Scalable Computing: Practice and Experience
Volume 10, Number 1, pp. 49–60. http://www.scpe.org
ISSN 1895-1767

REPLAY-BASED SYNCHRONIZATION OF TIMESTAMPS IN EVENT TRACES OF MASSIVELY PARALLEL APPLICATIONS

DANIEL BECKER*, JOHN C. LINFORD†, ROLF RABENSEIFNER‡, AND FELIX WOLF*

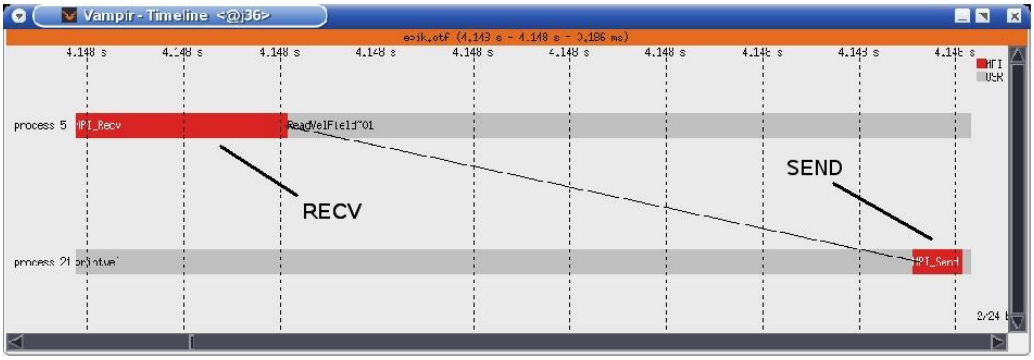
Abstract. Event traces are helpful in understanding the performance behavior of message-passing applications since they allow in-depth analyses of communication and synchronization patterns. However, the absence of synchronized hardware clocks may render the analysis ineffective because inaccurate relative event timings can misrepresent the logical event order and lead to errors when quantifying the impact of certain behaviors. Although linear offset interpolation can restore consistency to some degree, inaccuracies and time-dependent drifts may still disarrange the original succession of events—especially during longer runs. In our earlier work, we have presented an algorithm that removes the remaining violations of the logical event order postmortem and, in addition, have outlined the initial design of a parallel version. Here, we complete the parallel design and describe its implementation within the Scalasca trace-analysis framework. We demonstrate its suitability for large-scale applications running on more than thousand application processes and evaluate its accuracy by showing that it eliminates inconsistent inter-process timings while preserving the length of local intervals.

1. Introduction. Event tracing is a popular technique for the postmortem performance analysis of message-passing applications because it can be used to investigate temporal relationships between concurrent activities. Obviously, the accuracy of the analysis depends on the comparability of timestamps taken on different processors. Inaccurate timestamps may cause a given interval to appear shorter or longer than it actually was, or change the logical event order, which requires a message to be received only after it has been sent. This is also referred to as the *clock condition*. Besides leading to false conclusions during performance analysis when the impact of certain behaviors is quantified, clock condition violations may confuse the user of trace visualization tools such as Vampir [30] by causing arrows representing messages to point backward in time-line views (Figure 1.1(a)). Moreover, automatic trace-analysis tools such as Scalasca [19] may produce not only inaccurate but outright inconsistent results when calculating certain global metrics that rely on the correct logical order of message events. For example, the output of Scalasca shown in Figure 1.1(b) suggests that the difference between the total time spent in barrier synchronization (i. e., collective synchronization) and the time spent in the barrier before and immediately after the actual synchronization has taken place is negative, which cannot be true.

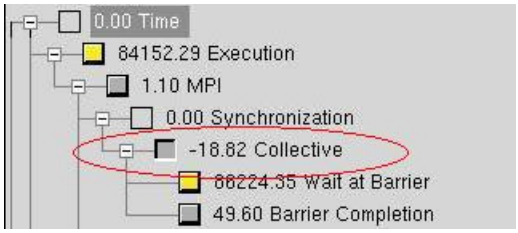
To avoid clock-condition violations, the error of timestamps should ideally be smaller than one half of the message latency. A typical clock quartz with a drift of only 1 min/year will cause a deviation of 2 μs already after 1 second. While some systems such as IBM Blue Gene offer a relatively accurate global clock, many other systems including most PC clusters provide only processor-local clocks that are either entirely non-synchronized or synchronized only within disjoint partitions (e.g., SMP-node or multicore-chip). Clock synchronization protocols such as NTP [29] can align the clocks to a certain degree, but are often not accurate enough for our purposes. Assuming that every local clock on a parallel machine runs at a different but constant speed (i. e., with a different but constant drift), the (global) time of an arbitrarily chosen master clock can be calculated locally as a linear function of the local time. However, as the assumption of constant drifts is only an approximation, violations of the clock condition may still occur - especially when the offset measurements needed for the interpolation are taken with long intervals in between. For example, temperature variations may cause variations of the drift rate of more than 10^{-8}, which will cause synchronization errors of more than 1 μs after 100 seconds execution time. Figure 1.2 shows clock offsets after linear end-to-end interpolation measured using a simple benchmark program that was executed for 500 seconds. One can see that the non-linearity of local clocks caused clock errors larger than the send-receive and allreduce latency.

While the errors of single timestamps are hard to assess, clock-condition violations can be easily detected and offer a toehold to increase the fidelity of inter-process timings. In our earlier work [3], we have presented an algorithm that retroactively corrects timestamps violating the clock condition in event traces of MPI applications. However, in view of rapidly increasing parallelism combined with advances in scalable trace-analysis technology [5, 19, 6], it is crucial that the algorithm scales to large numbers of application processes. For

*Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany; Department of Computer Science, RWTH Aachen University, 52056 Aachen, Germany, {d.becker, f.wolf}@fz-juelich.de
†Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, USA, jlinford@vt.edu
‡High-Performance Computing Center, University of Stuttgart, 70550 Stuttgart, Germany, rabenseifner@hlrs.de



(a) Time-line visualization of a message exchange in backward direction.



(b) A global time metric that must not be negative.

FIG. 1.1. *Inconsistent trace analyses due to insufficiently synchronized timestamps.*

this reason, we have designed and implemented a parallel version of the algorithm and integrated it into the Scalasca performance-analysis framework [1, 18]. Instead of sequentially processing a single global trace file, we follow Scalasca's scalable trace-analysis approach [19] and process separate process-local trace files in parallel by *replaying* the original communication on as many CPUs as have been used to execute the target application itself. Since trace processing capabilities (i. e., processors and memory) grow proportionally with the number of application processes, we can achieve good scalability at very large scales.

