Cluster Comput (2013) 16:171–189
DOI 10.1007/s10586-011-0181-8

Extending the scope of the controlled logical clock

Daniel Becker · Markus Geimer · Rolf Rabenseifner · Felix Wolf

Received: 21 July 2011 / Accepted: 11 August 2011 / Published online: 23 September 2011

Abstract Event traces are helpful in understanding the performance behavior of parallel applications since they allow the in-depth analysis of communication and synchronization patterns. However, the absence of synchronized clocks on most cluster systems 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 or confuse the users of time-line visualization tools by showing messages flowing backward in time. In our earlier work, we have developed a scalable algorithm called the *controlled logical clock* that eliminates inconsistent inter-process timings postmortem in traces of pure MPI applications, potentially running on large processor configurations. In this paper, we first demonstrate that our algorithm also proves beneficial in computational grids, where a single application is executed using the combined computational power of several geographically dispersed clusters. Second, we present an extended version of the algorithm that—in addition to message-passing event semantics—also preserves and restores shared-memory event semantics, enabling the correction of traces from hybrid applications.

Keywords Event tracing · Timestamp synchronization

D. Becker (✉) · F. Wolf
German Research School for Simulation Sciences, 52062 Aachen, Germany
e-mail: d.becker@grs-sim.de

F. Wolf
e-mail: f.wolf@grs-sim.de

M. Geimer · F. Wolf
Jülich Supercomputing Centre, 52425 Jülich, Germany

M. Geimer
e-mail: m.geimer@fz-juelich.de

R. Rabenseifner
University of Stuttgart, 70550 Stuttgart, Germany
e-mail: rabenseifner@hlrs.de

F. Wolf
RWTH Aachen University, 52056 Aachen, Germany

1 Introduction

One technique widely used by cluster tools is event tracing with a broad spectrum of applications ranging from performance analysis [40], performance prediction [32] and modeling [46] to debugging [27]. Recording time-stamped runtime events in event traces is especially helpful for understanding parallel performance because it enables the analysis of temporal relationships between concurrent activities. Obviously, the accuracy of such analyses depends on the comparability of timestamps taken on different processors. Measuring the time between concurrent events necessitates either a global clock or well-synchronized processor clocks. While some custom-built clusters such as IBM Blue Gene offer sufficiently accurate global clocks, most commodity clusters provide only processor clocks that are either entirely non-synchronized or synchronized only within disjoint partitions (e.g., SMP nodes). Moreover, external software synchronization via NTP is usually not accurate enough for the purpose of event tracing [38]. Assuming that potentially different drifts of local clocks remain constant over time, linear offset interpolation can be applied to map local onto global timestamps. However, given that the drift of realistic clocks is usually time dependent, the error of timestamps derived in this way can easily lead to a misrepresentation of the logical event order imposed by the semantics of the underlying communication substrate [6]. 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 by showing messages flowing backward in time.

In our earlier work [7], we have introduced a scalable timestamp-synchronization algorithm that eliminates inconsistent inter-process timings postmortem in traces of pure MPI applications. This algorithm, the most recent version of the *controlled logical clock* (CLC) [44], restores the consistency of inter-process event timings based on happened-before relations imposed by point-to-point and collective MPI event semantics. Scalability is ensured by performing the corrections for individual processes in parallel while replaying the original communication recorded in the trace. The algorithm has been integrated into the Scalasca trace-analysis framework [22]. In this paper, we extend the scope of our algorithm and its implementation within Scalasca in two ways.

We first demonstrate that the algorithm can be employed also in grid environments that allow geographically dispersed clusters to be used as a single coherent system, an arrangement also known as a *metacomputer*. This is needed if the computational power of any single cluster available to an application is insufficient for the computational task at hand. A particular challenge to be addressed before the algorithm can be applied is the accurate measurement of clock offsets across a hierarchical network with different latency levels. These measurements are a prerequisite of the linear interpolation performed before applying our algorithm to improve the quality of the timestamps delivered as its input.

Second, we extend the algorithm towards hybrid applications. To exploit the memory of shared-memory nodes with larger numbers of cores more efficiently, many code developers now resort to using OpenMP for node-internal work sharing, while employing MPI for parallelism among different nodes, making hybrid programming increasingly popular. However, the current version of the CLC algorithm is unsuitable for hybrid codes because it neither restores nor preserves happened-before relations between shared-memory events. To remove this limitation, we (i) identify happened-before relations in OpenMP constructs and library calls and integrate them into the existing algorithmic framework and (ii) extend the parallel replay mechanism such that it can replay traces from hybrid codes.

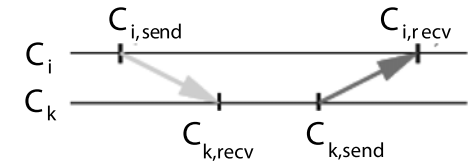
The remainder of this article is organized as follows: After reviewing related work in Sect. 2, we introduce the pure MPI version of the CLC algorithm along with its limitations in Sect. 3. In Sect. 4, we describe the infrastructure needed to run it on a metacomputer. In addition, we validate the behavior of our algorithm in such an environment with respect to (i) accuracy, showing that the collaterally introduced deviations of local interval lengths remain within acceptable limits, and (ii) scalability. In Sect. 5, we present the extensions required for correctly synchronizing the timestamps that occur in hybrid programs during the execution of OpenMP constructs. Here, we describe the hybrid parallelization of the extended algorithm and its implementation within Scalasca, again followed by an experimental evaluation of the algorithm's accuracy and scalability. Finally, in Sect. 6 we summarize our results and outline future work.

2 Related work

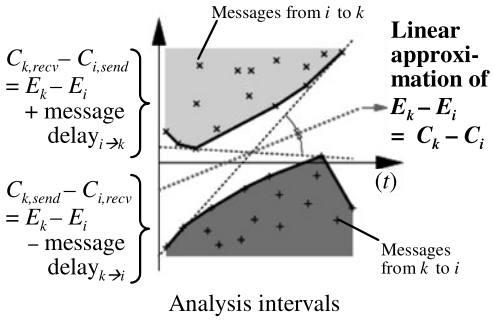
In this section we cite several approaches for avoiding or correcting inconsistent timestamps, applied either online or postmortem. Network-based synchronization protocols aim at synchronizing distributed clocks before reading them. The clocks query the global time from reference clocks, which are often organized in a hierarchy of servers. For instance, NTP [38] 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 (i.e., the rate at which the offset changes over time) 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 guarantee the correct total event order of event traces taken on clusters equipped with modern interconnect technology.

Time differences among distributed clocks can be characterized in terms of their relative offset and drift. 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 [11]. 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 [16] or at initialization and finalization [36], 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 [12]. While linear offset interpolation might prove satisfactory for short runs (or interpolation intervals), measurement errors and time-dependent drifts may create inaccuracies and violate happened-before relations during longer runs [6]. Additionally, repeated drift adjustments caused by NTP may impede linear interpolation, as they deliberately introduce non-constant drifts.

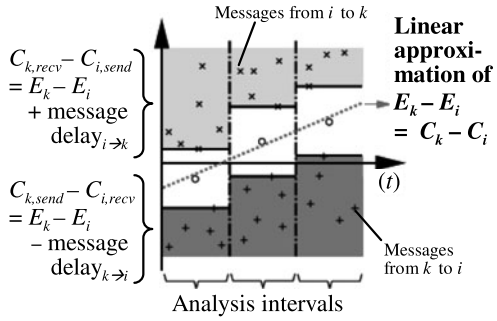
If linear interpolation alone turns out to be inadequate to achieve the desired level of accuracy, error estimation allows



(a) Time lines of two processes exchanging messages



(b) Duda et al



(c) Hofmann

Fig. 1 Duda et al. and Hofmann calculate clock errors through the differences of message transfer times in both directions between two processes

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 et al. [15] to determine the smoothing function. Using a minimal spanning tree algorithm, Jezequel [30] adopted Duda's algorithm for arbitrary processor topologies. In addition, Hofmann [25] improved Duda's algorithm using a simple minimum/maximum strategy and further proposed that the execution time should be divided into several intervals to compensate for different clock drifts in long running applications. Figure 1 shows the principles underlying Duda's and Hofmann's algorithms with two processes exchanging messages. Figure 1(a) shows the time lines of two processes i and k along with the process-local clock values C_i and C_k. The clock errors E_i and E_k of the process-local clocks can be described as the difference between the local clock values and the physical time t (i.e., wall-clock time), respectively. As can be seen, message arrows from process i to k are shown in light gray, whereas message arrows from process k to i are shown in dark gray. In Figs. 1(b) and 1(c), clock differences $C_{k.recv} - C_{i.send}$ of messages from process i to k are located in the upper part, whereas clock differences $C_{k.send} - C_{i.recv}$ of messages from process k to i are located in the bottom part. These differences are equal to the clock error differences plus (upper part) or minus (bottom part) the individual message delays. Any line between these areas is an approximation of the clock error differences. Using such an approximation function to correct clock values guarantees the logical event order. In Fig. 1(b), these time differences are enclosed by their convex hulls (i.e., light gray and dark gray areas). In addition, two of the dotted lines represent the lines with the largest and smallest possible slope between both convex hulls. Duda et al. calculate the approximation function as the interior bisector of the angle between the two linear functions, shown as the third dotted line. As can be seen in Fig. 1(c), Hofmann reduced the computational effort by introducing distinct analysis intervals, in which the convex hulls are determined by a simple minimum/maximum strategy.

