Parallel Computing 35 (2009) 595–607

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Scalable timestamp synchronization for event traces of message-passing applications

Daniel Becker [a,b,*], Rolf Rabenseifner [c], Felix Wolf [a,b], John C. Linford [d]

[a] Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany
[b] Department of Computer Science, RWTH Aachen University, 52056 Aachen, Germany
[c] High Performance Computing Center, University of Stuttgart, 70550 Stuttgart, Germany
[d] Department of Computer Science, Virginia Tech, Blacksburg, VA 24061, USA

ARTICLE INFO

Article history:
Received 7 May 2008
Accepted 27 December 2008
Available online 21 January 2009

Keywords:
Performance analysis
Event tracing
Clock synchronization

ABSTRACT

Event traces are helpful in understanding the performance behavior of message-passing applications since they allow the in-depth analysis of communication and synchronization patterns. However, the absence of synchronized clocks may render the analysis ineffective because inaccurate relative event timings may 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, time-dependent drifts and other inaccuracies may still disarrange the original succession of events – especially during longer runs. The controlled logical clock algorithm accounts for such violations in point-to-point communication by shifting message events in time as much as needed while trying to preserve the length of local intervals. In this article, we describe how the controlled logical clock is extended to collective communication to enable the correction of realistic message-passing traces. We present a parallel version of the algorithm scaling to more than thousand processes and evaluate its accuracy by showing that it eliminates inconsistent inter-process timings while preserving the length of local intervals.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Event tracing is a popular technique for the postmortem performance analysis of message-passing applications because it can be used to examine 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*. Inaccurate timestamps may lead to false conclusions during performance analysis, for example, when the impact of certain behaviors is quantified, or – even more strikingly – may confuse the user of trace visualization tools such as Vampir [22] by causing arrows representing messages to point backward in time-line views.

To avoid clock condition violations, the error of timestamps should ideally be smaller than one half of the message latency. 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 [21] 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

* Corresponding author. Address: Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany.
E-mail addresses: d.becker@fz-juelich.de (D. Becker), rabenseifner@hlrs.de (R. Rabenseifner), f.wolf@fz-juelich.de (F. Wolf), jlinford@vt.edu (J.C. Linford).

0167-8191/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2008.12.012

runs at a different but constant speed (i.e., drift), the (global) time of an arbitrarily chosen master clock can be calculated locally as a linear function of the local time. This approach is taken in the tracing library of Scalasca [14], a performance-analysis toolset that can be used to automatically identify wait states in event traces of large-scale MPI programs. For this purpose, Scalasca performs offset measurements between all local clocks and the master clock once at program initialization and once at program finalization. However, as the assumption of constant drifts is only an approximation, violations of the clock condition may still occur – especially when the offset measurements are taken with long intervals in between.

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. The *controlled logical clock* (CLC) [23] is a method to retroactively correct timestamps violating the clock condition. As the modification of individual timestamps might change the length of local intervals and even introduce new violations, the correction takes the context of the modified event into account by carefully stretching the local time axis in the immediate vicinity of the affected event. The current CLC algorithm, however, is limited by two factors. First, it covers only point-to-point operations and ignores collective ones. Second, it is a serial algorithm designed for a single global trace file. In this article, we describe how the controlled logical clock is extended to collective communication to enable a more complete correction of realistic message-passing traces. In addition, we present a parallel version of the algorithm scaling to more than thousand processes and evaluate its accuracy by showing that it eliminates inconsistent inter-process timings while preserving the length of local intervals.

The article is structured as follows: after reviewing related work in Section 2, we describe the original serial version of the algorithm in Section 3. In Section 4, we specify the extensions necessary to handle collective operations. Then, we present the parallel algorithm design in Section 5 and outline its implementation within Scalasca. We evaluate the scalability of the parallel version in Section 6, where we also show that the collaterally introduced deviations of local interval lengths remain within acceptable limits. Finally, we conclude our paper and give an outlook on future work in Section 7.

2. Related work

In this section we cite several approaches for avoiding or correcting inconsistent timestamps, most of them usually applied postmortem. Network-based synchronization protocols aim at synchronizing distributed clocks before reading them. The clocks query the global time from reference clocks, which are often organized in a hierarchy of servers. For instance, NTP [21] 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 1 ms 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, the global time can be established by measuring offsets to a designated master clock using Cristian's probabilistic remote clock reading technique [5]. After estimating the drift, the local time can be mapped onto the global (i.e., master) time via linear interpolation. Offset values among participating clocks are measured either at program initialization [9,10] or at initialization and finalization [19], and are subsequently used as parameters of the linear correction function. 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 [6]. 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.

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 differences between clock values of receive events and clock values of 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 [8] to determine the smoothing function. Using a minimal spanning tree algorithm, Jezequel [17] adopted Duda's algorithm for arbitrary processor topologies. In addition, Hofmann [15] 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. Using a graph-theory algorithm to calculate the shortest paths, Hofmann and Hilgers [16] simplified Jezequel's algorithm for handling multi-processor topologies. 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 subintervals for handling a non-linear drift of the physical clocks). Babaoglu and Drummond [2,7] 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, non-linear relations between message latency and message length, and one-sided communication topologies limit the usefulness of error estimation approaches.