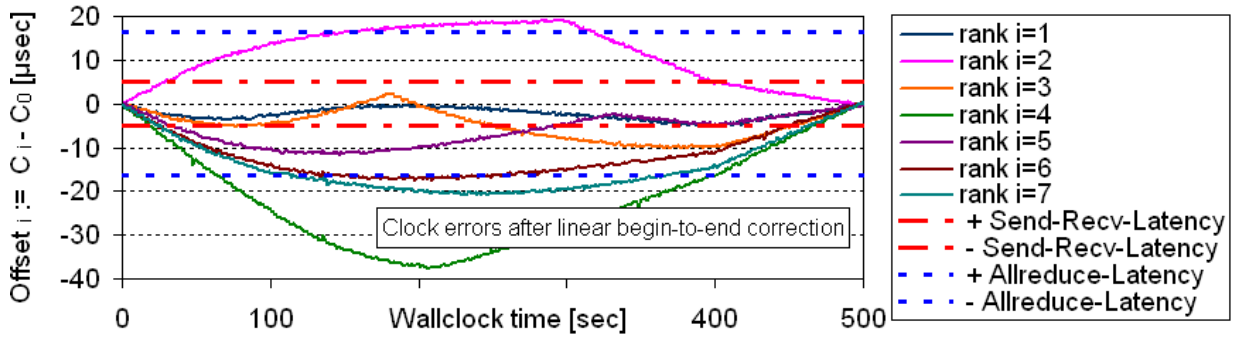


FIG. 1.2. *Non-linear offsets of physical clocks measured on an Infiniband cluster in comparison to the send-receive and allreduce latency.*

The outline of this article is as follows: After reviewing related work in Section 2, we briefly describe the serial version of the algorithm in Section 3. In Section 4, we complete the parallel design outlined in [3] and explain its implementation within the Scalasca trace-analysis infrastructure. We evaluate the scalability of the parallel version in Section 5, where we also show that the collaterally introduced deviations of local interval lengths remain within acceptable limits. Finally in Section 6, we conclude our paper and give an outlook on future work.

2. Related Work. This article describes an approach to retroactively synchronizing timestamps in event traces of parallel programs for the purpose of accurate program observation. While a detailed introduction to

event-based program observation as discussed here can be found in Riek et al. [35], we restrict our account of related work to the topic of clock synchronization.

Network-based synchronization protocols aim at synchronizing distributed clocks before reading them. Clocks distributed across the network query the global time from reference clocks, which are often organized in a hierarchy of servers. For instance, NTP [29] uses widely accessible and already synchronized primary time servers. Secondary time servers and clients can query time information via both private networks and the Internet. To reduce network traffic, the time servers are accessed only at regular intervals to adjust the local clock. Jumps are avoided by changing the drift while leaving the actual time unmodified. Unfortunately, varying network latencies limit the accuracy of NTP to about one millisecond compared to a few microseconds required to satisfy the clock condition for MPI applications running on clusters equipped with modern interconnect technology.

Time differences among distributed clocks can be characterized in terms of their relative offset and drift (i. e., the rate at which the offset changes over time). In a simple model assuming different but constant drifts, global time can be established by measuring offsets to a designated master clock using Cristian's probabilistic remote clock reading technique [7]. After estimating the drift, the local time can be mapped onto the global (i. e., master) time via linear offset interpolation. Offset values among participating clocks are measured either at program initialization [13] or at initialization and finalization, and are subsequently used as parameters of the linear correction function [22, 27]. So as not to perturb the program, offset measurements in between are usually avoided, although a recent approach proposes periodic offset measurements during global synchronization operations while limiting the effort required in each step by resorting to indirect measurements across several hops [10]. While linear offset interpolation might prove satisfactory for short runs (or interpolation intervals), measurement errors and time-dependent drifts may create inaccuracies and clock-condition violations during longer runs. Additionally, repeated drift adjustments caused by NTP may impede linear interpolation, as they deliberately introduce non-constant drifts.

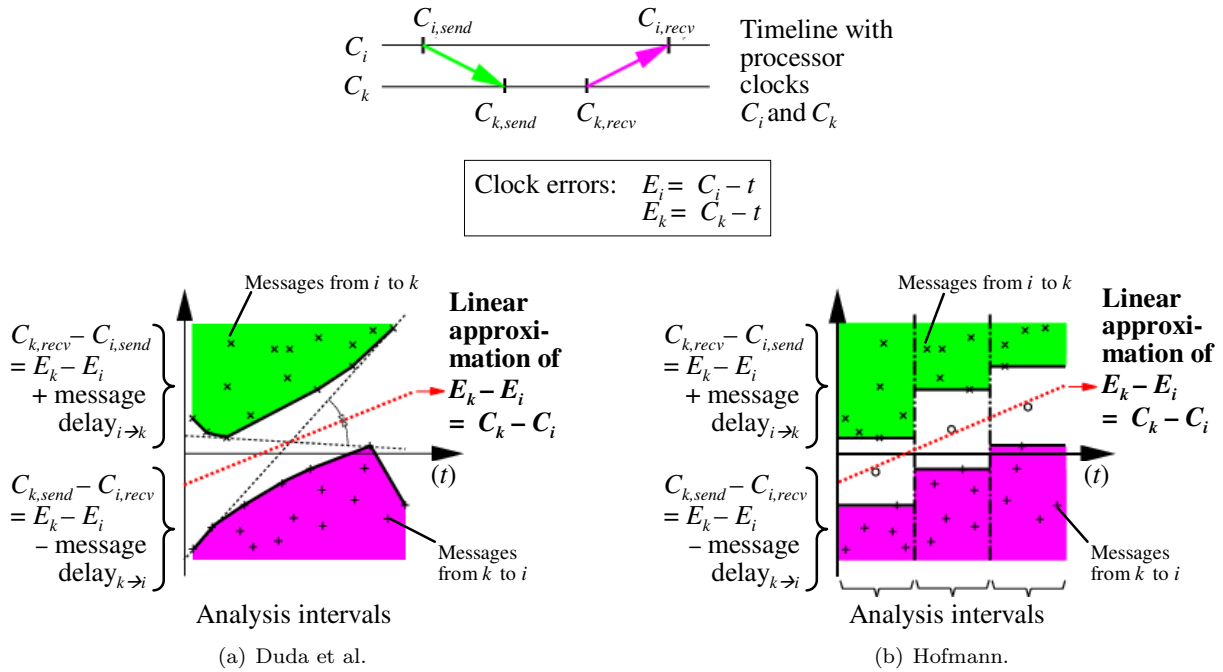


FIG. 2.1. *Algorithms that calculate the clock errors through the differences of the message transfer time in both directions between two processes.*

