Replay-based synchronization of timestamps in event traces of massively parallel applications

Daniel Becker[1,2], John C. Linford[3], Rolf Rabenseifner[4], and Felix Wolf[1,2]

[1]Forschungszentrum Jülich
Institute for Advanced Simulation
52425 Jülich, Germany
{d.becker, f.wolf}@fz-juelich.de

[2]RWTH Aachen University
Department of Computer Science
52056 Aachen, Germany

[3]Virginia Tech
Department of Computer Science
Blacksburg, VA 24061, USA
jlinford@vt.edu

[4]University of Stuttgart
High-Performance Computing-Center
70550 Stuttgart, Germany
rabenseifner@hlrs.de

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 show how the correction can improve the trace analysis of a real-world application example.*

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*. Inaccurate timestamps may also 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 [21] 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 [20], 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., 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 drift 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.

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 [4, 14, 5], it is crucial that the algorithm scales to large numbers of application processes. For this reason, we have designed and implemented a parallel version of the algorithm and integrated it into the SCALASCA performance-analysis framework [1]. Instead of sequentially processing a single global trace file, we follow SCALASCA's scalable trace-analysis approach [14] 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.

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 how our scheme can improve trace-analysis results on systems with insufficiently synchronized clocks. Finally in Section 6, we conclude our paper and give an outlook on future work.

2 Related Work

Network-based synchronization protocols aim at synchronizing distributed clocks before reading them. The distributed clocks query the global time from reference clocks, which are often organized in a hierarchy of servers. For instance, NTP [20] 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 in 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 Christian's probabilistic remote clock reading technique [6]. 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 [9] or at initialization and finalization [18], and are subsequently used as parameters of the linear correction function. While this scheme might prove satisfactory for short runs, 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 [16] has adopted Duda's algorithm for arbitrary processor topologies. In addition, Hofmann [15] has improved Duda's algorithm using a simple minimum/maximum strategy. Babaoğlu 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 in 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.

In contrast, logical synchronization uses happened-before relations among send and receive pairs to synchronize distributed clocks. Lamport has introduced a discrete logical clock [17] 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 [10, 11] and Mattern [19] have 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 Controlled Logical Clock

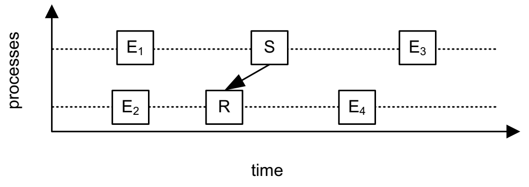
The *controlled logical clock* (CLC) algorithm by Rabenseifner [22, 23] retroactively corrects clock condition violations in event traces of message-passing applications by shifting message events in time while trying to preserve the length of intervals between local events. The algorithm restores the clock condition using happened-before relations derived from message semantics. The clock condition, given in Equation 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.

$$t_{recv} \geq t_{send} + l_{min} \tag{1}$$

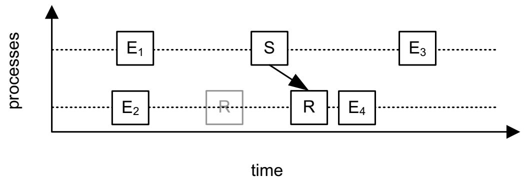
If the condition is violated for a send-receive event pair, the receive event is moved forward in time. To preserve the length of intervals between local events, events following or immediately preceding the corrected event are moved forward as well. These adjustments are called forward and backward 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 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 1(a) shows the initial event stream based on the measured timestamps with inefficiently 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 (Figure 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 both events (Figure 1(c)). 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 1(d).

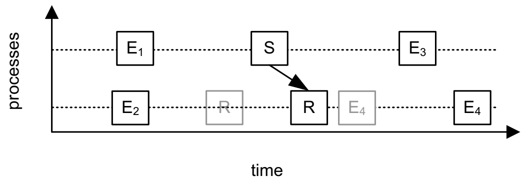
While the forward amortization is at least initially applied to all events following R, the backward amortization applies a linearly increasing correction to a limited amortization interval before R. However, in order to avoid new violations of the clock condition, the correction must not advance any 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, 23].