Hofmann and Hilgers [26] also simplified Jezequel's algorithm for handling multi-processor topologies with a shortest path algorithm from graph theory. A modification aimed at handling cases of non-existing communication relations between some of the application processes is described in [45]. Biberstein et al. [8] rewrote Hofmann and Hilgers's algorithm for use on the Cell BE architecture [10] using a short and intelligible notation. Their version solves the clock condition problem only for short intervals (i.e., without splitting them into sub-intervals for handling non-linear drifts of physical clocks). Babaoğlu and Drummond [2, 14] have shown that clock synchronization is possible at minimal cost if the application makes a full message exchange between all processors at sufficiently short intervals. However, jitter in message latency, nonlinear relations between message latency and message length, and one-sided communication topologies limit the usefulness of error estimation approaches. References to additional error estimation approaches can be found in a survey by Yang and Marsland [52].

In contrast, logical clock synchronization uses happened-before relations among send and receive pairs to synchronize distributed clocks. Lamport introduced a discrete logical clock [33] 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

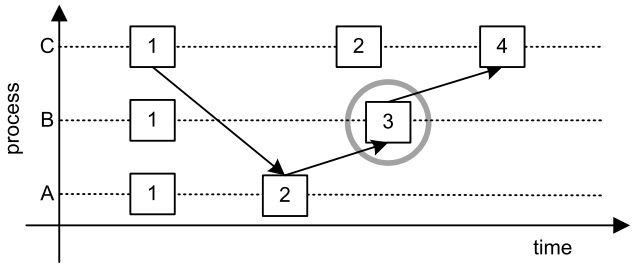


Fig. 2 Lamport's discrete logical clock: the clock value of the encircled event on process *B* is updated based on the maximum of both the incremented local clock value on process *B* and the incremented clock value of the sending event on process *A*

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. Figure 2 illustrates the different steps of Lamport's logical clock using a simple example consisting of three processes exchanging messages. The figure shows the time lines of the three processes along with different events and messages. While events are depicted as small squares, messages are shown as arrows pointing in the direction of the communication. The clock value is shown for each event in the center of its square. As can be seen, local clock values are incremented after every local event. The incremented values of send events are sent along with the message. The clock value of a receive event is calculated as the maximum of the incremented local clock value and the incremented clock value of the corresponding send event. For instance, the encircled event on process B (Fig. 2) is the second local event on that process and so its local clock value would be 2. Given that this event receives a message from process A, its clock value is updated based on both the local and remote clock value. Here, the incremented local clock value is 2 and the incremented clock value of the send event on process A is 3. Since the maximum of both is 3, the new value is set to 3. While Lamport's logical clock preserves the relative order of events, it does not account for different temporal distances between events. In particular, events happening at different wall-clock times may have the same logical clock value, as can be seen in Fig. 2 for the second events of processes A and C. Therefore, Lamport's logical clock cannot be used for certain performance-analysis applications such as measuring wait states.

As an enhancement of Lamport's discrete logical clock, Fidge [18, 19] and Mattern [37] proposed a vector clock. In their scheme, each processor maintains a vector representing all processor-local clocks. While the local clock is advanced with each local event as before, the local vector is updated after receiving a message using an element-wise maximum operation between the local vector and the remote vector that has been sent along with the message. The vector clock is used in some monitoring tools [17, 49] and, in a modified form, to distinguish in event traces between primary wait states and secondary ones that are merely caused by propagation [28]. Furthermore, global events (e.g., breakpoints) are introduced in [23], while in [43] spontaneous events (e.g., collisions on a network) are taken into account. Limits of the logical clock and the vector clock are eventually illustrated in [47].

Finally, Rabenseifner's controlled logical clock (CLC) algorithm [44], which is the subject of this paper, 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, which was recently extended and parallelized by Becker et al. [7], restores the clock condition using happened-before relations derived from both point-to-point and collective MPI event semantics. Starting from the parallel MPI version of the algorithm, this paper describes its application in grid environments and its hybridization, retaining its good accuracy and scalability characteristics.

3 Controlled logical clock

Recording time-stamped runtime events in event traces is especially helpful for understanding the parallel performance behavior because it enables the postmortem analysis of communication and synchronization patterns. For instance, timeline browsers such as Vampir [40] allow these patterns to be visually explored, while the Scalasca toolset [22] scans traces automatically for wait states that occur when processes or threads fail to reach synchronization points in a timely manner, for example, as a result of unevenly distributed workloads. Because we focus on a scalable timestamp synchronization method to be used within the Scalasca trace-analysis toolset, we further explain Scalasca's analysis workflow, which is depicted in Fig. 3. To ensure scalability of the wait-state search, the traces are scanned in parallel using as many processors as have been used to execute the target application itself. After loading the process-local traces into the potentially distributed main memory of the machine, Scalasca traverses them simultaneously while replaying the original communication recorded in the trace to exchange information relevant to the search. To increase the accuracy of this analysis on clusters without global clock, Scalasca applies the parallel CLC algorithm after the traces have been loaded and before the wait-state analysis takes place. To optimize the fidelity of the correction, the timestamps first undergo a pre-synchronization step, which performs linear offset interpolation based on offset measurements taken during initialization and finalization of the target application. As an alternative to the wait-state search, the corrected traces can also be rewritten and visualized using a third-party time-line browser such as Vampir.

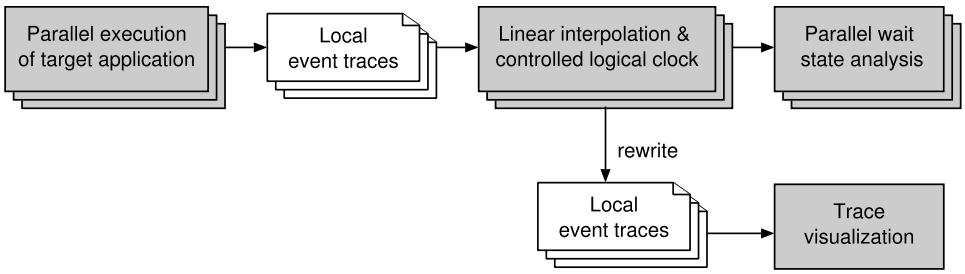


Fig. 3 Parallel trace-analysis workflow in Scalasca. *Gray rectangles* denote programs and white rectangles with the upper right corner turned down denote files. *Stacked symbols* indicate multiple instances of programs or files running or being processed in parallel

Table 1 Event sequences recorded for typical MPI and OpenMP operations

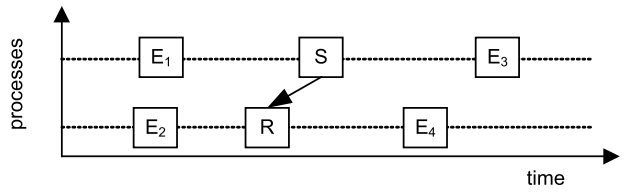
Operation name	Event sequence
MPI	
`MPI_Send()`	(enter, send, exit)
`MPI_Recv()`	(enter, receive, exit)
`MPI_Allreduce()`	(enter, MPI collective exit) for each participating process
OpenMP	
`parallel` construct	(fork, enter, OpenMP collective exit, join) for the participating master thread
	(enter, OpenMP collective exit) for each participating worker thread
Implicit and explicit barrier	(enter, OpenMP collective exit) for each participating thread
`omp_set_lock`	(enter, lock-acquisition, exit)
`omp_unset_lock`	(enter, lock-release, exit)
`critical` construct	(lock-acquisition, enter, exit, lock-release)
`atomic` construct	(enter, exit)

As the algorithm's foundation, the information Scalasca records for an individual event includes at least the timestamp, the location (e.g., the process or thread) 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 and OpenMP operations. MPI-related events include events representing point-to-point operations, such as sending and receiving messages, and an event representing the completion of collective MPI operations. OpenMP-related events, which are fashioned according to the POMP event model [39], include events that represent the creation and termination of a team of threads, leaving a parallel or barrier region, and acquiring or releasing lock variables. A fork event indicates that the master thread creates a team of threads (i.e., workers) and a join event record indicates that the team of threads is terminated. In addition, the collective OpenMP exit event indicates that the program leaves either a parallel or a barrier region. Furthermore, a lock-acquisition event indicates that a lock variable is set, whereas a lock-release event indicates that this variable is unset. Nested parallelism and tasking is not yet supported in POMP, although an extension for tasking is already in preparation [34]. Event sequences recorded for typical MPI and OpenMP operations are given in Table 1. In preparation of its extensions, we now briefly recapitulate the basic principles of the CLC algorithm in the remainder of this section, where we explain how it is currently used to synchronize the timestamps of pure MPI applications.

