Proceedings, Franck Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 315–325, Sep. 30 - Oct. 3, Paris, France, 2007.

Timestamp Synchronization for Event Traces of Large-Scale Message-Passing Applications

Daniel Becker[1], Rolf Rabenseifner[2], and Felix Wolf[1]

[1] Forschungszentrum Jülich, John von Neumann Institute for Computing (NIC)
52425 Jülich, Germany
{d.becker, f.wolf}@fz-juelich.de
www.fz-juelich.de
[2] University of Stuttgart, High-Performance Computing-Center (HLRS)
70550 Stuttgart, Germany
rabenseifner@hlrs.de
www.hlrs.de

Abstract. Identifying wait states in event traces of message-passing applications requires measuring temporal displacements between concurrent events. In the absence of synchronized hardware clocks, linear interpolation techniques can already account for differences in offset and drift, assuming that the drift of an individual processor is not time dependant. However, inaccuracies and drifts varying in time can still cause violations of the logical event ordering. 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 intervals between local events. 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 that is intended to scale to thousands of application processes and outline its implementation within the framework of the SCALASCA toolkit.

Key words: Performance analysis, event tracing, clock synchronization.

1 Introduction

Event tracing is a frequently applied technique for post-mortem performance analysis of message-passing applications because it can be used to analyze temporal relationships between concurrent activities. Obviously, the accuracy of such analyses depends on the comparability of timestamps taken on different processors. Inaccurate timestamps can not only cause a given interval to appear shorter or longer than it actually was, but also change the logical event order, which requires that a message can only be received after it has been sent. This is also referred to as the *clock condition*. To avoid violations of this condition, the error of timestamps should ideally be smaller than one half of the message latency.

Often, however, the clocks accessible from different processors are entirely non-synchronized or only synchronized within disjoint partitions (e.g., SMP-node

or multicore-chip). Clock synchronization protocols, such as NTP [4], can align the clocks to a certain degree, but are often not accurate enough for our purposes. Assuming that all local clocks on a parallel machine run at different but constant speeds (i.e., drifts), their time can be described as a linear function of the global time. This approach is used in the tracing library of the SCALASCA toolkit [2], which performs offset measurements between all local clocks and an arbitrarily chosen master clock once at program initialization and once at program finalization. However, as the assumption of constant drift is only an approximation, violations of the clock condition may still occur.

The *controlled logical clock* (CLC) [6] 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 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 that is intended to scale to thousands of application processes and outline its implementation design within the framework of SCALASCA [2], a performance-analysis tool that can be used to automatically identify idle times in event traces of large-scale message-passing programs.

The outline of this article is as follows: In Section 2, we start with a short description of SCALASCA's event model and its parallel trace analysis approach, followed by a review of the basic CLC mechanism in Section 3. In Section 4, we describe our extensions required to handle collective operations. After that, we present the new parallel algorithm design in Section 5. Finally in Section 6, we summarize our paper and give an outlook on future work.

2 Event Model and Replay-based Parallel Analysis

Because we plan to integrate the extended CLC algorithm with the SCALASCA trace-analysis tool, we describe it in terms of the SCALASCA event model, which is similar to the VAMPIR event model [5], for which the algorithm has been 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 anyway.

The information SCALASCA records for an individual event includes at least a 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 additional information (i.e., the communicator) that allows identifying concurrent collective exits belonging to the same collective operation instance. Table 1 illustrates the event sequences recorded for typical MPI operations.

To facilitate trace analysis for large numbers of application processes, the SCALASCA analyzer scans the trace data in parallel. After creating one analysis process per (target) application process, the analyzer loads the entire trace data into the potentially distributed main memory and performs a parallel replay of the applications communication behaviour, thereby examining each communication operation using an operation of similar type. During this procedure, the analyzer measures temporal differences both between remote and between local events, which requires the time stamps to be as accurate as possible. The execution time of the analyzer mainly depends on communication, which resembles the original communication of the target application. For details, please see [2].