In contrast, logical synchronization uses happened-before relations among send and receive pairs to synchronize distributed clocks. Lamport introduced a discrete logical clock [18] with each clock being represented by 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. As an enhancement of Lamport's discrete logical clock, Fidge [11,12] and Mattern [20] proposed a vector clock. In their scheme, each processor maintains a vector representing all processor-local clocks. While the local clock is advanced after each local event as before, the 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.

3. The controlled logical clock

Because we have extended and parallelized the CLC algorithm to use it within Scalasca, we describe it in terms of the Scalasca event model, which is similar to the VAMPIR event model [22], for which the algorithm was originally designed. As far as message passing is concerned, the two models differ only in the way they express collective communication, which the original algorithm ignores.

The information Scalasca records for an individual event includes at least the timestamp, the location (i.e., the process) causing the event, and the event type. Depending on the type, additional information may be supplied. The event model distinguishes between programming-model independent events, such as entering and exiting code regions, and events related to MPI operations. The latter include events representing point-to-point operations, such as sending and receiving messages, and events representing the completion of collective operations. These collective exit events are specializations of normal exit events carrying amongst other attributes information on the communicator. This information allows identifying concurrent collective exits belonging to the same collective operation instance. Event sequences recorded for typical MPI operations are given in Table 1.

Non-synchronized processor clocks may cause inaccurate timestamps in event traces. However, the *clock condition*, which is given in Eq. (1), requires that the *happened-before* relation $e \rightarrow e'$ [18] between two events e and e' with their respective timestamps $C(e)$ and $C(e')$ must be satisfied

$$\forall\ e, e' : e \rightarrow e' \Rightarrow C(e) < C(e'). \tag{1}$$

The *controlled logical clock* (CLC) algorithm by Rabenseifner [23,24] is an enhancement of Lamport's logical clock [18]. The algorithm requires timestamps with limited errors, which can be achieved through weak pre-synchronization, such as linear offset interpolation between program start and end. The CLC algorithm 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. Since messages need time to travel to their destination, we can reformulate the above condition, as given in Eq. (2), with l_{min} being the minimum message latency. Note that the hierarchical structure of many parallel systems allows the definition of multiple l_{min} per system, for example, depending on whether messages are exchanged within the same node or across different nodes

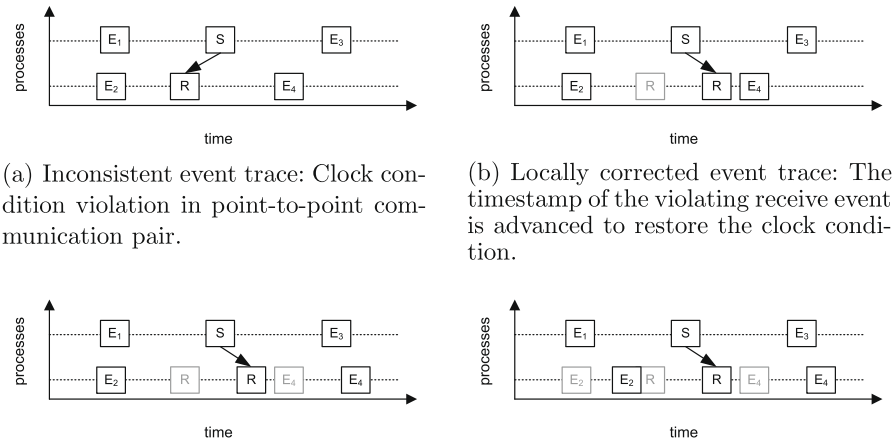
$$\forall\ e, e' : e \rightarrow e' \Rightarrow C(e) + l_{min} \leqslant C(e'). \tag{2}$$

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.

Fig. 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) or receive (R) event, each of them enclosed by two other events (E_i). Fig. 1a shows the initial event trace based on the measured timestamps with insufficiently synchronized local clocks. It 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 (Fig. 1b). 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 (Fig. 1c). However, the jump discontinuity introduced by adjusting R affects not only

Table 1
Event sequences recorded for typical MPI operations.

Function name	Event sequence
`MPI_Send()`	(enter, send, exit)
`MPI_Recv()`	(enter, receive, exit)
`MPI_Allreduce()`	(enter, collective exit) for each participating process



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

(b) Locally corrected event trace: The 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 the two events.

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

Fig. 1. Backward and forward amortization in the controlled logical clock algorithm.

events later than R but also events earlier than R. This is corrected during the backward amortization, which shifts E_2 closer to the new position of R (Fig. 1d).

While the forward amortization is at least initially applied to all events following R, the backward amortization is applied only to a limited amortization interval before R using a linearly increasing correction. However, not to violate the clock condition anew, the correction must not advance any send event located in this interval farther than its 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 different slopes for subintervals before, after, or between those sends [24] (Fig. 2).