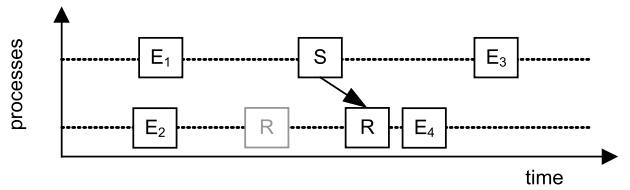
In general, clock errors may have both quantitative and qualitative effects. The first category includes changing the absolute position of an event in the trace or the distance between two consecutive events. As shown in a recent study [6], the second category of effects, which manifests as a change of the logical event order, are also very common as soon as an application is traced for more than a few minutes. If an event e happened before another event e', the *happened-before* relation $e \rightarrow e'$ between both events requires that their respective timestamps $C(e)$ and $C(e')$ satisfy the *clock condition* [33]:

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

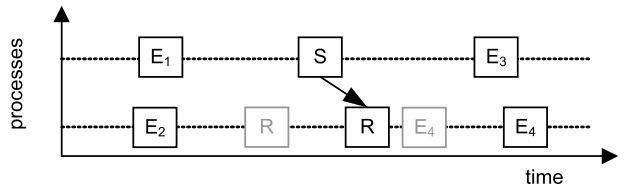
The equation given above can be refined by requiring a temporal minimum distance (i.e., latency) between the two events—if its amount is known. While the errors of single timestamps are hard to assess, obvious violations of the clock condition between events with a logical happened-before relation, such as sending and receiving a message, can be easily detected and offer a toehold to increase the fidelity of inter-process timings. If the clock condition is vi-



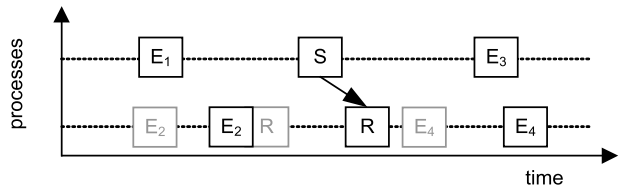
(a) Inconsistent event trace: Clock condition violation in point-to-point communication pair



(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. 4 Backward and forward amortization in the controlled logical clock algorithm

olated for a send-receive event pair, the receive event is corrected (i.e., moved forward in time). To preserve the length of intervals between local events, events following or immediately preceding the corrected event are also adjusted. These adjustments are called *forward* and *backward* amortization, respectively.

Figure 4 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). Figure 4(a) 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. 4(b)), with l_{min} being the minimum message latency. Because the distance between R and E_4 is now too short, E_4 is adjusted during the forward amortization to preserve the length of the interval between the two events (Fig. 4(c)). However, the jump discontinuity introduced by adjusting R affects not only events later than R but also events earlier than R. This is corrected during the backward amortization, which shifts E_2 closer to the new position of R (Fig. 4(d)). As can be seen in this example, the algorithm only moves events forward in time.

Moreover, happened-before relations also exist among the constituent events of collective MPI operations. The algorithm considers a single collective message-passing operation as being composed of multiple point-to-point operations, taking the semantics of the different flavors of such operations into account (e.g., *1-to-N*, *N-to-1*, ...). For instance, let us consider an *N-to-1* operation such as gather where 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. Because the algorithm synchronizes the timestamps of concurrent events using happened-before relations, the respective "receive" event is put forward in time whenever the matching "send" event appears too late in the trace to satisfy the clock condition. In reference to the fact that this method is based on logical clocks, the send and receive event types assigned during this mapping are called the *logical event types* as opposed to the actual event types (e.g., enter, collective exit) specified in the event trace. The logical event type can usually be derived from the name of the MPI operation and the role a process plays in it. For the sake of simplicity, the current implementation uses two different values for the latency: the inter-node and the intra-node latency. Following a conservative approach aimed at avoiding overcorrection, an extra collective latency was not considered, 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.

Although the CLC algorithm removes residual inconsistencies (i.e., those left after applying linear offset interpolation) in event traces of MPI applications postmortem, it is limited by the following factors:

- The offset interpolation, as discussed in Sect. 2, does not account for the hierarchy of latencies as found in a metacomputer and thus may generate timestamps not suitable as input data for the CLC algorithm [4].
- The current implementation of the CLC algorithm does not account for inter-machine latencies during wide-area communications as found in a metacomputer.
- The CLC algorithm does not account for direct violations of shared-memory event semantics in the original trace.

Although rare, instances of such violations have been reported [6].
– Probably most important, the CLC algorithm does not preserve happened-before relations in shared-memory operations, because the constituent events of such constructs are currently treated as events not involved in happened-before relations with events of other threads. Thus, the restoration of message-passing semantics may introduce violations of shared-memory event semantics even though they were not violated in the original trace.

For the reasons mentioned above, the current CLC algorithm is not suitable for metacomputer applications that use distributed clusters simultaneously and hybrid cluster applications that use MPI and OpenMP in combination.

To address the limitations for metacomputing applications, Sect. 4 describes the necessary infrastructure changes for metacomputing environments and demonstrates that the CLC algorithm is beneficial in such an environment. Finally, to address the limitations for hybrid cluster applications, Sect. 5 describes the algorithmic extensions required to restore and preserve not only point-to-point and collective message-passing but also shared-memory event semantics, which are relevant for hybrid codes. Since rapidly increasing parallelism demands that this correction scales to large numbers of processes and threads, this section also shows how the hybrid version is parallelized and integrated into the scalable Scalasca trace-analysis framework.

4 Metacomputer

The solution of critical numerical problems may require more processing power and memory capacity than is available on a single cluster. Often, coupling multiple independent clusters (i.e., metahosts) to form a more powerful metacomputer [48] is the only viable method to increase the resources available for a single application. Although applications can benefit from the increased parallelism offered by a metacomputer, as supported by a study by Wong and Goscinski [51], achieving satisfactory application performance is difficult. In particular, algorithm design must account for the hierarchies of latencies and bandwidths in addition to the heterogeneous hardware architectures found in such environments. Hence, performance optimization is a crucial but non-trivial task that needs adequate tool support. For example, wait states introduced as a result of using a metacomputer can be identified via automatic pattern search in event traces, an analysis mode offered by Scalasca. In fact, Scalasca has already been used to optimize the performance of the multi-physics code MetaTrace on the heterogeneous and geographically dispersed metacomputing testbed Viola by more than a factor of two [5].

4.1 Infrastructural extensions

Because its distributed memory and processing scheme can establish the global view of trace data in the absence of a global file system, the parallelized CLC algorithm is well suited for increasing the accuracy of trace-analysis tools in computational grids. However, the algorithm necessitates timestamps with limited errors, which can be achieved through linear offset interpolation between program start and end.

Scalasca's simple linear offset interpolation mechanism, as discussed in Sect. 3, is inaccurate because of the network links between different metahosts, whose latencies may be an order of magnitude larger than those of the internal networks. As a consequence, offset measurements across these links are less accurate in absolute terms than those across the internal networks. When processes living on different nodes of the same metahost measure their offset relative to a master process living on another metahost, they might be well-synchronized relative to the master because the accuracy of the offset is sufficient in relation to the message latency of the external network. However, the errors of offsets relative to each other, which are calculated by subtracting their offsets relative to the master, might be unacceptably high in relation to the latency of the internal network between them [4]. More precisely, the error of the offset measurement between two processes at a given moment (calculated or measured) should be smaller than half of the message latency between them to ensure the clock condition. As explained above, this requirement may be violated if the offset between processes connected by a low-latency link is derived from offsets between processes connected by a high-latency link because it is assumed that the absolute error of offset measurements grows with the latency.

As a consequence, we use a hierarchical offset measurement scheme to account for the hierarchy of latencies which is typical for metacomputing environments with different geographically dispersed clusters (i.e., metahosts) connected via wide-area networks. Given that higher latencies make offset measurements across these wide-area networks less accurate than measurements across local-area networks, offset values within and across metahosts are determined separately and finally combined to individual offsets between arbitrary processes. More precisely, each metahost first determines a local master process. After that, one metamaster is chosen among all the local masters. Now all local masters measure their offset relative to the metamaster. After this has been done, all worker processes exchange ping-pongs with their local master to determine the offset relative to the local master. If a metahost already provides a metahost-global clock, this second step is omitted. Finally, the offset to the metamaster is calculated by adding the two measured offset values. Compared to measuring

all offsets relative to just one single master clock, the hierarchical scheme has the advantage that all clocks within the same metahost use the same inter-metahost offset measurement and thus their relative offset remains unaffected from distorted wide-area measurements. More details can be found in [4]. Although this hierarchical offset measurement scheme already significantly increases the accuracy of the overall analysis, traces may still exhibit clock condition violations.

To remove residual inconsistencies in event traces taken in metacomputing environments, the implementation of the CLC algorithm itself has been extended such that it uses an inter-machine latency to account for wide-area communications. This introduces a third latency level—in addition to the inter-node and intra-node latency. The next section presents results showing that the CLC algorithm removes remaining inconsistencies in event traces taken in metacomputing environments and so demonstrates that the algorithm also proves beneficial in such environments. Specifically, it shows evaluation results of the CLC algorithm taken on the NGS grid using the SMG2000 benchmark.

