Association for Computing Machinery, New York, NY, USA, Article 15, 7 pages. https://doi.org/10.1145/3491418.3530773

[35] Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. 2021. Understanding the use of Message Passing Interface in Exascale Proxy Applications. *Concurrency and Computation: Practice and Experience* 33, 14 (2021), e5901. https://doi.org/10.1002/cpe.5901 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5901

[36] Jesper Larsson Träff and Sascha Hunold. 2020. Decomposing MPI Collectives for Exploiting Multi-lane Communication. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 270–280. https://doi.org/10.1109/CLUSTER49012.2020.00037

[37] Thomas Worsch, Ralf H. Reussner, and Werner Augustin. 2002. On Benchmarking Collective MPI Operations. In *EuroMPI/PVM (LNCS, Vol. 2474)*. Springer, 271–279. https://doi.org/10.1007/3-540-45825-5_43

[38] Christopher Zimmer, Scott Atchley, Ramesh Pankajakshan, Brian E. Smith, Ian Karlin, Matthew L. Leininger, Adam Bertsch, Brian S. Ryujin, Jason Burmark, André Walker-Loud, M. A. Clark, and Olga Pearce. 2019. An Evaluation of the CORAL Interconnects. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 39, 18 pages. https://doi.org/10.1145/3295500.3356166