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 scales the timestamps of events beyond the corrected one using control variables to ensure that the overall error remains within predefined boundaries. Below, we give a formal explanation of forward and backward amortization.

3.1. Forward amortization

In the following, we use the symbol LC' to denote timestamps computed by the CLC algorithm. LC' is modeled with t as the wall clock time and $T(t)$ as the global time to which the process clocks $C_i(t)$ $(i = 0, \ldots, n-1)$ are synchronized. Next, n is the number of processes, e_i^j is the jth event on process i, and so is $E = \left\{ e_i^j \mid i = 0, \ldots, n-1,\ j = 0, \ldots, j_{\max}(i) \right\}$ the set of all events in the trace. In addition, the set of matching send and receive pairs is defined as

$$M = \left\{ (e_k^l, e_m^n) | e_k^l = \text{send event},\ e_m^n = \text{matching receive event} \right\}. \tag{3}$$

Note that the send event always marks the beginning of a send operation, whereas a receive event marks the end of a receive operation. Moreover, e_i^j is an internal event if it is neither a send nor a receive event. Furthermore, δ_i is the minimal difference

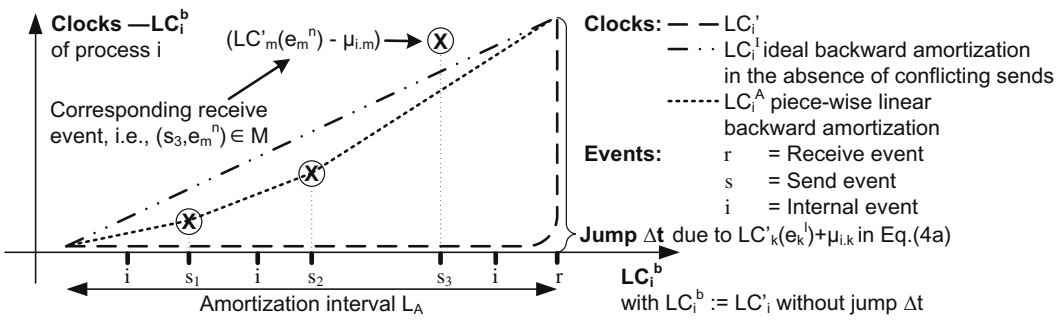


Fig. 2. Backward amortization algorithm.

between two events on process i and $\mu_{k,i}$ is the minimum message delay of messages from process k to process i. Finally, γ_i^j is a control variable with $\gamma_i^j \in [0,1]$. For each process i, LC'_i is now defined as

$$LC'_i\left(e_i^j\right) := \begin{cases} \max\Big(LC'_k(e_k^l) + \mu_{k,i}, \\ \quad LC'_i\left(e_i^{j-1}\right) + \delta_i, \\ \quad LC'_i\left(e_i^{j-1}\right) + \gamma_i^j\left(C_i\left(t\left(e_i^j\right)\right) - C_i\left(t\left(e_i^{j-1}\right)\right)\right), \\ \quad C_i\left(t\left(e_i^j\right)\right)\Big) \quad \text{if } \exists e_k^l : \left(e_k^l, e_i^j\right) \in M & \text{(a)} \\ \max\Big(LC'_i\left(e_i^{j-1}\right) + \delta_i, \\ \quad LC'_i\left(e_i^{j-1}\right) + \gamma_i^j\left(C_i\left(t\left(e_i^j\right)\right) - C_i\left(t\left(e_i^{j-1}\right)\right)\right), \\ \quad C_i\left(t\left(e_i^j\right)\right)\Big) \quad \text{otherwise.} & \text{(b)} \end{cases} \tag{4}$$

As can be seen, the algorithm consists of two equations. Eq. (4a) adjusts the timestamps of receive events while Eq. (4b) modifies timestamps of internal and send events. Note that for each process, the terms $LC'_i\left(e_i^{j-1}\right) + \delta_i$ and $LC'_i\left(e_i^{j-1}\right) + \gamma_i^j\left(C_i\left(t\left(e_i^j\right)\right) - C_i\left(t\left(e_i^{j-1}\right)\right)\right)$ must be omitted for the first event ($j = 0$).

Through the term $C_i\left(t\left(e_i^j\right)\right)$ in Eqs. (4a) and (4b), the algorithm ensures that a correction is only applied if the trace violates the clock condition. The new timestamps satisfy the clock condition because the term $LC'_k(e_k^l) + \mu_{k,i}$ in Eq. (4a) guarantees that $LC'\left(e_i^j\right)$ is put forward compared to $C_i\left(t\left(e_i^j\right)\right)$ if required in the case of a clock condition violation. To make sure that the clock does not stop after a clock condition violation, the term $LC'_i\left(e_i^{j-1}\right) + \gamma_i^j\left(C_i\left(t\left(e_i^j\right)\right) - C_i\left(t\left(e_i^{j-1}\right)\right)\right)$ in Eqs. (4a) and (4b) approximates the duration of the original communication after a clock condition violation.