4.2 National Grid Service

To evaluate the controlled logical clock algorithm on a metacomputer, measurements were taken on the UK National Grid Service (NGS) grid [41]. The NGS is the core UK academic research grid and is intended for the production use of computational and data grid resources. It provides coherent electronic access for UK researchers to all computational- and data-based resources and facilities required to carry out their research, independent of resource or researcher location. This section describes the network topology, hardware architecture, and middleware of the NGS grid.

Network topology and hardware architecture The network behind the NGS grid uses the Janet backbone, which links various academic research sites in the UK—including the sites at NGS—through a high-bandwidth and low latency wide-area network. The network topology of the Janet backbone is illustrated in Fig. 5. NGS resources are linked to this backbone and comprise the four founding members, which are the Science and Technology Facilities Council's e-Science Centre, the University of Oxford, the White Rose Grid (University of Leeds), and the University of Manchester, plus various partner and affiliate sites.

The metahosts at the University of Leeds and the University of Manchester are located roughly 65 km apart from each other with a measured MPI inter-machine latency of 1.3 ms:

- Located at the University of Leeds is a PC Linux cluster with 48 nodes, each with 2 dual-core AMD Opteron processors running at 2.6 GHz. The compute nodes communicate primarily through a Myrinet network with Myrinet adapters integrated on each node. The measured MPI inter-node latency was 4.4 µs, the measured MPI intra-node latency was 1.5 µs.
- Located at the University of Manchester is a PC Linux cluster with 48 nodes, each with 2 dual-core AMD Opteron processors running at 2.6 GHz. The compute nodes communicate primarily through a Myrinet network with Myrinet adapters integrated on each node. The measured MPI inter-node latency was 4.4 µs, the measured MPI intra-node latency was 1.5 µs.

The nodes of the connected compute clusters are linked to the backbone with 1 Gbps adapters. The high bandwidth of the backbone can only be used if the data transmission between the clusters is done in parallel. Given that the network is not only assigned to the NGS grid but also to other UK academic research facilities, parallel applications on the NGS grid share the available network resources with other applications. Nonetheless, compute clusters can internally use their own Myrinet network.

Middleware The co-scheduling of jobs on different clusters in the NGS grid is managed by the Globus grid middleware and the Highly-Available Resource Co-allocator (HARC) [20, 35]. While Globus provides software services and libraries for resource monitoring and management, HARC creates and manages reservations of single resources and groups of resources. In a typical scenario, Globus is responsible for transparent application startup at each site, whereas HARC creates reservations for cross-site jobs and implements the co-allocation across the wide-area network.

In addition, NGS uses MPIg, an MPICH-based grid-enabled MPI implementation, to establish direct connections to the external network from each node [31]. MPIg allows users to couple multiple machines of potentially different architectures to run message-passing applications. In order to take advantage of usually fast intra-machine networks, MPIg is built on top of several devices supporting different flavors of communication substrates. These substrates include vendor-specific interconnects for fast intra-machine communication as well as distinct TCP/IP interconnects for inter-machine communication across a wide-area network.

4.3 Experimental evaluation

As a test application, the MPI version of the SMG2000 benchmark [9] was used on the NGS grid. 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. Therefore a longer run was emulated by inserting sleep statements immediately before and after the main computational phase so that it was

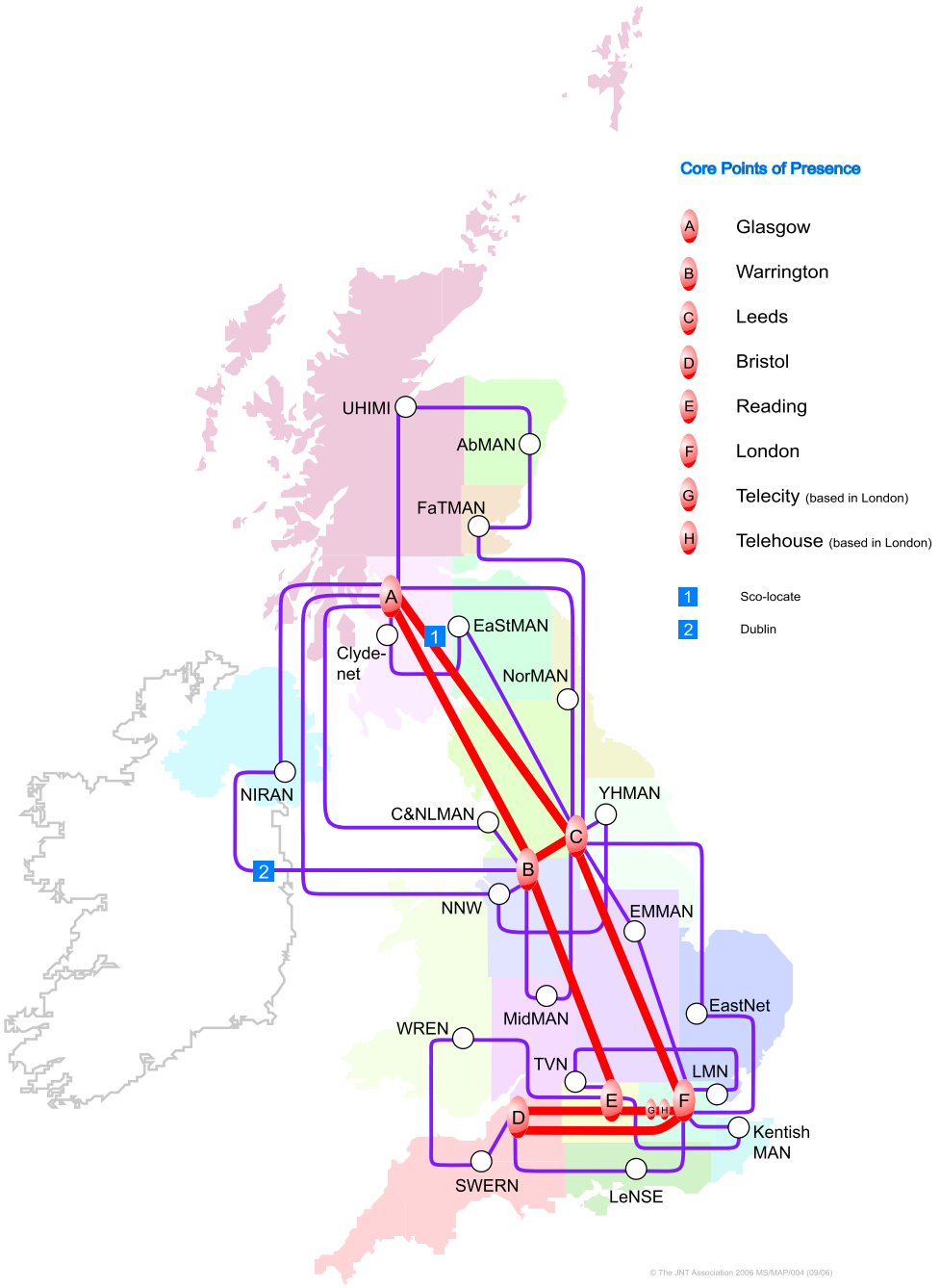


Fig. 5 Network topology of the Janet backbone [29]

carried out ten minutes after initialization and ten minutes before finalization. This corresponds to a scenario, in which only distinct intervals of a longer run are traced with tracing being switched off in between. Since full traces of long running applications may consume a prohibitive amount of storage space, the "partial" tracing emulated here mimics the recommended practice of tracing only pivotal points that warrant a more detailed analysis. For this 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, this method had the advantage that the total runtime including the actual computational activity and therefore the distance between the two offset measurements was roughly independent of the processor configurations. Note that unless stated otherwise, all numbers presented in this section represent the average across at least three measurements.

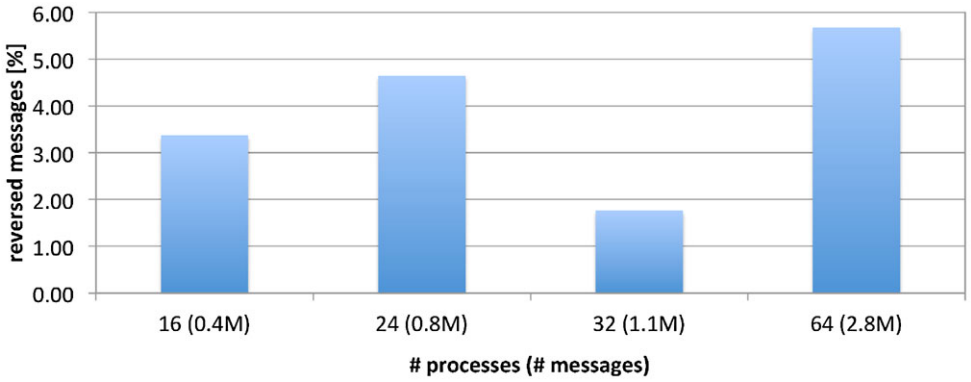


Fig. 6 Percentage of (logical) MPI messages with the order of send and receive events being reverse in the original trace