If linear interpolation alone turns out to be inadequate to achieve the desired level of accuracy, error estimation allows the retroactive correction of clock values in event traces after assessing synchronization errors among all distributed clock pairs. First, difference functions among clock values are calculated from the difference between clock values of receive and send events (plus the minimum message latency). Second, a medial smoothing function can be found and used to correct local clock values because for each clock pair two difference functions

exist. Regression analysis and convex hull algorithms have been proposed by Duda et al. [12] to determine the smoothing function. Using a minimal spanning tree algorithm, Jezequel [25] adopted Duda's algorithm for arbitrary processor topologies. In addition, Hofmann [21] improved Duda's algorithm using a simple minimum/maximum strategy and further proposed that the execution time should be divided into several intervals to compensate for different clock drifts in long running applications. Figure 2.1 shows the principles underlying Duda's and Hofmann's algorithms. Using a graph-theory algorithm to calculate the shortest paths, Hofmann and Hilgers [23] simplified Jezequel's algorithm for handling multi-processor topologies. A modification aimed at handling cases of non-existing communication relations between some of the application processes is described in [33]. Biberstein et al. [4] rewrote Hofmann's and Hilgers' algorithm for use on the Cell BE architecture using a short and intelligible notation. Their version solves the clock condition problem only for short intervals (i. e., without splitting into sub-intervals for handling a non-linear drift of the physical clocks). Babaoğlu and Drummond [2, 11] have shown that clock synchronization is possible at minimal cost if the application makes a full message exchange between all processors at sufficiently short intervals. However, jitter in message latency, nonlinear relations between message latency and message length, and one-sided communication topologies limit the usefulness of error estimation approaches. References to additional error estimation approaches can be found in a survey by Yang and Marsland [37].

In contrast, logical synchronization uses happened-before relations among send and receive pairs to synchronize distributed clocks. Lamport introduced a discrete logical clock [26] with each clock being represented as a monotonically increasing software counter. As local clocks are incremented after every local event and the updated values are exchanged at synchronization points, happened-before relations can be exploited to further validate and synchronize distributed clocks. If a receive event appears before its corresponding send event, that is, if a *clock-condition violation* occurs, the receive event is shifted forward in time according to the clock value exchanged. Lamport's discrete logical clock [26] can be used directly for monitoring [8]. Moreover, an algorithm to prevent the drift between the logical clocks has been proposed by Raynal [34]. As an enhancement of Lamport's discrete logical clock, Fidge [15, 16] and Mattern [28] proposed a vector clock. In their scheme, each processor maintains a vector representing all processor-local clocks. While the local clock is advanced with each local event as before, the local vector is updated after receiving a message using an element-wise maximum operation between the local vector and the remote vector that has been sent along with the message. The vector clock is used in some monitoring tools [9, 14] and, in a modified form, also to distinguish in event traces between primary wait states and secondary ones that are merely caused by propagation [24]. Furthermore, global events are introduced in [20], while in [31] spontaneous events (e.g. collisions on a network) are taken into account. Finally, limits of the logical clock and the vector clock are illustrated in [36].

3. Controlled Logical Clock. The *controlled logical clock* (CLC) algorithm by Rabenseifner [32, 33] retroactively corrects clock condition violations in event traces of message-passing applications by shifting message events in time while trying to preserve the length of intervals between local events. The algorithm restores the clock condition using happened-before relations derived from message semantics. The clock condition, given in Equation (3.1), requires that a receive event occurs at the earliest l_{min} after the matching send event, with l_{min} being the minimum message latency.

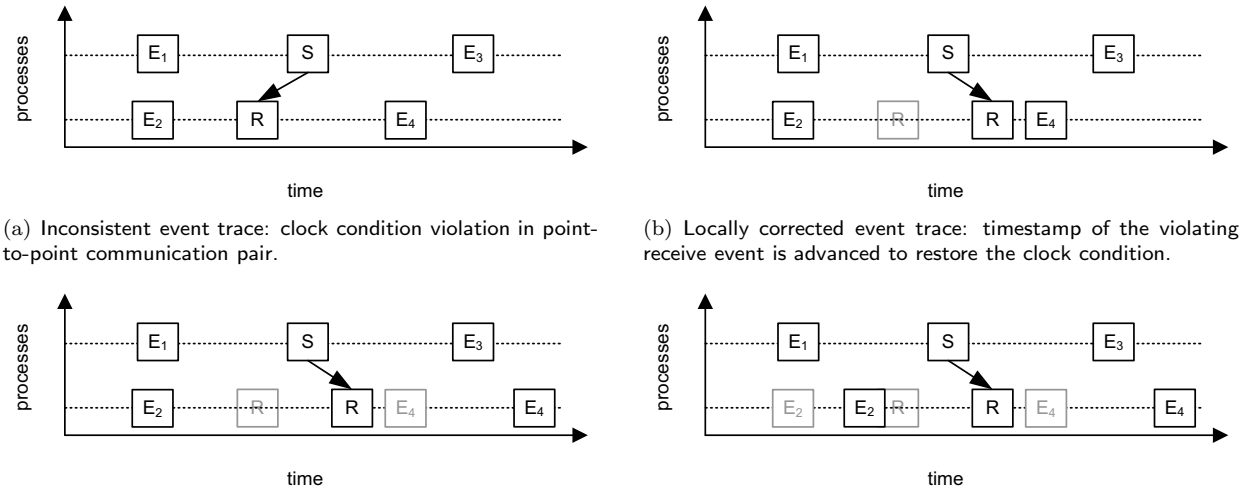
$$t_{recv} \geq t_{send} + l_{min} \tag{3.1}$$

If the condition is violated for a send-receive event pair, the receive event is moved forward in time. To preserve the length of intervals between local events, events following or immediately preceding the corrected event are moved forward as well. These adjustments are called forward and backward amortization, respectively. Note that the accuracy of the adjustment depends on the accuracy of the original timestamps. Therefore, the algorithm benefits from weak pre-synchronization such as the aforementioned linear offset interpolation.

Figure 3.1 illustrates the different steps of the CLC algorithm using a simple example consisting of two processes exchanging a single message. The subfigures show the time lines of the two processes along with their send (S) and receive (R) events, each of them enclosed by two other events (E_i). Figure 3.1(a) shows the initial event trace based on timestamps measured with insufficiently synchronized local clocks. The trace exhibits a violation of the clock condition by having the receive event appear earlier than the matching send event. To restore the clock condition, R is moved forward in time to be l_{min} ahead of S (Figure 3.1(b)). Since now the distance between R and E_4 becomes too short, E_4 is adjusted during the forward amortization to preserve the length of the interval between the two events (Figure 3.1(c)). However, the jump discontinuity introduced

by adjusting R affects not only events later than R but also events earlier than R. This is corrected by the backward amortization which shifts E_2 closer to the new position of R, see Figure 3.1(d).