Table 1. Exemplary 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

3 Controlled Logical Clock

Non-synchronized processor clocks may cause inaccurate timestamps in event traces. A clock condition violation occurs if the receive event of a message has an earlier timestamp than its matching send event. That is, the *happened-before* relation $e \rightarrow e'$ [6] between two events e and e' with their respective timestamps $C(e)$ and $C(e')$ does not hold. A *clock condition violation* between two events is defined as:

$$\exists\ e, e' : e \rightarrow e' \wedge C(e) \geq C(e'). \tag{1}$$

The CLC algorithm restores the clock condition using happened-before relationships between distributed events derived from point-to-point communication event semantics. More precisely, if the condition is violated for a send-receive event pair, the receive event is moved forward in time. The correction is only applied if the trace contains clock condition violations. To preserve the length of intervals between local events, events immediately following or preceding the corrected event are moved forward as well. This adjustment is 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 interpolation. In this section, we review the CLC algorithm including forward and backward amortization. The interested reader can find a detailed description of the CLC algorithm and a review of further synchronization approaches in [6] and [7].

3.1 CLC with Forward Amortization

The CLC algorithm is an enhancement of Lamport's logical clock [3] and was introduced by Rabenseifner [6]. The algorithm requires timestamps with limited errors, which can be achieved through weak pre-synchronization. To denote timestamps computed by CLC, we use the symbol LC'.

In the following, LC' is modeled with t as the wall clock time and $T(t)$ as the global time to which the process clocks $C_i(t)$ $(i = 0..n-1)$ are synchronized. Next, n is the number of processes, e_i^j is the j^{th} event on process i and so $E = \{e_i^j | i = 0..n-1, j = 0..j_{\max}(i)\}$ is the set of all events in the trace. In addition, the set of matching send and receive pairs is defined with

$$M = \{(e_k^l, e_m^n) | e_k^l \ = \ \text{send event}, \ e_m^n \ = \ \text{matching receive event}\}. \tag{2}$$

Note that the send event always marks the beginning of a send operation whereas a receive event marks the end of a receive operation. By contrast, e_i^j is an internal event if it is neither a send nor a receive event. Furthermore, δ_i is the minimal difference 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, LC_i' is now defined as

$$LC_i'(e_i^j) := \begin{cases} \max(LC_k'(e_k^l) + \mu_{k,i}, \\ \qquad LC_i'(e_i^{j-1}) + \delta_i, \\ \qquad LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), \\ \qquad C_i(t(e_i^j))) \qquad \text{if } \underset{e_k^l}{\exists} (e_k^l, e_i^j) \in M \end{cases} \tag{3}$$

$$\begin{cases} \max(LC_i'(e_i^{j-1}) + \delta_i, \\ \qquad LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), \\ \qquad C_i(t(e_i^j))) \qquad \text{otherwise.} \end{cases} \tag{4}$$

As can be seen, the algorithm consists of two equations. Equation (3) adjusts the timestamps of receive events while Equation (4) modifies timestamps of internal and send events. Note that for each process, the terms $LC_i'(e_i^{j-1}) + \delta_i$ and $LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1})))$ must be omitted for the first event $(j = 0)$.

Through the term $C_i(t(e_i^j))$ in Equation (3) and Equation (4), the algorithm ensures that a correction is only applied if the trace violates the clock condition. The new timestamps satisfy the clock condition, since the term $LC_k'(e_k^l) + \mu_{k,i}$ in Equation (3) ensures that $LC'(e_i^j)$ is put forward compared to $C_i(t(e_i^j))$ if needed in case of a clock condition violation. To ensure that the clock does not stop after a clock condition violation, the term $LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))$ in Equation (3) and Equation (4) approximates the duration of the

original communication after a clock condition violation. This mechanism is called forward amortization.

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 [7]. 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, i.e., the controller estimates the output error indirectly because $T(t(e_i^j))$ is unknown. If $1-\gamma$ is chosen smaller than the maximal drift differences, the controller will enlarge $1-\gamma$ (e.g., to 1%) to ensure 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. Mainly, γ_i^j is kept less than 1 minus the maximal 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 $LC'_i(e_i^{j-1}) + \delta_i$ in Equation (3) and Equation (4) 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 [7].