Moreover, Rabenseifner has shown that γ_i^j with a constant value can cause LC' to be faster than the fastest clock among all process-local clocks C_i [24]. Cyclic changes of physical clock drifts may cause an avalanche effect that enlarges the value of clock corrections and propagates until the end. To avoid this effect, a control loop is used to find the optimal value of γ_i^j. The controller tries to limit the differences between LC' and T, that is, the controller estimates the output error indirectly because $T\left(t\left(e_i^j\right)\right)$ is unknown. If $1-\gamma$ is chosen smaller than the maximal drift differences, the controller will enlarge $1-\gamma$ (e.g., to 1%) to guarantee that any propagation is bounded by this factor. To calculate γ_i^j for each event, the controller requires a global view of the event data. Typically, γ_i^j is kept less than 1 minus the maximum drift of the clocks, however, in most cases a fixed $\gamma = 0.99$ or 0.999 is good enough because physical clock drifts are normally less than 10^{-4}. For subsequent events of the same process, the term $LC'_i\left(e_i^{j-1}\right) + \delta_i$ in Eqs. (4a) and (4b) causes LC' to advance at least a small number of ticks δ_i if the controller has reduced γ_i^j to nearly zero. Rabenseifner describes the control mechanism in more detail in [24].

A jump discontinuity in LC' of Δt is caused by the term $LC'_k(e_k^l) + \mu_{k,i}$ in Eq. (4a) if $LC'\left(e_i^j\right)$ of the violating receive event is put forward compared to $C_i\left(t\left(e_i^j\right)\right)$. The term $LC'_i\left(e_i^{j-1}\right) + \gamma_i^j\left(C_i\left(t\left(e_i^j\right)\right) - C_i\left(t\left(e_i^{j-1}\right)\right)\right)$ in Eq. (4a) implements the forward amortization of such a jump. That is, the clock LC'_i for subsequent events of process i runs with the speed of C_i reduced by the factor γ_i^j.

3.2. Backward amortization

Backward amortization is applied to smooth jump discontinuities caused by forward amortization. This is achieved by slowly building up the ascension to a jump Δt using a piecewise process-local linear correction in an amortization interval L_A of appropriate size before the violating receive event (Fig. 2) [24]. The compensation is realized by setting the timestamps forward. If there is no violating send event in the backward amortization interval of a process i, then the dash-dotted linear interpolation can be used. In Fig. 2, the horizontal axis represents LC_i^b, which is equal to LC'_i (i.e., the state after forward amortization) but without the jump Δt at the corrected receive event r (shown on the right). The vertical axis shows offsets to LC_i^b after applying different stages of backward amortization. Naturally, the offset at r corresponds to the jump Δt. Note that the smaller the gradient of a clock in this figure, the better the correction and the smaller the perturbation of preceding events. Therefore, the ratio $\Delta t/L_A$ should be only a few percent. Clearly, adjacent clock condition violations cause a larger perturbation.

In order to avoid new violations of the clock condition, the correction must not advance the timestamps of send events farther than $LC'_m - \mu_{i,m}$ of the corresponding receive event e_m^n of a remote process m. These upper limits are shown as circled values above the locations of the send events. If these limits are smaller than the dashed-dotted line (here at events s_1 and s_2), then a reduced piecewise linear interpolation function must be used (see the dotted line in Fig. 2).As can be seen, the clock error rate is higher than the desired $\Delta t/L_A$ in the interval (s_2, r). For each receive event with a jump, the backward amor-

tization algorithm is applied independently. If there are additional receive events inside the amortization interval during such a calculation step, then these events can be treated like internal events, because advancing the timestamp of a receive event further cannot violate the clock condition.

4. Collective operations

Unfortunately, the original CLC algorithm has been designed to correct clock condition violations only related to point-to-point communication. Collective communication semantics are ignored. In this section, we explain how the algorithm can be made suitable for realistic MPI applications that perform not only point-to-point but also collective communication. We start with a discussion of forward amortization, followed by a discussion of backward amortization.

4.1. Forward amortization for collectives

The CLC algorithm synchronizes the timestamps of concurrent events based on happened-before relations. A receive event is put forward in time whenever the matching send event appears too late in the trace to satisfy the clock condition. In our event model, a collective operation instance consists of multiple pairs of enter and collective exit events (i.e., one pair for each participating process). The basic idea behind our extension is to map these events onto point-to-point communication events. For this purpose, we consider a single collective operation as being composed 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 enter events of all sending processes and the exit event of the receiving 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 types assigned during this mapping the *logical event types* as opposed to the actual event types (e.g., enter or collective exit) specified in the event trace.

Below, we review the different types of collective operations to identify happened-before relationships based on the decomposition of collective operations into send and receive pairs. With S and R we denote the set of logical send and receive events in a collective operation instance, respectively. For each call to a collective operation, the set of all send–receive pairs M is enlarged by adding $S \times R$ with two exceptions, which are discussed later.