While the forward amortization is at least initially applied to all events following R, the backward amortization applies a linearly increasing correction to a limited amortization interval before R. However, in order to avoid new violations of the clock condition, the correction must not advance any send event located in this interval farther than the matching receive event (minus the minimum message latency). In such a case, we apply the linear correction piecewise, advancing the send events as far as possible and calculating a different slope for each subinterval before, after, or between those sends [3, 33].



(a) Inconsistent event trace: clock condition violation in point-to-point communication pair.

(b) Locally corrected event trace: timestamp of the violating receive event is advanced to restore the clock condition.

(c) Forward amortized event trace: event E_4 following the receive event is adjusted to preserve the length of the interval between both events.

(d) Backward amortized event trace: event E_2 preceding the receive event is advanced to smooth the jump.

FIG. 3.1. *Backward and forward amortization in the controlled logical clock algorithm.*

Note that the algorithm only moves events forward in time. To prevent an increase of the overall time represented by the trace that may occur as a result of a domino-style propagation of forward amortizations, the algorithm applies scaling factors (i. e. control variables) to ensure that the overall error remains within predefined boundaries. The CLC algorithm always tries to advance all processor clocks to the fastest clock when correcting the non-linearity of the clocks. Given that the original timestamps may be logically wrong, this correction leads to logically correct timestamps with marginal local inaccuracies. As a result, timestamp differences between events on different processes normally become more accurate than the original ones because the clocks are advanced to the fastest one. In comparison, the aforementioned algorithms of Duda, Hofmann, and colleagues align the timestamps with the average of the local clocks. However, for monitoring purpose this difference is not significant because it is in the range of the drift rates among local clocks (i.e, in the range of about $10^{-6} - 10^{-4}$). Combined with linear offset interpolation between program start and end, the expected differences are in the range of 10^{-8}.

Since the original (CLC) algorithm takes only point-to-point messages into account, it has been extended in our earlier work [3] to make it applicable to realistic MPI applications that perform not only point-to-point but also collective communication. In our event model, a collective operation instance consists of multiple pairs of enter and exit events (i. e., one pair for each participating process). The basic idea behind extending the CLC algorithm to collective communication is to map collective communication onto point-to-point communication. For this purpose, we consider a single collective operation as a composition of multiple point-to-point operations, taking the semantics of the different flavors of MPI collective operations into account (e.g. *1-to-N*, *N-to-1*, etc.). For instance, in an *N-to-1* operation one root process receives data from N other processes. Given that the root process is not allowed to exit the operation before it has received data from the last process to enter the operation, the clock condition must be observed between the last enter event and the exit event of the root process. Depending on the flavor of the collective operation, different enter and exit events are mapped onto send and receive events, respectively. In reference to the fact that our method is based on logical clocks, we call

the send and receive event type assigned during this mapping the *logical event type* as opposed to the actual event type (e.g., enter or collective exit) specified in the event trace.

Until recently, only a serial implementation of the original (CLC) algorithm existed. In the next section, we describe how the extended version of the algorithm has been parallelized and how the parallel version has been integrated into the Scalasca trace-analysis framework.

4. Parallel Timestamp Synchronization. Scalasca, which has been specifically designed for large-scale systems, scans event traces of parallel applications for wait states that occur when processes fail to reach synchronization points in a timely manner, for example, as a result of an unevenly distributed workload. Such wait states can present severe challenges to achieving good performance, especially when trying to scale communication-intensive applications to large processor counts. As a first step towards reducing their impact, Scalasca provides a diagnostic method that allows their localization, classification, and quantification, in particular at larger scales. Scalability is achieved by analyzing the process-local traces in parallel, making Scalasca a parallel program in its own right.

Similar to the wait-state analysis [19] performed by Scalasca, the CLC algorithm requires comparing events involved in the same communication operation, which makes it a suitable candidate for the same parallelization strategy. Instead of sequentially processing a single global trace file, Scalasca processes separate process-local trace files in parallel by *replaying* the original communication on as many CPUs as have been used to execute the target application itself. Since trace processing capabilities (i. e., processors and memory) grow proportionally with the number of application processes, we can achieve good scalability at very large scales. During the replay, sending and receiving processes exchange relevant information needed to analyze the communication operation being replayed. The parallel CLC algorithm is divided into two replay phases: a forward phase for the forward amortization and a backward phase for the backward amortization. The backward phase is only needed if clock condition violations appear during the forward phase.

4.1. Integration with Scalasca. Almost all the postmortem trace-analysis functionality of Scalasca including the parallel CLC algorithm is implemented on top of PEARL [17], a parallel library that offers higher-level abstractions to read and analyze large volumes of trace data. A typical PEARL application is a parallel program having as many processes as the target application had that generated the trace data, resulting in a one-to-one mapping of target application and analysis processes. All analysis processes read the trace data of "their" application process into main memory and traverse the traces in parallel while exchanging information at synchronization points.

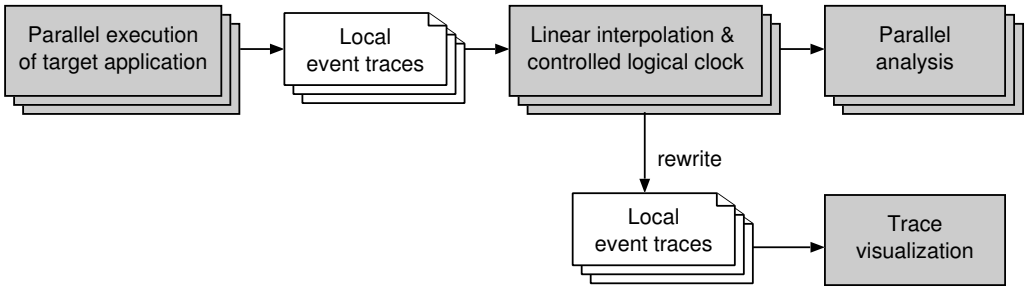


FIG. 4.1. *Parallel trace-analysis process. Grey rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs or files running or being processed in parallel.*

In Scalasca, the parallel CLC algorithm is applied after the traces have been loaded and before the wait-state analysis takes place. To increase the fidelity of the CLC outcome, the timestamps first undergo a pre-synchronization step. This step performs linear offset interpolation based on offset measurements taken during initialization and finalization of the target application. Once the offset values are known to each analysis process, the interpolation operation is performed locally and does not require any further communication. As an alternative to the native Scalasca wait state analysis, the traces can also be rewritten with modified timestamps, converted, and visualized using Vampir. The full analysis process is illustrated in Figure 4.1.