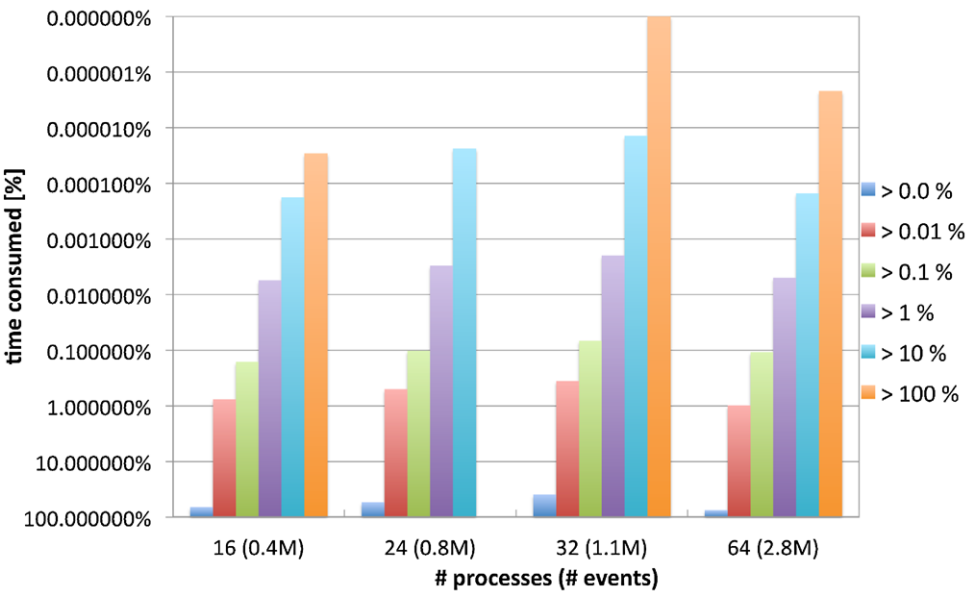


Fig. 7 Relative deviation of the event distance: percentage of SMG2000 execution time consumed by intervals with deviation above threshold on NGS

To provide evidence of the frequency and the extent of clock condition violations, Fig. 6 shows the percentage of messages with the order of send and receive events being reversed in the original trace. All traces contained clock condition violations which shows that further synchronization is important for enabling accurate analyses of event traces taken in metacomputing environments. The maximum absolute displacement error in the original trace was 95.16 µs, whereas the average error among violated events was 6.72 µs across all configurations.

The CLC algorithm eliminates these violated event semantics and thus improves the accuracy of inter-process timings, taking the hierarchy of latencies found on a metacomputer into account. However, the necessary corrections also modify relative process-local event timings. To assess the collateral error inflicted on local timings while applying the CLC algorithm, we determined the relative deviation of local interval lengths, considering two different types of intervals:

– intervals between an event and the first event on the same processor, which is referred to as the *event position*, and
– intervals between adjacent processor-local events (i.e., intervals between an event and its immediate successor), which is referred to as the *event distance*.

To account for the relatively long "correct" stretches artificially introduced by the sleep statements before and after the main computation, only the middle section of the trace between the sleep statements was considered. The maximum relative deviation of the event position across all measurements was 0.37% and the maximum absolute deviation of the event position was 99.57 µs, roughly corresponding to the maximum displacement error observed. Moreover, Fig. 7 shows the relative deviation of the event distance across different numbers of processors. Each bar indicates the percentage of execution time consumed by intervals in a certain error class. It can be seen that larger deviations are still possible in spite of very small averages. The results given in Fig. 7 indicate that larger deviations are rare and that

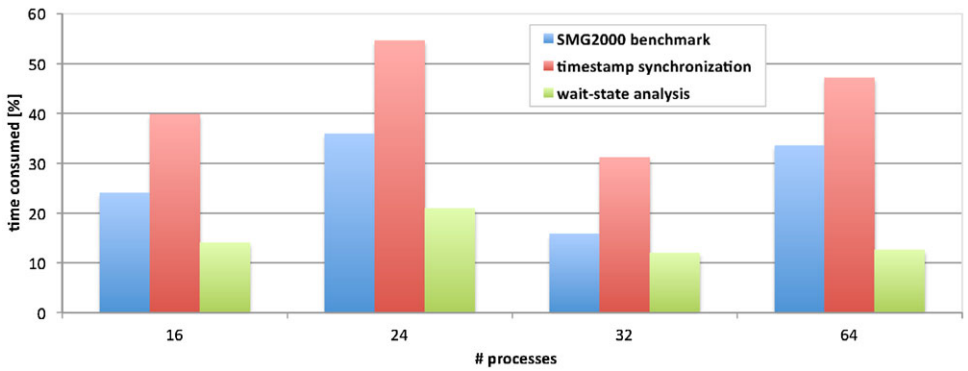


Fig. 8 Scalability of the parallel timestamp synchronization and SMG2000 benchmark on the NGS grid

their influence on performance analysis results will usually be negligible.

The runtime behavior of the parallel CLC algorithm was also evaluated on the NGS grid. Figure 8 presents scaling results of the parallel timestamp synchronization, the Scalasca wait-state analysis, and the SMG2000 benchmark itself. The results demonstrate that the wait-state analysis, the parallel timestamp synchronization, and the execution of the SMG2000 benchmark 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. We can therefore conclude that the CLC algorithm is suitable to run efficiently in a metacomputing environment.

5 OpenMP

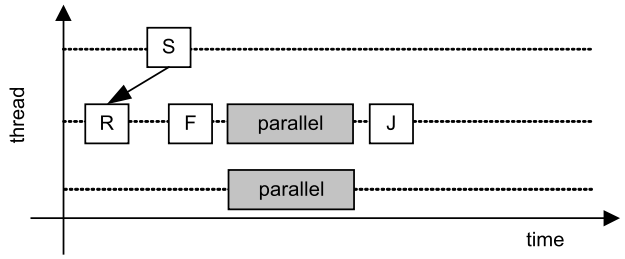
As a common trend that can be observed in response to the proliferation of multicore processors with their rising numbers of cores per chip, the shared-memory nodes most clusters are composed of are becoming much wider. At the same time, the memory-per-core ratio is expected to shrink in the long run. To utilize the available memory more efficiently, many code developers now resort to using OpenMP for node-internal work sharing, while employing MPI for parallelism among different nodes. This has the advantage that (i) the extra memory needed to maintain separate private address spaces (e.g., for ghost cells or communication buffers) is no longer needed, (ii) the effort to copy data between these address spaces can be reduced, and (iii) the number of external MPI links per node can be kept at a minimum to improve scalability.

The use of inherently different programming models in a complimentary manner is usually referred to as *hybridization*. While potentially improving efficiency and scalability, hybridization usually comes at the price of increased programming complexity. To ameliorate unfavorable effects of hybrid parallelization on programmer productivity, developers therefore depend even more on powerful and robust software tools that help them find errors in and tune the performance of their codes. Hence, this section presents an extended version of the CLC algorithm that in addition to message-passing event semantics also preserves and restores shared-memory event semantics (e.g., OpenMP-related event semantics). Along with necessary algorithmic extensions and the applied hybrid parallelization strategy, this section also presents an experimental evaluation of the algorithms accuracy and scalability.

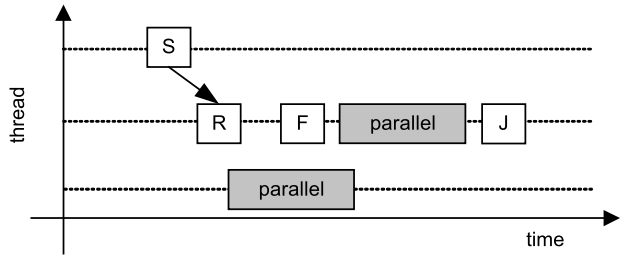
5.1 Algorithmic extensions

So far, the CLC algorithm accounts for happened-before relations in MPI event semantics. It does not yet account for violations of OpenMP event semantics in the original trace, making it unsuitable for OpenMP applications and hybrid applications that use MPI and OpenMP parallelism in combination. The potential implications of isolated corrections based on MPI event semantics for the semantics of OpenMP events are exemplified in Fig. 9 using the time lines of three threads. Shown is a violated message exchange between a send and receive pair followed by the execution of an OpenMP parallel region. Here, the execution of the OpenMP parallel region by two threads is enclosed by a fork (F) and a join (J) event of the master thread. Whereas in Fig. 9(a) the point-to-point event order is violated, the parallel regions appear clearly after the worker has been forked. However, while in Fig. 9(b) the logical point-to-point event order is restored, now one thread enters the parallel region before it has been forked, which is impossible. In other words, the algorithm detects and corrects the clock condition violation in the point-to-point message exchange, while the subsequent forward amortization introduces a new violation as a result of the algorithm not accounting for event semantics in shared-memory operations.