1-to-N: One root process sends its data to *N* other processes. Examples are `MPI_Bcast()`, `MPI_Scatter()`, and `MPI_Scatterv()`. S only contains the send event of the root process (i.e., its enter event), whereas R contains receive events from all processes in the communicator (i.e., all collective exit events) with a data length greater zero, that is, the set may be smaller than the size of the communicator in the case of variable length operations (`MPI_....v()`).

N-to-1: One root process receives its data from *N* processes. Examples are `MPI_Reduce()`, `MPI_Gather()`, and `MPI_Gatherv()`. R only contains the receive event on the root process (i.e., its collective exit event). S is the set of send events (i.e., all enter events) from all processes in the communicator with a data length greater zero. Given that the root process is not allowed to exit the operation until the last process has entered it, the latest enter event is the relevant send event to fulfill the collective clock condition. Hence, if S contains more than one element, the term $LC'_k(e^l_k) + \mu_{k,i}$ in Eq. (4a) must be replaced by the maximum of $LC'_k(e^l_k) + \mu_{k,i}$ over all $e^l_k \in S$. That is, Eq. (4) must be replaced by Eq. (4′).

$$LC'_i\left(e^j_i\right) := \begin{cases} \max\left(\max\limits_{\{e^l_k \mid (e^l_k, e^j_i) \in M\}} \left(LC'_k(e^l_k) + \mu_{k,i}\right), \right. \\ \qquad LC'_i\left(e^{j-1}_i\right) + \delta_i, \\ \qquad \left. LC'_i\left(e^{j-1}_i\right) + \gamma^j_i\left(C_i\left(t\left(e^j_i\right)\right) - C_i\left(t\left(e^{j-1}_i\right)\right)\right)\right), \\ \qquad C_i\left(t\left(e^j_i\right)\right) \quad \text{if } \exists e^l_k : \left(e^l_k, e^j_i\right) \in M & \text{(a)} \\ \ldots \quad [\text{same as}(4\text{b})]. & \text{(b)} \end{cases} \tag{4′}$$

N − to − N′: All processes of the communicator are at the same time sender and receiver. Examples are `MPI_Allreduce()`, `MPI_Allgather()`, `MPI_Alltoall()`, and `MPI_Barrier()` with $N' = N$, and the variable length operations `MPI_Reduce_scatter()`, `MPI_Allgatherv()`, and `MPI_Alltoallv()`. S and R are defined by all those enter and collective exit events whose processes contribute input data or receive output data, respectively. For a call to `MPI_Barrier()`, all processes in the communicator contribute to S and R.

Special cases: For `MPI_Scan()` and `MPI_Exscan()`, the set of messages added to M cannot be expressed as the Cartesian product $S \times R$. Instead, the set of messages added to M has the form

$$\left\{\left(e^l_k, e^j_i\right) \mid k = 0, \ldots, N-1,\ i = 0, \ldots, k-x\right\}$$

with e^l_k referring to the enter event and e^j_i to the collective exit event of a collective operation instance, and with $x = 0$ for `MPI_Scan()` and $x = 1$ for `MPI_Exscan()`.

Regardless of the collective operation type, it is important to optimize the handling of $S \times R$ in Eq. (4′a). Our parallelized version of the CLC algorithm achieves this by taking advantage of the way collectives are usually implemented, typically reducing the effort to $\mathcal{O}(\log N)$.

4.2. Backward amortization for collectives

To extend the backward amortization algorithm for collective routines, the upper bounds for the send events (see Fig. 2) must be adapted to collective events: if e_i^{j-m} is the send event of a collective routine, an upper bound for the piecewise linear interpolation at e_i^{j-m} is defined by $\min_{e_k^l \in R} LC'_k(e_k^l) - \mu_{i,k}$ with R being the set of receive events defined in Section 4.1.

5. Parallel timestamp synchronization

Scalasca, a performance analysis toolset 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 major challenges to achieving good performance, especially when trying to scale communication-intensive applications to large processor counts.

Similar to the wait-state analysis [14] 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.

5.1. Integration with Scalasca

Almost all the postmortem trace-analysis functionality of Scalasca including the parallel CLC algorithm is implemented on top of PEARL [13], 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 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.

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 correction, 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 operation is performed locally and does not require any further communication. The resulting timestamps are taken as the C_i. Inaccurate C_i can occur for two reasons: (i) inaccurate offset measurements and (ii) time-dependent clock drifts. Fig. 3 shows the non-linear behavior of hardware clocks on an Infiniband cluster after linear begin-to-end offset interpolation. As can be seen, clock deviations are still significantly larger than point-to-point or collective latencies, which implies that clock condition violations can still occur and must be accounted for.

As an alternative to the native Scalasca wait-state analysis, the traces can also be rewritten with modified timestamps, converted, and visualized using standard trace browsers such as Vampir. The full analysis process is illustrated in Fig. 4.