4.2. Forward Amortization. During the forward phase, the communication replay proceeds in the same direction as it did in the target application. For every pair of logical send and receive events, the sending process

sends the timestamp of the logical send event to the receiving process, which compares it to the timestamp of the matching logical receive event (minus the minimum message latency) and, if necessary, applies the forward-amortization equation described in [3]. Recall that, in addition to actual send and receive events, events pertaining to entering or leaving collective communication operations may be classified as logical send or receive events. In this case, the logical event type is derived from the name of the collective operation and the role (e.g., root) a particular process plays in the operation.

In its treatment of events the algorithm distinguishes between (logical) send or receive events and *internal* events that neither send nor receive any kind of message. A different action is performed for each of the three types. Since the correction of an internal event does not require any extra communication, the timestamp adjustment is immediately applied. A send event is adjusted locally and the new timestamp is sent via forward-replay to the receiving process. On the receiver side, the order of these two steps is reversed. The adjusted send timestamp must be obtained from the sender, before the correction can be performed. Finally, the receiver saves detected clock condition violations temporarily along with the associated error so that this information can be reused during the backward amortization phase.

While the direction of inter-process exchange of timestamps is determined by the (logical) type of an event (i. e., send or receive), the actual communication operation invoked to accomplish the transfer depends on the operation originally used by the target application. For this purpose, communication operations are classified according to the number of peers involved on either side: point-to-point, 1-to-N, N-to-1, N-to-N, and two special classes for scan and exscan operations. In brief, point-to-point operations are replayed using point-to-point communication, while collective operations are replayed using different flavors of collective communication.

For the sake of simplicity, our current implementation uses two different values for the minimum message latency l_{min} (see Equation (3.1)): the minimum inter-node and the minimum intra-node latency. Following a conservative approach aimed at avoiding overcorrection, we refrained from considering an extra collective latency, as the duration of collective operations may depend on many factors that are hard to identify, some of them even hidden inside the underlying MPI implementation. Thus, the algorithm requires exchanging the timestamps and the node identifiers to know which of the two latency values must be used.

As mentioned earlier, the CLC algorithm uses so-called control variables. Control variables are scaling factors that are applied to interval expressions when calculating new event timestamps with the purpose of preserving the length of local intervals and avoiding an avalanche-like propagation of corrections [3]. Usually, the control variables are kept less than 1 minus the maximal drift of the clocks. To determine their exact value, however, a global view of the trace data is needed, which is too expensive to establish in our parallel scheme as global communication would be required for every single event. Instead, we approximate a (single) suitable value for all control variables by performing multiple passes of forward replay through the trace data until the maximum error is below a predefined threshold. In practice, more than one pass is seldom needed.

4.3. Backward Amortization. The purpose of the backward amortization phase is to smooth jump discontinuities introduced during the forward amortization by slowly building up the ascension to the jump. This is achieved by applying a process-local linear correction to the interval immediately preceding the jump. However, to preserve the clock condition, the algorithm must not advance the timestamp of any send event located in this interval farther than that of the matching receive event (minus the minimum message latency), leading to the piecewise linear interpolation mentioned earlier. In addition to what has already been stated in our initial design [3], determining these upper limits requires a backward replay, starting at the end of the trace with communication proceeding in backward direction to avoid the danger of deadlocks. While replaying the communication backward, we store with each logical send event the timestamp of the matching receive event after forward amortization. With this information available, an appropriate piecewise linear interpolation function can be calculated for the amortization interval behind every receive event shifted during the forward replay. Note that during the backward amortization the roles of sender and receiver are reversed: the timestamp of a receive event must be available at the process of the matching send event.

4.4. Remark. Given that most MPI implementations use binomial tree algorithms to perform their collective operations, our replay-based approach reduces the communication complexity of replaying collectives automatically to $\mathcal{O}(\log N)$. Moreover, the stepwise parallel replay during the backward amortization phase can, in theory, be replaced by a single collective operation per communicator for the entire trace, but would impose impractical memory requirements. For the actual operations used during both replay phases and the timestamps being exchanged, please refer to [3].

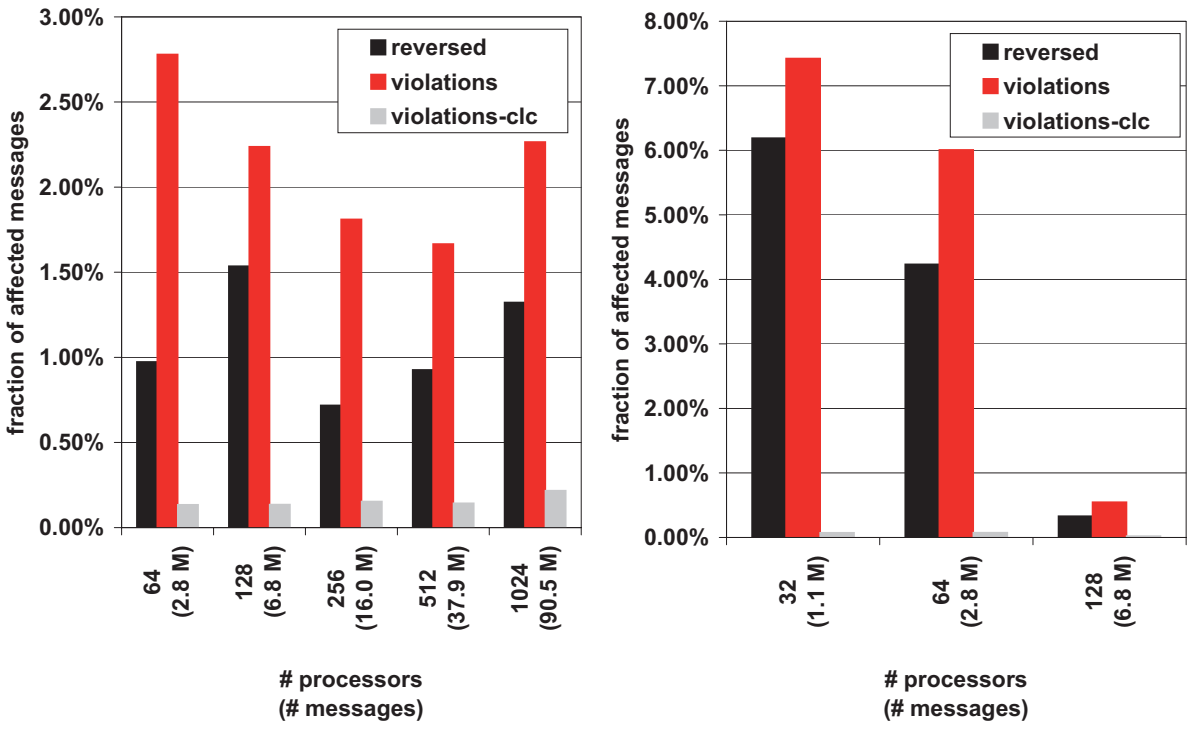
5. Experimental Evaluation. Here we evaluate the scalability and accuracy of the parallel controlled logical clock algorithm and also give evidence of the frequency and the extent of clock condition violations in event traces of a realistic MPI application. We ran experiments on the following two platforms:

MareNostrum consists of 2560 JS21 blade computing nodes, each with 2 dual-core IBM 64-bit PowerPC 970MP processors running at 2.3 GHz. The computing nodes of MareNostrum communicate primarily through a Myrinet network with Myrinet adapters integrated on each server blade. The measured MPI inter-node latency was 7.7μs, the measured MPI intra-node latency was 1.3μs.