To enforce the correction also of happened-before relations in OpenMP regions (i.e., executed constructs) alongside those implied by the MPI standard, we treat the events involved as logical point-to-point communication events. Thus, a happened-before relation between two events in an OpenMP region is modeled as the exchange of a logical



(a) Inconsistent point-to-point event semantics followed by consistent shared-memory fork semantics: All threads enter the parallel region after the fork has occurred



(b) The correction of inconsistent point-to-point event semantics may lead to inconsistent shared-memory fork semantics: Here, one thread enters the parallel region before it has been forked

Fig. 9 Violations of OpenMP event semantics in the wake of restoring MPI event semantics

message between the two events. Depending on the temporal dependencies among the events characterizing an OpenMP region, an event can be mapped either onto a logical send or onto a logical receive event. Once those mappings are defined, our earlier algorithmic framework [7] can essentially be reused. This is why we do not repeat the formulas here again and, instead, concentrate on the identification of happened-before relations in OpenMP. Table 2 lists all OpenMP regions currently supported by our event model where we can identify happened-before relations and divides them into groups with very similar logical communication patterns, which are depicted in Fig. 10. Although not yet provided by our implementation, we also consider tasking, whose integration into our event model is already in progress. Note that the algorithmic extensions do neither cover thread migration between cores nor shared-memory event semantics imposed by cluster-wide OpenMP implementations (e.g., Intel Cluster OpenMP [24]) because in those cases additional communication may be used, introducing further constraints which are currently ignored by the algorithm.

Team creation and termination: Figure 10(a) shows the time-line visualization of three threads executing a parallel region. The master thread creates a team of threads (at fork event F) whose members subsequently enter the parallel region (at enter events E_i). In such a situation, the master thread sends a logical message to all worker threads. The

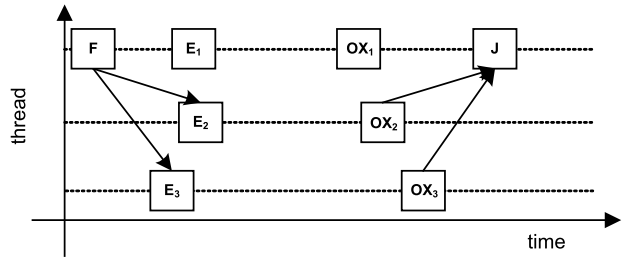
Table 2 Classification of OpenMP regions

Category	OpenMP region
Team creation	begin of `parallel` region
Team termination	end of `parallel` region
Barrier	explicit `barrier` region
	implicit barrier (if executed) at the end of
	`parallel` region
	loop region (i.e., `for`, `do`)
	`single` region
	`workshare` region
	`sections` region
Locking	`omp_set_lock`
	`omp_unset_lock`
	`critical` region
Tasking	`task` region
	`taskwait` region

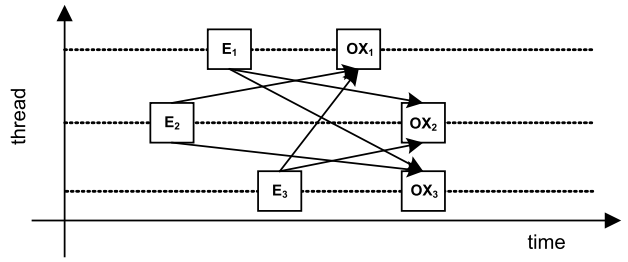
fork event of the master thread is considered a logical send event, whereas all the enter events of the corresponding parallel region, one from each worker in the team, are considered logical receive events. After each thread has left the parallel region (at OpenMP collective exit events OX_i), this team of threads is terminated, as indicated by the join event (J) of the master thread. Here, the master thread receives logical messages from all worker threads. The join event (J) of the master thread is considered a logical receive event. All OpenMP collective exit events (OX_i) of the corresponding parallel region, one from each worker in the team, are considered logical send events.

Barrier: OpenMP barrier constructs are similar to MPI barriers and therefore adhere to the same execution semantics, which require that no thread is allowed to exit a barrier region before the last thread has entered it. Such a situation is illustrated in Fig. 10(b). All threads in the team are at the same time sender and receiver. All enter events (E_i) are considered logical send events and all OpenMP collective exit events (OX_i) are considered logical receive events. This situation shows up in explicit and implicit barrier regions.

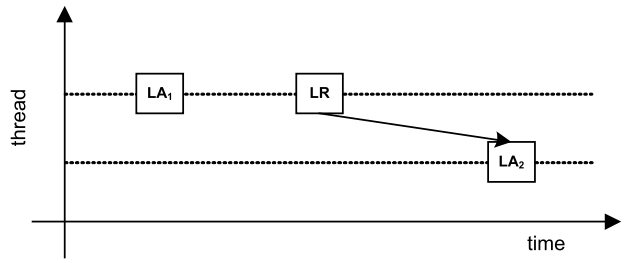
Locking: Figure 10(c) visualizes two threads competing for a lock variable. First, one thread acquires (LA_1) and releases (LR) the lock variable. Then, the other thread locks (LA_2) the same variable after the lock has been released by the first thread. In such a situation, two different happened-before relations exist. First, the thread represented by the upper time line is allowed to release the lock only after it has been acquired ($LA_1 \rightarrow LR$). Since these two events occur on the same time line, this relation is trivially enforced. Second, the thread represented by the lower time line can only acquire the lock once it has been released by



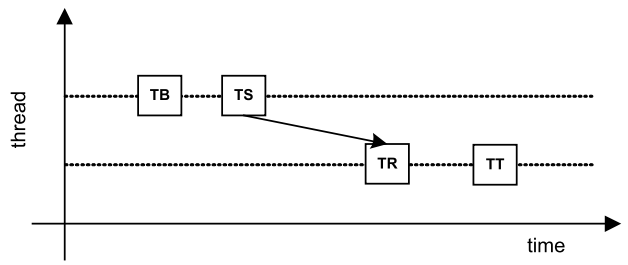
(a) Team creation and termination: The execution of a parallel region is chronologically enclosed by fork and join events



(b) OpenMP barrier: A thread is allowed to exit a barrier region only after the last thread has entered it



(c) OpenMP lock sequence: A thread can acquire a lock only after it has been released – if it was acquired earlier



(d) OpenMP task sequence: A task can only be resumed after it has been suspended

Fig. 10 Happened-before relations in OpenMP regions visualized as arrows representing logical messages

the other thread ($LR \rightarrow LA_2$). Given that a lock variable can be owned only by one thread at a time, the releasing thread sends a logical message to the next thread acquiring the lock. The lock-acquisition event is considered a logical receive event, whereas the lock-release event of the thread releasing the lock is considered a logical send event. Since at program start none of the locks is occupied, the first thread acquiring a given lock does not need to wait for a preceding lock-release event.

As Scalasca models the execution of critical constructs with lock events, the above-mentioned happened-before relations also exist in critical regions. Given that a critical construct restricts the execution of a structured block to a single thread at a time, a lock-acquisition event is recorded before a thread enters the critical region, whereas a lock-release event is recorded after the thread leaves the critical region. Because the same unspecified name or a user-defined name is used to identify a critical region, the event model provides lock identifiers representing the name of a critical region. In addition, similar happened-before relations are also found in atomic and flush constructs, but the source-code instrumentation applied by Scalasca does not allow events inside these regions to be recorded [39], although this would be necessary to determine when a thread enters or leaves such a region. For instance, an atomic construct ensures that a specific storage location is updated atomically. Similar to critical constructs, it would be required to record when a thread executes inside the atomic construct. However, the execution of an atomic construct is restricted to statements that can be calculated atomically, which prevents the insertion of tracing calls. The flush directive, which does not have any code attached to it, is even more restrictive in this regard. Since we cannot record events inside such regions, these constructs are currently ignored by the algorithm.

Given that our current event model does not provide event attributes such as a sequence count indicating the logical order of lock events, this order can only be derived from the timestamps as they are recorded in the trace. Assuming that on most systems the thread-local clocks within a team are synchronized, these timestamps provide a reasonably reliable sequence indicator. However, as on some systems this assumption cannot be maintained, the timestamp alone may be insufficient to determine the correct precedence order of lock events and their violation in the course of MPI-related CLC corrections. Nevertheless, on all systems the algorithm can preserve the event order as found in the original trace. Hence, the original order of lock events is determined prior to the synchronization and subsequently used when the roles of logical senders and receivers are determined.

Tasking: Figure 10(d) shows the time-line visualization of two threads executing an untied task. One thread creates a task at the task-begin event (TB) and subsequently suspends this task at the task-suspend event (TS). Afterward, a thread different from the one that executed the task before it was suspended resumes the task at the task-resume event (TR) and finally terminates the task at the task-termination event (TT). Note that a task may be suspended at any point, not only at implied scheduling points, although some compilers respect scheduling points even for untied tasks. The task-suspend event (TS) is considered a logical send event, whereas the task-resume event (TR) is considered a logical

receive event. Note that this happened-before relation is naturally fulfilled for tied tasks and does not demand any correction. Finally, taskwait regions impose further happened-before relations (not shown) between the exit events of the child tasks created by the surrounding task region and the exit event of an associated taskwait region, which can be handled accordingly.