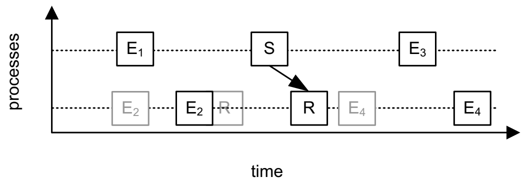
(a) Inconsistent event trace: clock condition violation in point-to-point communication pair.



(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.

Figure 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. Here, the CLC algorithm always tries to advance all processor clocks to the fastest clock when correcting the non-linearity of the clocks. Given that 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.

Since the original (CLC) algorithm takes only point-to-point messages into account, it has been extended by Becker et al. [3] to apply 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, especially 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 [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.

Integration with SCALASCA. Almost all the post-mortem 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 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.

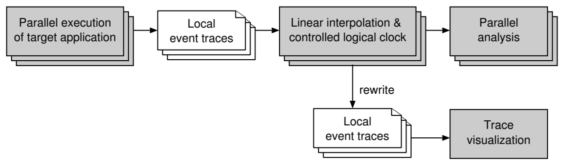

Parallel execution of target application
Local event traces
Linear interpolation & controlled logical clock
Parallel analysis
rewrite
Local event traces
Trace visualization

Figure 2. Parallel trace-analysis process.

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 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 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/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.

For the sake of simplicity, our current implementation uses two different values for the minimum message latency l_{min} (see Equation 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. 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 with the purpose of preserving the length of local intervals and avoiding an avalanche-like propagation of corrections [3]. Usually, γ_i^j is kept less than 1 minus the maximal drift of the clocks. 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 is below a predefined threshold. In practice, more than one pass is seldom needed.

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. A backward replay is needed to determine these upper limits. 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. In addition to what has already been stated in our initial design [3], 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.

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. 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 of the parallel controlled logical clock algorithm and also give evidence of the frequency and extent of clock condition violations in event traces of realistic MPI applications. For our experiments, we used the following three 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 measured MPI inter-node latency was $7.7 \mu s$, the measured MPI intra-node latency was $1.3 \mu s$.

JUMP consists of 41 IBM p690 nodes, each with 32 Power4+ processors running at 1.7 GHz. The measured MPI inter-node latency of $4.5 \mu s$, the measured MPI intra-node latency was $3.7 \mu s$.

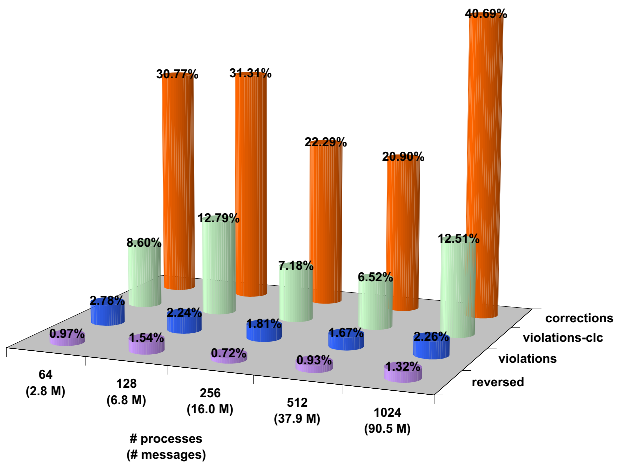
CACAU consists of 200 compute nodes with 400 Intel Xeon EM64T CPU's running at 3.2GHz. The measured MPI inter-node latency was $4.7 \mu s$, the measured MPI intra-node latency was $1.0 \mu s$.

The first application we tested was 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.