CACAU consists of 200 compute nodes, each with 1 dual-core Intel Xeon EM64T CPU running at 3.2GHz. The nodes are linked with a Voltaire Infiniband Network and a Gigabit Ethernet. The measured MPI inter-node latency was 4.7μs, the measured MPI intra-node latency was 1.0μs.

As a test application, we used the MPI version of the ASC SMG2000 benchmark, a parallel semi-coarsening multigrid solver that uses a complex communication pattern and performs a large number of non-nearest-neighbor point-to-point communication operations. Applying a weak scaling strategy, a fixed $16 \times 16 \times 8$ problem size per process with five solver iterations was configured.

While linear interpolation can remove most of the clock condition violations in traces of short runs, it is usually insufficient for longer runs. We therefore emulated a longer run by inserting sleep statements immediately before and after the main computational phase so that it was carried out ten minutes after initialization and ten minutes before finalization. This corresponds to a scenario, in which only distinct intervals of a longer run are traced with tracing being switched off in between. Since full traces of long running applications may consume a prohibitive amount of storage space, the "partial" tracing emulated here mimics the recommended practice of tracing only pivotal points that warrant a more detailed analysis. For our purposes, the artificial chronological distance to the offset measurements on either end of the run adjusted the interpolation interval to roughly twenty minutes execution time. However, with many realistic codes running for hours, this can still be regarded as an optimistic assumption. Compared to true partial tracing of a longer SMG2000 run, our method had the advantage that the total runtime including the actual computational activity and therefore the distance between the two offset measurements was roughly the same for all configurations.



(a) MareNostrum. (b) Cacau.

FIG. 5.1. *Percentage of messages with the order of send and receive events being reversed, of messages with clock condition violations, and of clock condition violations explicitly corrected by the* CLC *algorithm during forward amortization.*

TABLE 5.1
Average and maximum errors of message events in reversed order.

Platform	Avg. error [μs]	Max. error [μs]
MareNostrum	2.6	323
Cacau	4.3	186

Figure 5.1 shows the frequency of clock condition violations on MareNostrum and Cacau for a range of scales. Since the number of violations varies between runs, the numbers represent averages across three measurements for each configuration. The numbers show the percentage of messages with the order of send and receive events being reversed in the original trace, of messages with clock condition violations ($t_{recv} < t_{send} + l_{min}$) in the original trace, and of clock condition violations explicitly corrected by the CLC algorithm during forward amortization. We also counted logical messages that can be derived by mapping collective communication onto point-to-point semantics. When visualized, messages with the order of send and receive events being reversed seem to flow backward in time. The number of violations explicitly corrected by the CLC algorithm is usually smaller than the initial number of violations because some of them are already implicitly removed during forward amortization before a correction can be applied. On MareNostrum, around 1% of the messages flow backward in time, while on Cacau the percentage ranges between 1 and 6%. Higher latencies on MareNostrum offer a potential explanation for the smaller number of violations detected on this system because higher latencies naturally insert a larger temporal distance between send and receive events of the same message. Although the number of inconsistent messages on Cacau seem to decrease with growing numbers of processes, the results on MareNostrum do not confirm a clear correlation between the two indicators. Table 5.1 lists the average and maximum displacement errors (i. e. the time the receive event appears earlier than the send event) of message events in backward order, as seen in the original trace.

5.1. Scalability. Because it is the larger system, we evaluated our algorithm's scalability on MareNostrum. According to Figure 5.2, the parallel timestamp synchronization, the Scalasca wait-state analysis, and the execution time of SMG2000 itself exhibit roughly equivalent scaling behavior - a result of the replay-based nature of the two analysis mechanisms and the communication-bound performance characteristics of SMG2000. The fact that the total time needed by the integrated Scalasca analysis (synchronization and wait-state analysis) including loading the traces grows more steeply suggests that I/O will increasingly dominate the overall behavior beyond 1024 processes, rendering the additional cost of the synchronization negligible.

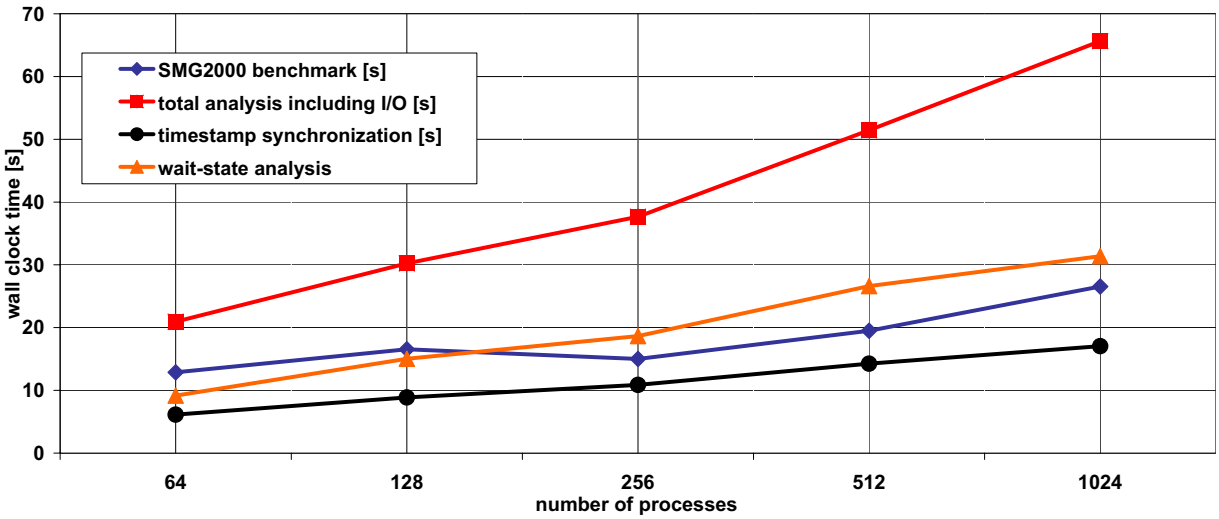


FIG. 5.2. *Scalability of parallel timestamp synchronization on MareNostrum.*

5.2. Accuracy. The transformation performed by the CLC algorithm raises the question of how accurate the modified traces actually are. To answer this question, it must first be acknowledged that traces with clock-condition violations are inaccurate because they are inconsistent. The behavior they reflect violates causation and is therefore impossible. The CLC algorithm eliminates these inconsistencies, improving the accuracy of inter-process timings. A very simple metric quantifying this improvement is the fraction of clock-condition violations found in the original trace (second row in Figure 5.1), which are removed in the modified trace.