A jump discontinuity in LC' of Δt is caused by the term $LC'_k(e_k^l) + \mu_{k,i}$ in Equation (3) if $LC'(e_i^j)$ of the violating receive event is put forward compared to $C_i(t(e_i^j))$. The term $LC'_i(e_i^{j-1}) + \gamma_i^j(C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))$ in Equation (3) implements a 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 the first part of the CLC algorithm. This is done 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 [7] (Figure 1). The compensation is realized by setting the timestamps forward. If there are no violating send events in the backward amortization interval of a process i, then the dash-dotted linear interpolation can be used. In Figure 1, 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 event r. 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. Apparently, adjacent clock condition violations cause a larger perturbation.

In addition, not to violate 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 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 Figure 1. As can be

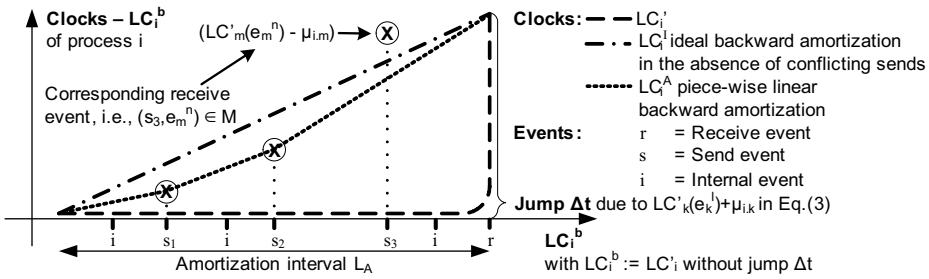

Fig. 1. Algorithm of the backward amortization.

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 amortization 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 Extended Controlled Logical Clock

Unfortunately, the CLC algorithm in its present state is only designed to correct clock condition violations related to point-to-point communication. Collective communication semantics are ignored. In this section, we extend the algorithm including forward and backward amortization to correctly handle collective communication. Again, we start with considering happened-before relationships among collective communication events. We start with a description of the extended forward amortization followed by the extended backward amortization.

4.1 Extended CLC with Forward Amortization

The CLC algorithm requires the detection of clock condition violations. The happened-before relation is used to synchronize the timestamp of the receive event with the timestamp of the corresponding send event, i.e., the receive event is put forward in time if a clock condition violation has occurred.

A single collective operation can be considered as a composition of many point-to-point communications. Using this model, we determine collective send and receive pairs occurring during a collective operation instance. We distinguish several types of collective operations (e.g., 1-to-N, N-to-1, etc.). Depending on the type, some of the enter events in a collective operation instance can be regarded as send events and some of the collective exit events as receive events.

In the following, 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 send and receive events in a collective operation instance i, respectively. For each call

to a collective operation, the set of all send-receive pairs M is enlarged by adding $S \times R$.

1-to-N: One root process sends its data to N other processes. Example 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 of the communicator (i.e., all collective exit events) with a data length greater zero, i.e., 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) on all processes of the communicator with a data length greater zero. Given that the root process is not allowed to exit the operation until the last process enters the operation, 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 Equation (3) must be replaced by the maximum of $LC'_k(e^l_k) + \mu_{k,i}$ over all $e^l_k \in S$. That is, Equation (3) must be replaced by

$$LC'_i(e^j_i) := \begin{cases} \max(\ (\max_{\forall e^l_k \text{with} (e^l_k, e^j_i) \in M} LC'_k(e^l_k) + \mu_{k,i}), \\ \qquad LC'_i(e^{j-1}_i) + \delta_i, \\ \qquad LC'_i(e^{j-1}_i) + \gamma^j_i (C_i(t(e^j_i)) - C_i(t(e^{j-1}_i))), \\ \qquad C_i(t(e^j_i))) \quad \text{if } \exists\limits_{e^l_k} (e^l_k, e^j_i) \in M & (3') \\ \\ \ldots \quad \text{otherwise.} & (4) \end{cases}$$

N-to-N': All processes of the communicator are 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. For a call to `MPI_Barrier`, all processes of 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 $S \times R$. Below, e^l_k refers to the enter event of a collective operation instance and e^j_i refers to the collective exit event and, thus, the set of messages added to M has the form

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

with $x = 0$ for `MPI_Scan` and $x = 1$ for `MPI_Exscan`.