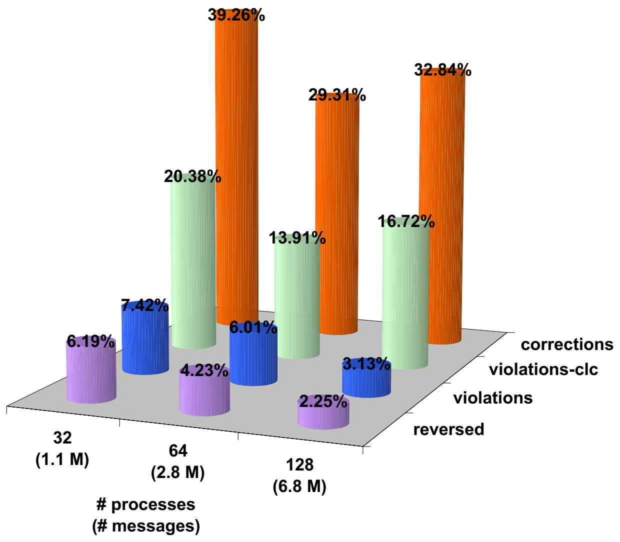
Table 1. Average and maximum errors of message events in reversed order.

Platform	Avg. error [μs]	Max. error [μs]
MareNostrum	32	323
Cacau	32	186

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 application 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.

Figure 3. Percentage of logical messages with the order of send and receive events being reversed, of logical messages with direct clock condition violations, of logical messages with clock condition violations detected by the CLC algorithm, and of event timings modified by the CLC algorithm.

Figure 3 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, of messages with clock condition violations detected by the CLC algorithm, and of event timings modified by the CLC algorithm. 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 violations detected by the CLC algorithms also include those that appear correct in the original trace, but turn into violations after preceding violations have been amortized and therefore require correction as well. On MareNostrum, around 1% of the messages flow backward in time, while on Cacau the percentage ranges between 2 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. As can be observed, a smaller number of inconsistent messages usually implies a larger number of corrections during amortization. Although the number of inconsistent messages on Cacau seems to decrease with growing numbers of processes, the results on MareNostrum do not confirm a clear correlation between the two factors. Table 1 lists the average and maximum displacement errors of message events in backward order, as seen in the original trace.

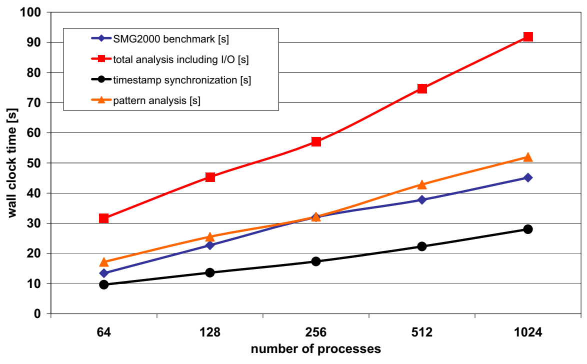


Figure 4. Scalability of parallel timestamp synchronization on MareNostrum.

According to Figure 4, the parallel timestamp synchronization, the SCALASCA pattern search, 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 pattern search) 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.

The second application we investigated was Metatrace [12], a multi-physics application based on MPI that simulates the spread of pollutants in groundwater. Metatrace combines two submodels, one to calculate the flow field of the groundwater and another to track individual particles in the precalculated flow field. Both models occupy disjoint subsets of the processors assigned to the application as a whole. Using Metatrace we demonstrate how clock condition violations can adversely affect the trace analysis and how the situation can be improved using our algorithm.

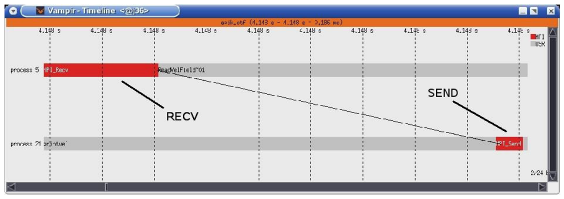
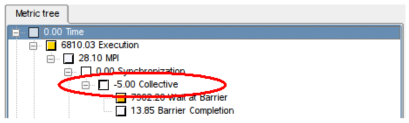


Figure 5. Time-line visualization of a message exchange in backward direction.

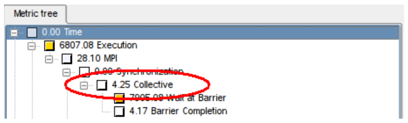
We ran the application with 64 processes on JUMP. Figure 5 contains the VAMPIR time line visualization of the original trace, showing two processes exchanging a message in backward direction. Without reading the names of the two communication operations, the user would most likely confuse sender and receiver. After applying our algorithm, all messages flow in the correct direction.