However, the necessary corrections applied in the course of the algorithm also modify—as a collateral effect—relative process-local event timings, which may lead to differences in the lengths of local intervals when comparing the original with the modified trace. Again, since the original timestamps have been taken with clocks that exhibit unstable drifts, there is, of course, no guarantee that the measured interval lengths are correct, although the deviation can be expected to be small in comparison to the total interval length. In addition, the original timings are all we "know" about the target application's local execution behavior. For this reason, we decided to evaluate how this knowledge has been preserved in the modified trace.

To assess the fidelity of process-local timings after applying the CLC algorithm, we determined the relative deviation of local interval lengths, considering two different types of intervals: (i) intervals between an event and the first event of the same process, which roughly corresponds to the timestamp of an event relative to the begin of the trace and (ii) intervals between adjacent process-local events (i. e., intervals between an event and its immediate successor). We refer to the first interval type as the *event position* and to the second interval type as the *event distance*.

The maximum relative deviation of the event position across all our measurements was below 0.0001 %; the maximum absolute deviation was 425.18 μs, roughly corresponding to the maximum displacement error observed (Table 5.1). Table 5.2 shows the relative deviation of the event distance across different combinations of platform and number of processors. The numbers in individual columns are percentages and are rounded to two digits after the decimal point. They represent the maximum across three measurements. To account for the relatively long "correct" stretches artificially introduced by the sleep statements before and after the main computation, only the middle section of the trace between the sleep statements was considered. Furthermore, since deviations in larger intervals are more relevant to performance analysis than those in smaller intervals, the average was calculated using $\sum |\Delta t| / \sum |t|$ to assign appropriate weight to larger intervals, with Δt being the absolute deviation and t being the original interval length.

TABLE 5.2
Relative deviation of event distance. All numbers are given in percent and rounded to two digits after the decimal point.

	MareNostrum					**Cacau**		
# CPUs	64	128	256	512	1024	32	64	128
Average	0.00	0.01	0.01	0.00	0.01	0.00	0.01	0.00
Maximum	27.17	461.74	411.52	311.64	974.44	82.78	69.31	16.77
Percentage of intervals with deviation above threshold								
$> 0\%$	80.80	96.21	97.34	98.23	99.07	92.43	93.67	24.27
$> 0.01\%$	0.15	0.63	0.39	0.48	0.81	0.47	0.27	0.04
$> 0.1\%$	0.15	0.61	0.38	0.46	0.79	0.45	0.26	0.04
$> 1\%$	0.01	0.10	0.06	0.07	0.18	0.09	0.04	0.00
$> 10\%$	0.00	0.01	0.00	0.00	0.01	0.01	0.00	0.00
$> 100\%$	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Percentage of execution time consumed by intervals with deviation above threshold								
$> 0\%$	80.51	96.22	96.86	92.20	95.17	50.37	92.50	23.00
$> 0.01\%$	0.41	0.95	0.58	0.71	0.72	0.24	0.68	0.05
$> 0.1\%$	0.20	0.23	0.28	0.21	0.24	0.19	0.55	0.04
$> 1\%$	0.01	0.02	0.03	0.02	0.02	0.04	0.11	0.00
$> 10\%$	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
$> 100\%$	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

It can be seen that in spite of very small averages, deviations of occasionally more than 100% are still possible. Although the backward amortization is designed to smooth sudden jumps introduced by the forward amortization, it can happen that a send event cannot be advanced far enough without causing a new clock condition violation when passing the corresponding receive event. To evaluate frequency and extent of such situations, we calculated (i) the percentage of intervals whose deviation exceeds a certain threshold and (ii) the percentage of execution time (accumulated across all processes) consumed by intervals whose deviation exceeds the threshold. The results given in Table 5.2 indicate that larger deviations are rare and that their influence on performance-analysis results will usually be negligible.

6. Conclusion. Event traces of parallel applications may suffer from inaccurate timestamps in the absence of synchronized hardware clocks. As a consequence, the analysis of such traces may yield wrong quantitative and even qualitative results or confuse the user of time-line visualizations with messages flowing backward in time. Because linear offset interpolation based on offset measurements can account for such deficiencies only for very short runs, we have designed and implemented a parallel algorithm for the retroactive correction of timestamps based on logical clocks that eliminates inconsistent inter-process timings while only marginally changing the length of local intervals. Our replay-based implementation scales easily to more than thousand application processes and shows potential for even larger configurations. The algorithm has been incorporated into the Scalasca framework to facilitate trace analyses of longer runs on larger cluster systems. In the future, we want to extend our algorithm to hybrid applications employing a mix of message passing and shared-memory parallelism, which will require paying attention to happen-before relationships, for example, imposed by OpenMP barrier or lock event semantics.

Acknowledgment. This work has been funded by the Helmholtz Association under Grant No. VH-NG-118. The authors also thankfully acknowledge the computer resources, technical expertise and assistance provided by the Barcelona Supercomputing Center. In particular, we would like to express our gratitude to Judit Gimenez, Jesus Labarta, and David Vicente for their generous support.