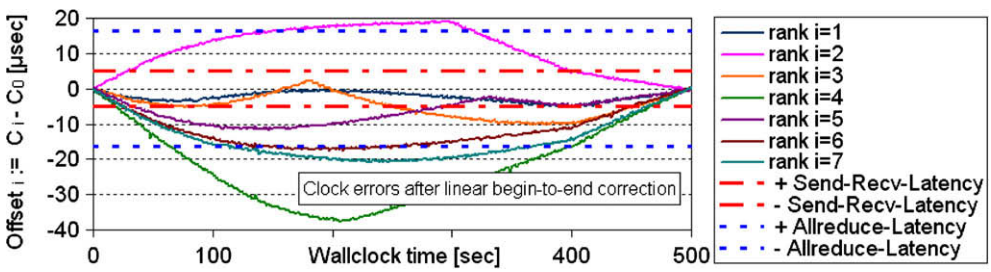


Fig. 3. Non-constant drifts of physical clocks measured on an Infiniband cluster in comparison to the send–recv and allreduce latencies.

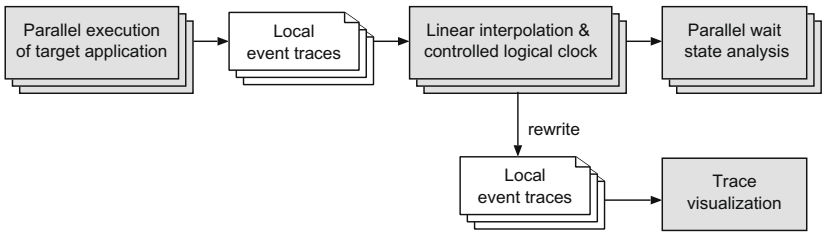


Fig. 4. Parallel trace-analysis process.

5.2. Parallel forward amortization

During the forward phase, the communication replay proceeds in the same direction as the original communication did while the target application was running. For every pair of send and receive events, the sending process sends the timestamp of the send event to the receiving process, which compares it to the timestamp of the matching receive event and, if necessary, applies the forward amortization expressed in Eqs. (4a) and (4′a). 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 for the purpose of the algorithm. 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 send/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. The corresponding operations used during the replay are listed in Table 2 (top) along with the events which will have their timestamps exchanged.

For the sake of simplicity, our current implementation uses only two different values for $\mu_{k,i}$: 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. Now, the parallel calculation of the maximum over all corresponding send events via $\max_{\{e_k^l \mid (e_k^l, e_i^j) \in M\}} \left(LC'_k(e_k^l) + \mu_{k,i} \right)$ in the case of *N*-to-1, $N - \text{to} - N'$, `MPI_Scan()`, and `MPI_Exscan()` according to Eq. (4′a) only 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. The control variable $\gamma_i^j \in [0, 1]$ for e_i^j (the *j*th) event on process *i*) is a scaling factor that is applied to interval expressions when calculating the new timestamp for e_i^j. This

Table 2
Timestamps exchanged during replay for different communication types.

Type of operation	Timestamp exchanged	MPI function
Forward		
P2P	send	`MPI_Send()`
1-to-*N*	root enter	`MPI_Bcast()`
N-to-1	max (all enters)	`MPI_Reduce()`
$N - \text{to} - N'$	max (all enters)	`MPI_Allreduce()`
`MPI_Scan()`	max (some enters)	`MPI_Scan()`
`MPI_Exscan()`	max (some enters)	`MPI_Exscan()`
Backward		
P2P	receive	`MPI_Send()`
1-to-*N*	min (all exits)	`MPI_Reduce()`
N-to-1	root exit	`MPI_Bcast()`
N-to-*N*	min (all exits)	`MPI_Allreduce()`
`MPI_Scan()`	min (some exits)	`MPI_Scan()`
`MPI_Exscan()`	min (some exits)	`MPI_Exscan()`

fulfills the purpose of preserving the length of local intervals and avoiding an avalanche-like propagation of corrections [3]. To determine the exact value for γ_i^j, 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 suitable value for γ by performing multiple passes of forward replay through the trace data until the maximum error across all processes is below a predefined threshold. During the first pass through the trace, we propose to use $\gamma = const < 1 - \epsilon$; for subsequent passes a $\gamma_{j+1} < \gamma_j$ should be used. In practice, however, more than one pass is seldom needed.

5.3. Parallel 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, in order 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. A backward replay is needed to determine these upper limits. While replaying the communication backward, with each logical send event we store 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 the backward amortization must be performed as a backward replay starting at the end of the trace with communication proceeding in backward direction to avoid the danger of deadlocks. During the backward amortization the roles of sender and receiver are reversed: the timestamp of a logical receive event must be made available to the process of the matching send event.

Table 2 (bottom) shows the operations used during backward replay along with the events which will have their timestamps exchanged. For `MPI_Scan()` and `MPI_Exscan()`, a communicator with reverse rank ordering must be used. The exchanged timestamps reflect the state after forward amortization.

Given that most MPI implementations use binomial tree algorithms to perform their collective operations, our replay-based approach reduces the communication complexity 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.