5.2 Hybrid parallelization

Just like Scalasca's wait-state analysis, the CLC algorithm requires comparing events involved in the same communication operation, which is why it follows a similar parallelization strategy, adopting the general idea of performing a parallel trace replay. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of application processes, we can achieve good scalability on large processor configurations. During the replay, sending and receiving processes exchange the information needed to synchronize the event timestamps. However, different from the wait-state search, which requires only a forward replay, the CLC algorithm performs the replay in both directions—forward and backward. The backward replay, during which the roles of sender and receiver are reversed, is needed because the backward amortization requires knowledge of receiver timestamps on the sender side. To make the parallel CLC implementation applicable to realistic traces from hybrid codes, we

- enabled the replay engine to deal with hybrid traces,
- added logic for a proper identification of logical senders and receivers among OpenMP-related events, and
- facilitated an exchange of timestamps between threads responsible for these events as a prerequisite for the synchronization.

The parallel CLC algorithm is, again like the wait-state analysis, implemented on top of PEARL [21], a parallel library that offers higher-level abstractions to read and analyze large volumes of trace data including random access to individual events, links between related events, functionality to transfer and access remote events, and replay support. In the pure MPI case, the usage model of the library assumes a one-to-one mapping between analysis (i.e., correction) and target-application processes. That is, for every process of the target application, one correction process responsible for the trace data of this application process is created. Data exchange during the replay is accomplished via MPI. The basic idea behind our new hybrid scheme was again to mirror the process and thread structure of the target application and to make the CLC implementation a hybrid program in its own right. To this end, our trace-access library was extended such that the events of every application thread including new OpenMP-specific event types can be accessed and processed by a dedicated analysis thread. Now, the one-to-one correspondence between the executions of the target application and the CLC algorithm exist on two levels: processes and threads. The resulting parallel processing scheme becomes a *hybrid parallel replay* of the target application. Note that the current usage model is restricted in that it supports only MPI calls on the master thread (i.e., MPI funneled mode) and only a fixed number of threads per process.

To synchronize the timestamps, each thread scans the event trace for clock-condition violations and applies forward and backward amortization, as introduced earlier. During the forward amortization, events belonging to OpenMP regions may be classified as logical senders or receivers according to their role in these regions. Events indicating the creation or termination of a team of threads and events indicating the acquisition and release of lock variables are easily classified based on their event type as specified in the trace. For events related to entering or leaving parallel or barrier regions, the logical event type is derived from the region name (e.g., `parallel`, `barrier`) and the role (e.g., master thread) a particular thread plays therein.

Furthermore, we defined functions to exchange and compare timestamps between threads during the different replay phases—mostly representing different flavors of reduction operations. The required communication pattern depends on the type of OpenMP regions whose event timestamps are to be synchronized. As mentioned earlier, in the absence of more precise order attributes the logical event order of lock events is currently derived from their relative timings as recorded in the trace, which may be inaccurate only in those rare cases where the thread-local clocks within a team exhibit significant errors. For this purpose, the chronological event order of lock events is determined in advance and stored in a global data structure before the actual replay is applied. During the replay of lock operations, timestamps are exchanged between the threads competing for the same lock—similar to the replay of point-to-point messages.

5.3 Experimental evaluation

In this section, we evaluate the accuracy and scalability of the parallel controlled logical clock algorithm when applied to traces of hybrid codes and also give evidence of the frequency and the extent of clock condition violations in such traces. We ran our experiments on the Nicole cluster at the Jülich Supercomputing Centre. This cluster consists of 32 compute nodes, each with two quad-core AMD Opteron processors running at 2.4 GHz. The individual compute nodes of the Nicole cluster are linked with an Infiniband network. The measured MPI inter-node latency was 4.5 µs, the measured MPI intra-node latency was 1.5 µs. Unless stated otherwise, all numbers presented in this section again represent the average across at least three measurements.

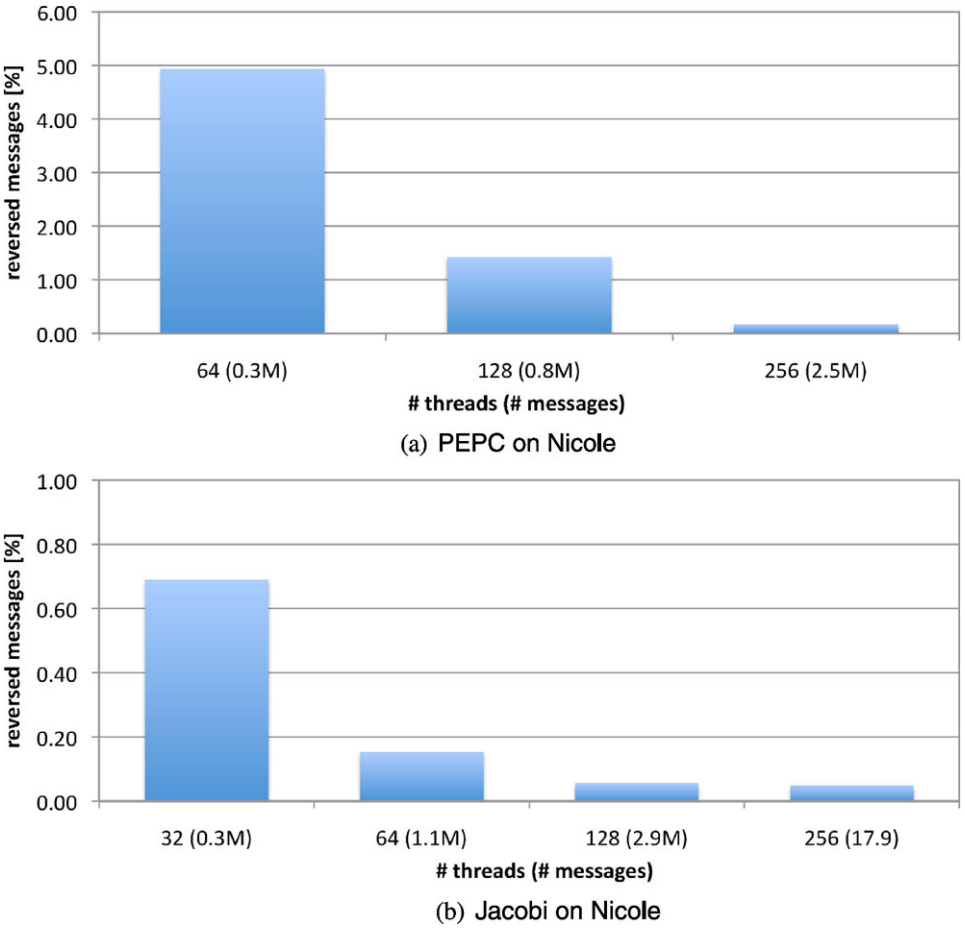


Fig. 11 Percentage of (logical) MPI messages with the order of send and receive events being reversed in the original trace

As a first test case served the application PEPC, a parallel tree-code for rapid computation of long-range Coulomb forces in n-body particle systems [42]. In the course of this evaluation study, the original parallel processing scheme, an MPI implementation of the Barnes-Hut tree algorithm [3] according to the Warren-Salmon hashed oct-tree structure [50], was enriched with shared-memory parallelism within the solver and integrator parts. Applying a strong scaling strategy, a fixed overall number of particles (i.e., 524288) with 100 solver iterations was configured, resulting in an approximately ideal speedup behavior [1]. In our test configurations, the runtime was approximately 30, 15, or 7.5 minutes with 64, 128, or 256 threads. Given that tracing the full run would consume a prohibitively large amount of storage space, selective tracing was applied so that the solver and integrator parts were traced only during iteration 50, which mimics the common practice of tracing only critical intervals worth a more detailed analysis.

A hybrid version of the Jacobi solver, which originally comes along with the OpenMP Source Code Repository of the Parallel Computing Group at the La Laguna University, was used as a second test case [13]. This benchmark solves the Poisson equation on a rectangular grid assuming uniform discretization in each direction and Dirichlet boundary conditions. The original benchmark, a pure OpenMP implementation, had been combined with MPI-based parallelism. Following a strong scaling strategy, a fixed matrix size of 2000×2000 was configured. To emulate a run long enough so that drift deviations may have a noticeable effect, we inserted 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, resulting in a total execution time of roughly twenty minutes.

Table 3 lists the investigated execution configurations along with the distribution of reversed logical messages with respect to the programming model semantics they violate (i.e., MPI or OpenMP). Apparently, in the original trace only violations of MPI event semantics occurred. However, to preserve the logical event order in the corrected trace, OpenMP event semantics were temporarily violated and subsequently restored by the hybrid version of the CLC algorithm. While the pure MPI version that does not account for OpenMP event semantics would leave these violations unnoticed, the hybrid CLC algorithm recognizes such situations and restores the correct order of OpenMP events in the synchronized event trace. After applying the algorithm, the traces were free of any clock-condition violations.