Independently of collective operation type, it is important to optimize the handling of $S \times R$ in Equation (3'). A parallelized algorithm of the extended CLC should attempt to reduce the effort to $\mathcal{O}(\log N)$.

4.2 Extended Backward Amortization

To extend the backward amortization algorithm for collective routines, the upper bounds for the send events (see Figure 1) 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 receive event data set defined in Section 4.1.

5 Parallel Timestamp Synchronization

Event tracing of applications running on thousands of processes [8] requires a scalable synchronization scheme. In this section, we present a parallel version of the extended CLC algorithm.

5.1 Pre-Synchronization

The accuracy of the CLC algorithm depends on the accuracy of the original timestamps and therefore a pre-synchronization is required. This can be achieved through a linear interpolation where all process-local clocks are mapped onto a single master clock. Given that different clocks vary in offset and drift, offset values between worker processes and one master process measured at program start and at program end are used to find a linear correction function. The offset values are measured using the remote clock reading technique introduced by Cristian [1]. As a byproduct, the minimum transfer delay can be estimated during the offset measurements.

5.2 Parallel Post-Mortem Timestamp Synchronization

SCALASCA's replay-based approach of analyzing separate process-local trace files in parallel can handle traces from thousands of processes. We can achieve comparable scalability for the CLC algorithm if we also implement it using a parallel replay. This has the additional advantage that it can be seamlessly integrated into the existing analysis framework.

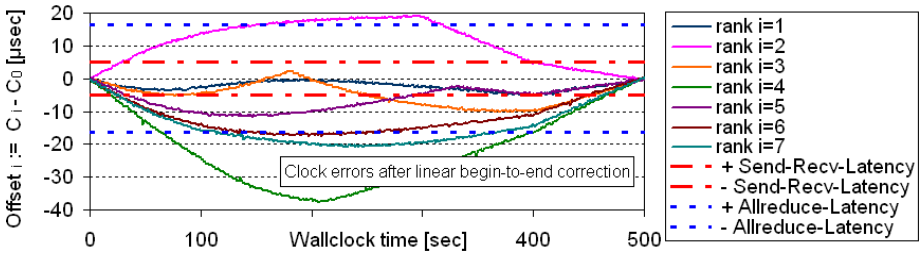

Fig. 2. Non-linear drifts of physical clocks measured on an Infiniband cluster in comparison to Send-Recv and Allreduce latency.

Preparation: While each SCALASCA analysis process reads the local trace file of the corresponding application process into memory, the linear correction is applied to all timestamps based on the previous offset measurements at program start and end. The resulting timestamps are taken as the C_i. Inaccurate C_i can occur for two reasons: (i) inaccurate offset measurements and (ii) time-dependant clock drift. Figure 2 shows the non-linear behavior of the clocks C_i after such linear correction on an INFINIBAND cluster. Clock errors are still significantly larger than point-to-point and collective latencies, i.e., violations of the clock condition can still occur.

Logical clock synchronization algorithm: To apply the extended CLC algorithm, a parallel traversal of the event stream is performed. Whenever reaching communication events, the corresponding communication operation is replayed to exchange the timestamps of communication events for their later comparison. For each event, a new timestamp is calculated using the extended CLC algorithm. The comparison between the remote timestamp and the local timestamp is used to find clock condition violations. Depending on the type of the original communication operation, different timestamps are exchanged using different MPI function calls, as listed in Table 2.

Table 2. Timestamps exchanged depending on the type of operation during forward amortization.