6. Experimental evaluation

Here we evaluate the scalability and accuracy of the parallel controlled logical clock algorithm for point-to-point and collective communication 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:

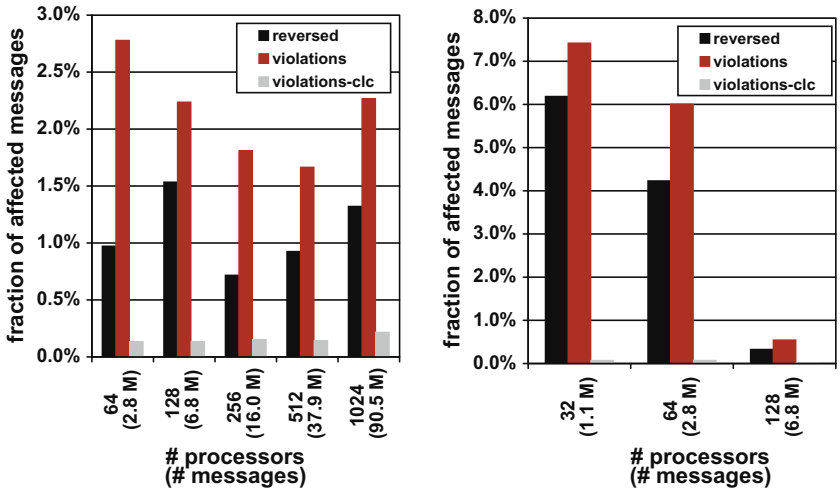
MareNostrum consists of 2560 JS21 blade computing nodes, each with two dual-core IBM 64-bit PowerPC 970MP processors running at 2.3 GHz. The computing nodes of MareNostrum communicate primarily through a Myrinet network with Myrinet adapters integrated on each server blade. The measured MPI inter-node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

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

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 10 min after initialization and 10 min 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 20 min 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.

Fig. 5 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.



(a) MareNostrum. (b) Cacau.

Fig. 5. 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.

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%. Lower latencies on Cacau offer a potential explanation for the higher number of violations detected on this system because lower latencies naturally insert a smaller temporal distance between send and receive events of the same message and so even slight clock deviations may lead to infringements of the logical event ordering. 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 3 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.

6.1. Scalability

Because it is the larger system, we evaluated our algorithm's scalability on MareNostrum. According to Fig. 6, 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.

6.2. 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

Table 3
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

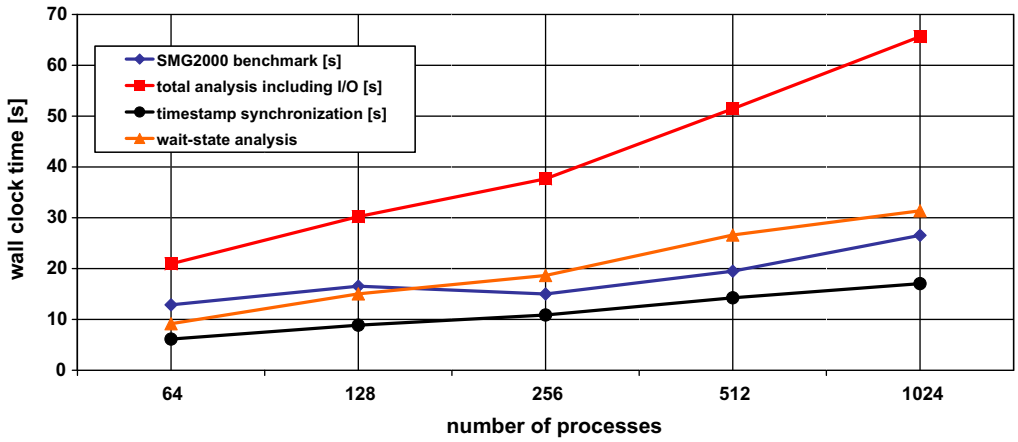


Fig. 6. Scalability of parallel timestamp synchronization on MareNostrum.

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 column in Fig. 5), 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 3). Table 4 shows the relative deviation of the event distance across different combinations of platform and number of processes. 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 state-

Table 4
Relative deviation of event distance. All numbers are given in percent and rounded to two digits after the decimal point.

# CPUs	MareNostrum					Cacau		
	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

ments 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.

In the top section of Table 4, 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 matching 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 4 indicate that larger deviations are rare and that their influence on performance-analysis results will usually be negligible.

7. Conclusion

Event traces of parallel applications may suffer from inaccurate timestamps in the absence of synchronized clocks. As a consequence, the analysis of such traces may yield wrong quantitative results and confuse the users of time-line visualizations with messages flowing backward in time. Because linear offset interpolation can account for such deficiencies only for very short runs, a retroactive correction of timestamps is required. For this reason, we have extended the controlled logical clock, an algorithm that eliminates inconsistent inter-process timings in point-to-point messages, to take collective communication semantics into account so that a more complete correction of realistic message-passing traces can be achieved. In addition, we have parallelized the extended version of the algorithm to make it more suitable for large-scale parallel applications. Finally, the algorithm has been incorporated into the Scalasca trace-analysis framework to facilitate trace analyses of longer runs on larger cluster configurations. The scalability and accuracy of our implementation has been validated using a real-world application example.