REFERENCES

[1] *Scalasca*. `http://www.scalasca.org/`.
[2] O. Babaoğlu and R. Drummond, *(Almost) no cost clock synchronization*, in Proceedings of 7th International Symposium on Fault-Tolerant Computing, IEEE Computer Society Press, July 1987, pp. 42–47.
[3] D. Becker, R. Rabenseifner, and F. Wolf, *Timestamp synchronization for event traces of large-scale message-passing applications*, in Proc. of the 14th European PVM/MPI Users' Group Meeting, Paris, France, vol. 4757 of Lecture Notes in Computer Science, Springer, September-October 2007, pp. 315–325.
[4] M. Biberstein, Y. Harel, and A. Heilper, *Clock synchronization in Cell BE traces*, in Proc. of the 14th Euro-Par Conference, Las Palmas de Gran Canaria, Spain, vol. 5168 of Lecture Notes in Computer Science, Springer, August - September 2008, pp. 3–12.
[5] H. Brunst and W. E. Nagel, *Scalable performance analysis of parallel systems: Concepts and experiences*, in Proceedings of the Parallel Computing Conference (ParCo), Dresden, Germany, 2003.
[6] A. Chan, W. Gropp, and E. Lusk, *Scalable log files for parallel program trace data (draft)*, tech. report, Argonne National Laboratory, 2000.
[7] F. Cristian, *Probabilistic clock synchronization*, Distributed Computing, 3 (1989), pp. 146–158.
[8] J. E. Cuny, A. A. Hough, and J. Kundu, *Logical time in visualizations produced by parallel programs*, in Proceedings of Visualization '92, Boston, MA, USA, IEEE Computer Society Press, October 1992, pp. 186–193.
[9] G. v. Dijk and A. v. d. Wal, *Partial ordering of sychronization events for distributed debugging in tightly-coupled multiprocessor systems*, in 2nd European Conference on Distributed Memory Computing (EDMCC2), Munich, Germany, vol. 487 of Lecture Notes in Computer Science, Springer, April 1991, pp. 100–109.
[10] J. Doleschal, A. Knüpfer, M. S. Müller, and W. E. Nagel, *Internal timer synchronization for parallel event tracing*, in Proceedings 15th European PVM/MPI Users' Group Meeting, Dublin, Ireland, Lecture Notes in Computer Science, Dublin, Ireland, September 2008, Springer.
[11] R. Drummond and O. Babaoğlu, *Low-cost clock synchronization*, Distributed Computing, 6 (1993), pp. 193–203.
[12] A. Duda, G. Harrus, Y. Haddad, and G. Bernard, *Estimating global time in distributed systems*, in Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, Germany, IEEE Computer Society Press, September 1987, pp. 299–306.
[13] T. H. Dunigan, *Hypercube clock synchronization*, Technical Report ORNL TM-11744, Oak Ridge National Laboratory, TN, February 1991.
[14] D. Edwards and P. Kearns, *DTVS: A distribute trace visualization system*, in Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing, Dallas, TX, IEEE Computer Society Press, October 1994, pp. 281–288.
[15] C. J. Fidge, *Timestamps in message-passing systems that preserve partial ordering*, in Proceedings of 11th Australian Computer Science Conference, February 1988, pp. 56–66.
[16] ———, *Partial orders for parallel debugging*, ACM SIGPLAN Notices, 24 (1989), pp. 183–194.

[17] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, and B. J. N. Wylie, *A parallel trace-data interface for scalable performance analysis*, in Proceedings of the Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Umeå, Sweden, vol. 4699 of Lecture Notes in Computer Science, Springer, June 2006, pp. 398–408.

[18] M. Geimer, F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, and B. Mohr, *The Scalasca performance toolset architecture*, in Proceedings of the Int'l Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece, June 2008.

[19] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, *Scalable parallel trace-based performance analysis*, in Proc. 13th European PVM/MPI Users' Group Meeting, Bonn, Germany, vol. 4192 of Lecture Notes in Computer Science, Springer, September 2006, pp. 303–312.

[20] D. Haban and W. Weigel, *Global events and global breakpoints in distributed systems*, in Proceedings of the 21st Hawaii International Conference on System Sciences, 1988, pp. 166–175, vol. II.

[21] R. Hofmann, *Gemeinsame Zeitskala für lokale Ereignisspuren*, in Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, 7. GI/ITG-Fachtagung, Aachen, Germany, Springer, September 1993.

[22] ———, *Gesicherte Zeitbezüge für die Leistungsanalyse in parallelen und verteilten Systemen*, PhD thesis, Universität Erlangen-Nürnberg, Technische Fakultät, 1993.

[23] R. Hofmann and U. Hilgers, *Theory and tool for estimating global time in parallel and distributed systems*, in Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing, Madrid, Spain, January 1998, pp. 173–179.

[24] H. Jafri, *Measuring causal propagation of overhead of inefficiencies in parallel applications*, in Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, MA, November 2007, pp. 237–243.

[25] J.-M. Jézéquel, *Building a global time on parallel machines*, in Proceedings of the 3rd International Workshop on Distributed Algorithms, J.-C. Bermond and M. Raynal, eds., vol. 392 of Lecture Notes in Computer Science, Springer, 1989, pp. 136–147.

[26] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, Communications of the ACM, 21 (1978), pp. 558–565.

[27] E. Maillet and C. Tron, *On efficiently implementing global time for performance evaluation on multiprocessor systems*, Journal of Parallel and Distributed Computing, 28 (1995), pp. 84–93.

[28] F. Mattern, *Virtual time and global states of distributed systems*, in Proceedings of the International Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France, M. Cosnard and P. Quinton, eds., Elsevier, October 1989, pp. 215–226.

[29] D. L. Mills, *Network Time Protocol (Version 3)*. The Internet Engineering Task Force - Network Working Group, March 1992. RFC 1305.

[30] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, *VAMPIR: Visualization and analysis of MPI resources*, Supercomputer 63, XII (1996), pp. 69–80.

[31] R. L. Probert, H. Yu, and K. Saleh, *Relative-clock-based specification and test result analysis of distributed systems*, in Eleventh Annual International Phoenix Conference on Computers and Communications, Scottsdale, AZ, USA, IEEE, New York, April 1992, pp. 687–694.

[32] R. Rabenseifner, *The controlled logical clock - a global time for trace based software monitoring of parallel applications in workstation clusters*, in Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed (PDP'97), London, UK, January 1997, pp. 477–484.

[33] ———, *Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen (The controlled logical clock, a global clock for trace-based monitoring of parallel applications)*, PhD thesis, University of Stuttgart, Faculty of Computer Science, March 2000. `http://elib.uni-stuttgart.de/opus/volltexte/2000/600/`.

[34] M. Raynal, *A distributed algorithm to prevent mutual drift between n logical clocks*, Information Processing Letters, 24 (1987), pp. 199–202.

[35] M. v. Riek, B. Tourancheau, and X.-F. Vigouroux, *Monitoring of distributed memory multicomputer programs (A general approach to the monitoring of distributed memory MIMD multicomputers)*, Technical Report CS-93-204, University of Tennessee, 1993. `http://www.netlib.org/tennessee/ut-cs-93-204.ps`.

[36] R. Schwarz and F. Mattern, *Detecting causal relationships in distributed computations: in search of the holy grail*, Distributed Computing, 7 (1994), pp. 149–174.

[37] Z. Yang and T. A. Marsland, *Annotated bibliography on global states and times in distributed systems*, Operating Systems Review, 27 (1993), pp. 55–74.

Edited by: Fatos Xhafa, Leonard Barolli
Received: September 30, 2008
Accepted: December 15, 2008