Metric tree

```
0.00 Time
  6810.03 Execution
    28.10 MPI
      0.00 Synchronization
        -5.00 Collective
          7902.20 Wait at Barrier
          13.85 Barrier Completion
```

(a) Non-synchronized.



Metric tree

```
0.00 Time
  6807.08 Execution
    28.10 MPI
      0.00 Synchronization
        4.25 Collective
          7905.08 Wait at Barrier
          4.17 Barrier Completion
```

(b) Synchronized.

Figure 6. SCALASCA output for non-synchronized and synchronized traces.

Additionally, the calculation of wait states performed by the SCALASCA pattern analysis relies on the correct logical order of message events. In Figure 6(a), the output of SCALASCA 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. By contrast, Figure 6(b) shows the SCALASCA output after timestamp synchronization. The result is now consistent.

6 Conclusion

The 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 results and confuse the user 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. 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.

With the ability to produce more accurate traces of long-running applications, we plan to support selective tracing of critical intervals in a more automated way based on a well-defined successive measurement refinement process. Finally, we want to extend our algorithm to hybrid applications employing a mix of message passing and shared-memory parallelism.

Acknowledgment. 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] Scalasca. `www.scalasca.org`.

[2] O. Babaoğlu and R. Drummond. (Almost) no cost clock synchronization. In *Proceedings of 7th International Symposium on Fault-Tolerant Computing*, pages 42–47. IEEE Computer Society Press, July 1987.

[3] D. Becker, R. Rabenseifner, and F. Wolf. Timestamp synchronization for event traces of large-scale message-passing applications. In *Proceedings of the 14th European PVM/MPI Conference*, pages 315–325, Paris, France, October 2007. Springer.

[4] H. Brunst and W. E. Nagel. Scalable performance analysis of parallel systems: Concepts and experiences. In *Proc. of the Parallel Computing Conference (ParCo)*, Dresden, Germany, 2003.

[5] A. Chan, W. Gropp, and E. Lusk. Scalable log files for parallel program trace data (draft). Technical report, Argonne National Laboratory, 2000.

[6] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1998. Springer Verlag.

[7] R. Drummond and O. Babaoğlu. Low-cost clock synchronization. *Distributed Computing*, 6(4):193–203, July 1993.

[8] 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, September 21-25, 1987*, pages 299–306. IEEE Computer Society Press, 1987.

[9] T. H. Dunigan. Hypercube clock synchronization. Technical Report ORNL TM-11744, Oak Ridge National Laboratory, TN, February 1991.

[10] C. J. Fidge. Timestamps in message-passing systems that preserve partial ordering. In *Proceedings of 11th Australian Computer Science Conference*, pages 56–66, February 1988.

[11] C. J. Fidge. Partial orders for parallel debugging. *ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[12] Forschungszentrum Jülich. *Solute Transport in Heterogeneous Soil-Aquifer Systems*. http://www.fz-juelich.de/icg/icg-iv/modeling.

[13] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, and B. J. Wylie. A parallel trace-data interface for scalable performance analysis. In *Proc. of the Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*, volume 4699 of *LNCS*, Umeå, Sweden, June 2006. Springer.

[14] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *Proc. 13th European PVM/MPI Conference*, Bonn, Germany, September 2006. Springer.

[15] R. Hofmann. Gemeinsame Zeitskala für lokale Ereignisspuren. In B. Walke and O. Spaniol, editors, *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, 7. GI/ITG-Fachtagung, Aachen, 21.-23. September 1993*. Springer-Verlag, Berlin, 1993.

[16] J.-M. Jézéquel. Building a global time on parallel machines. In J.-C. Bermond and M. Raynal, editors, *Proceedings of the 3rd International Workshop on Distributed Algorithms,* LNCS 392, pages 136–147. Springer-Verlag, 1989.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

[19] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard and P. Quinton, editors, *Proceedings of International Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France, October 1988*, pages 215–226. Elsevier Science Publishers B. V., Amsterdam, 1989.

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

[21] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer 63*, XII(1):69–80, January 1996.

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

[23] R. Rabenseifner. *Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen*. PhD thesis, Universität Stuttgart, March 2000.