Although the new version of the algorithm only needs information about the respective event semantics (e.g., root sends to all other processes), we would like to point out that the accuracy of our model could be improved if the MPI-internal messaging inside collective operations was exposed using interfaces such as PERUSE [1]. In this case, the decomposition into (additional) send and receive events is given naturally.

In our future work, 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 Open MP barrier event semantics.

Acknowledgements

The authors 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 and Jesus Labarta for their generous support.

References

[1] MPI PERUSE - An MPI Extension for Revealing Unexposed Implementation Information, <www.mpi-peruse.org>.
[2] O. Babaoglu, R. Drummond, (Almost) no cost clock synchronization, in: Proceedings of the Seventh International Symposium on Fault-Tolerant Computing, IEEE Computer Society Press, 1987.
[3] D. Becker, R. Rabenseifner, F. Wolf, Timestamp synchronization for event traces of large-scale message-passing applications, in: Proceedings of the 14th European PVM/MPI Conference, Springer, France, 2007.
[4] M. Biberstein, Y. Harel, A. Heilper, Clock synchronization in Cell BE traces, in: Proceedings of the 14th Euro-Par Conference, Las Palmas de Gran Canaria, Spain, vol. 5168 of LNCS, Springer, 2008.
[5] F. Cristian, Probabilistic clock synchronization, Distributed Computing 3 (3) (1989) 146–158.
[6] J. Doleschal, A. Knüpfer, M.S. Müller, W.E. Nagel, Internal timer synchronization for parallel event tracing, in: Proceedings of the 15th European PVM/MPI Users' Group Meeting, Dublin, Ireland, Lecture Notes in Computer Science, Springer, Dublin, Ireland, 2008.
[7] R. Drummond, O. Babaoglu, Low-cost clock synchronization, Distributed Computing 6 (4) (1993) 193–203.
[8] A. Duda, G. Harrus, Y. Haddad, G. Bernard, Estimating global time in distributed systems, in: Proceedings of the Seventh International Conference on Distributed Computing Systems, Berlin, September, IEEE, 1987.
[9] T.H. Dunigan, Hypercube clock synchronization, Technical Report ORNL TM-11744, Oak Ridge National Laboratory, TN, February 1991.
[10] T.H. Dunigan, Hypercube clock synchronization, ORNL TM-11744 (updated), <http://www.epm.ornl.gov/~dunigan/clock.ps>, September 1994.
[11] C.J. Fidge, Timestamps in message-passing systems that preserve partial ordering, in: Proceedings of the 11th Australian Computer Science Conference, 1988.
[12] C.J. Fidge, Partial orders for parallel debugging, ACM SIGPLAN Notices 24 (1) (1989) 183–194.
[13] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, B.J. 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), vol. 4699 of LNCS, Springer, Umeå, Sweden, 2006.
[14] M. Geimer, F. Wolf, B.J.N. Wylie, B. Mohr, Scalable parallel trace-based performance analysis, in: Proceedings of the 13th European PVM/MPI Conference, Springer, Bonn, Germany, 2006.
[15] R. Hofmann, Gemeinsame Zeitskala für lokale Ereignisspuren, in: B. Walke, O. Spaniol (Eds.), Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, 7. GI/ITG-Fachtagung, Aachen, 21–23 September 1993, Springer-Verlag, Berlin, 1993.
[16] R. Hofmann, U. Hilgers, Theory and tool for estimating global time in parallel and distributed systems, in: Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing, 1998.
[17] J.-M. Jézéquel, Building a global time on parallel machines, in: J.-C. Bermond, M. Raynal (Eds.), Proceedings of the Third International Workshop on Distributed Algorithms, LNCS 392, Springer-Verlag, 1989.

[18] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565.
[19] E. Maillet, C. Tron, On efficiently implementing global time for performance evaluation on multiprocessor systems, Journal of Parallel and Distributed Computing 28 (1995) 84–93.
[20] F. Mattern, Virtual time and global states of distributed systems, in: M. Cosnard, P. Quinton (Eds.), Proceedings of International Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France, October 1988, Elsevier Science Publishers B.V., Amsterdam, 1989.
[21] D.L. Mills, Network Time Protocol (Version 3), The Internet Engineering Task Force – Network Working Group, RFC 1305, March 1992.
[22] W.E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, K. Solchenbach, VAMPIR: visualization and analysis of MPI resources, Supercomputer 12 (1) (1996) 69–80.
[23] R. Rabenseifner, The controlled logical clock – a global time for trace based software monitoring of parallel applications in workstation clusters, in: Proceedings of the Fifth Euromicro Workshop on Parallel and Distributed (PDP'97), London, UK, 1997.
[24] R. Rabenseifner, Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen, Ph.D. thesis, Universität Stuttgart, March, 2000.