Type of operation	**timestamp exchanged**	**MPI function**
P2P	timestamp of send event	`MPI_Send`
1-to-N	timestamp of root enter event	`MPI_Bcast`
N-to-1	max(all enter event timestamps)	`MPI_Reduce`
N-to-N'	max(all enter event timestamps)	`MPI_Allreduce`
`MPI_Scan`	max(some enter event timestamps)	`MPI_Scan`
`MPI_Exscan`	max(some enter event timestamps)	`MPI_Exscan`

Note, that in Equation (3'), the parallel calculation of the maximum over all corresponding send events ($\max_{\forall e_k^l \text{with}(e_k^l, e_i^j) \in M} LC'_k(e_k^l) + \mu_{k,i}$) in the case of N-to-1, N-to-N', `MPI_Scan`, and `MPI_Exscan` can not be implemented with the MPI function identified in Table 2 if $\mu_{k,i}$ is not the same for all pairs of processes. Therefore in Equation (3'), $\mu_{k,i}$ must be substituted by $min_{\forall k,i}(\mu_{k,i})$. The exchanged timestamps are based on the LC' values calculated up to the specific event.

The control mechanism used for the controlled logical clock requires a global view of the trace data to calculate γ_i as described in Section 3. Establishing a global view of the trace data is not feasible with the replay-based approach since communication would be required for each single event. Therefore, we eventually have to perform multiple passes until the maximum error e is below a predefined threshold ϵ. For the first pass through the trace files, we propose to use $\gamma = const < 1$, for subsequent passes a $\gamma_{j+1} < \gamma_j$ should be used.

Backward amortization algorithm: The backward amortization requires a second replay of the target application's communication behavior. Timestamps are

Table 3. Timestamps exchanged depending on the type of operation during backward amortization.

Type of operation	timestamp exchanged	MPI function
P2P	timestamp of receive event	`MPI_Send`
1-to-N	min(all collective exit event timestamps)	`MPI_Reduce`
N-to-1	timestamp of root collective exit event	`MPI_Bcast`
N-to-N	min(all collective exit event timestamps)	`MPI_Allreduce`
`MPI_Scan`	min(some collective exit event timestamps)	`MPI_Scan`
`MPI_Exscan`	min(some collective exit event timestamps)	`MPI_Exscan`

exchanged at synchronization points of the application. However, as explained in Section 3, the former sender now needs data from the former receiver and so the roles between sender and receiver are switched during the backward amortization. Depending on the type of operation, the collective receiver needs the timestamp of the relevant collective send event which are shown in Table 3. For `MPI_Scan` and `MPI_Exscan`, a communicator with reverse rank ordering must be used. The exchanged timestamps are based on the LC' values after completion of the extended CLC algorithm. After receiving the data, each process temporally stores the timestamps to locally apply the backward amortization if LC' exhibits a jump disconutinity. Note that this happens after the forward amortization has already been applied.

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 could be replaced by a single collective operation per communicator for the entire trace - provided that sufficient memory is available.

6 Conclusion

In this paper, we have extended the CLC algorithm to take collective communication semantics into account so that now a more complete correction of realistic message-passing traces can be achieved. Although the extended CLC 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. In this case, the decomposition into (additional) send and receive events is naturally given.

Finally, we have presented a design how the previously sequential algorithm can de parallelized and implemented within the framework of the SCALASCA toolkit. Once we have completed the actual implementation, we will perform a detailed quantitative evaluation using real message-passing codes.

References

1. F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1998. Springer Verlag.
2. 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.
3. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
4. D. L. Mills. Network Time Protocol (Version 3). The Internet Engineering Task Force - Network Working Group, March 1992. RFC 1305.
5. W. Nagel, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
6. 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.
7. R. Rabenseifner. *Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen.* PhD thesis, Universität Stuttgart, March 2000.
8. Felix Wolf, Felix Freitag, Bernd Mohr, Shirley Moore, and Brian Wylie. Large event traces in parallel performance analysis. In *8th Workshop Parallel Systems and Algorithms (PASA), Lecture Notes in Informatics*, Frankfurt/Main, Germany, March 13-16 2006. Gesellschaft für Informatik.
http://icl.cs.utk.edu/projectsfiles/kojak/pubs/pasa06.pdf.