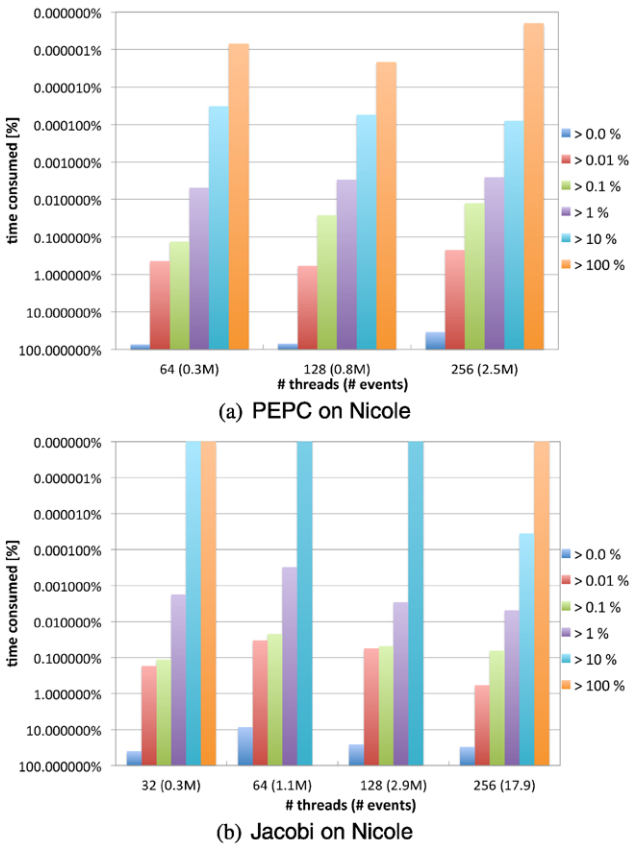


Fig. 12 Relative deviation of the event distance: percentage of execution time consumed by intervals with deviation above threshold

Table 3 Execution configurations and the distribution of reversed messages in the original trace and violated messages detected during the timestamp synchronization with respect to the programming-model semantics they violate

	PEPC			Jacobi			
# CPUs	64	128	256	32	64	128	256
# processes	16	32	64	16	32	64	128
# threads	4	4	4	2	2	2	2
Distribution of reversed messages in the original trace (%)							
MPI	100	100	100	100	100	100	100
OpenMP	0	0	0	0	0	0	0
Violated messages detected during synchronization (%)							
MPI	44	46	42	81	84	81	40
OpenMP	55	53	57	18	15	18	59

Moreover, Fig. 11 shows the frequency of reversed messages as percentage of the total number of messages. Given that none of the OpenMP event semantics was violated in the original trace, the numbers only refer to point-to-point messages and logical messages that can be derived by mapping collective MPI communication onto point-to-point communication. Although the graphs suggest that the number of violations decreases as the number of processors is increased, such a relationship was not generally confirmed in other studies [7]. In the case of PEPC, the drop in the overall runtime might offer an explanation though. The extent of these clock condition violations can be assessed by the average and maximum displacement errors (i.e., the time the receive event appears earlier than the send event) of logical message events in backward order, as seen in the original trace. Among violated events, the PEPC traces exhibit an average error of 21.7 µs and a maximum error of 531.0 µs, whereas the Jacobi exhibit an average error of 3.5 µs and a maximum error of 98.0 µs. The numbers demonstrate that clock-condition violations may appear frequently and that individual violations can be large in absolute terms.

The CLC algorithm eliminates these violated event semantics and thus improves the accuracy of inter-thread timings. However, the necessary corrections also modify relative thread-local event timings and thus the relative deviation of event position and event distance is again determined. For the Jacobi experiments only the middle section of the trace between the sleep statements was considered. The maximum relative deviation of the event position across all PEPC and Jacobi measurements was negligible. The maximum absolute deviation of the event position was 535.78 µs for PEPC and 102.67 µs for Jacobi, roughly corresponding to the respective maximum displacement error observed. Moreover, Fig. 12 shows the relative deviation of the event distance across different numbers of processors for both test applications. Each bar indicates the percentage of execution time consumed by intervals in a certain error class. It can be seen that in spite of very small averages, deviations of occasionally more than 100% are still possible, but the aggregate time consumed by those deviations is very small and their influence on performance analysis results will usually be negligible, as in the cases previously discussed.

On identical configurations, the timestamp synchronization was a factor of 2–3 slower than the equivalent uninstrumented execution of PEPC, which we hope to optimize in future versions of our implementation. To evaluate the scaling behavior of the hybrid synchronization method, Fig. 13 shows a comparison to the Scalasca wait-state analysis and the uninstrumented PEPC solver. The numbers for each configuration are normalized with respect to the execution time of PEPC in the 64 thread configuration. The results demonstrate that the parallel timestamp synchronization, the wait-state analysis, and the execution of PEPC itself exhibit roughly equivalent scaling behavior, which was to be expected due to the replay-based nature of the two trace processing mechanisms.

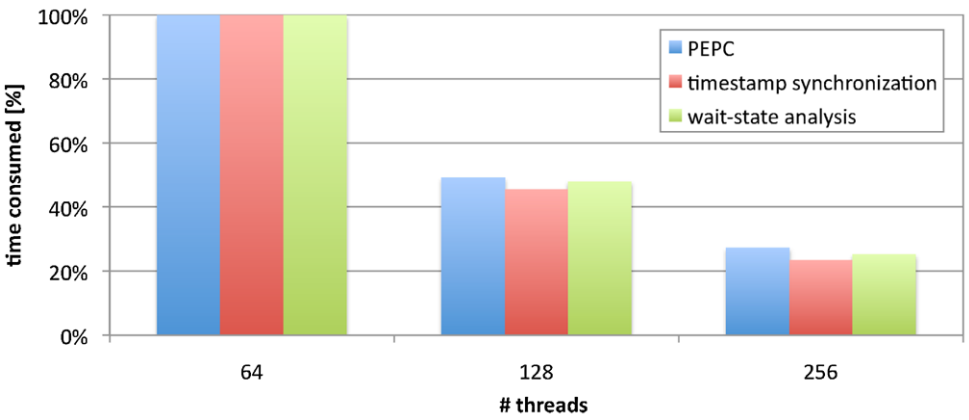


Fig. 13 Normalized execution time of the parallel timestamp synchronization on Nicole

6 Conclusion

Event traces of parallel applications on metacomputers or single clusters may suffer from inaccurate timestamps in the absence of synchronized clocks. As a consequence, the analysis of such traces may yield wrong quantitative and qualitative results, among other effects confusing the users of timeline visualizations with messages flowing backward in time. Because linear offset interpolation can account for such deficiencies only for very short runs, the CLC algorithm retroactively synchronizes timestamps in event traces and restores the correct logical event order. It does so in a scalable manner by replaying the traces in parallel. In this paper, we extended the scope of our algorithm and its implementation within Scalasca in two ways. First, we demonstrated that the algorithm can be employed also in metacomputing environments that allow geographically dispersed clusters to be used as a single coherent system. As a prerequisite for an accurate use of the algorithm, this contribution also encompasses the accurate measurement of clock offsets across a hierarchical network with different latency levels. Second, we extended the CLC algorithm, which was previously designed for pure MPI applications, to also cover hybrid applications that use MPI and OpenMP in combination. The major contribution was the identification of happened-before relations in OpenMP to be taken into account by the algorithm and the hybridization of the parallel replay mechanism. Finally, the hybrid CLC version was integrated into the Scalasca performance-analysis toolset. Our experimental evaluation showed that the good accuracy and scalability characteristics of the pure MPI version were retained in the hybrid version.

In the future, we plan to adapt our algorithm to more advanced OpenMP features such as nested parallelism and tasking as those features are successively integrated into the POMP event model used by Scalasca. Moreover, we want to increase the accuracy of the CLC algorithm further by improving the preceding pre-synchronization via linear clock-offset interpolation, which currently rests on only two offset measurements taken during program initialization and finalization. With low-overhead offset measurements periodically taken during globally synchronizing operations, as introduced by Doleschal et al. [12], the linear interpolation can better account for drift deviations, reducing the number of violations our algorithm needs to correct in the first place and further improving the overall quality of the event timestamp.

References

1. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proc. of the AFIPS Joint Computer Conferences, Atlantic City, NJ, USA, pp. 483–485. ACM Press, New York (1967). doi:10.1145/1465482.1465560
2. Babaoğlu, O., Drummond, R.: (Almost) no cost clock synchronization. Technical Report TR86-791, Cornell University (1986)
3. Barnes, J.E., Hut, P.: A hierarchical O(N log N) force-calculation algorithm. Nature **324**(6096), 446–449 (1986). doi:10.1038/324446a0
4. Becker, D., Wolf, F., Frings, W., Geimer, M., Wylie, B.J.N., Mohr, B.: Automatic trace-based performance analysis of metacomputing applications. In: Proc. of the International Parallel and Distributed Processing Symposium, Long Beach, CA, USA. IEEE Press, New York (2007)
5. Becker, D., Frings, W., Wolf, F.: Performance evaluation and optimization of parallel grid computing applications. In: Proc. of the 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Toulouse, France, pp. 193–199. IEEE Press, New York (2008)
6. Becker, D., Rabenseifner, R., Wolf, F.: Implications of non-constant clock drifts for the timestamps of concurrent events. In: Proc. of the IEEE Cluster Conference, Tsukuba, Japan, pp. 59–68. IEEE Press, New York (2008)
7. Becker, D., Rabenseifner, R., Wolf, F., Linford, J.C.: Scalable timestamp synchronization for event traces of message-passing applications. Parallel Comput. **35**(12), 595–607 (2009)
8. Biberstein, M., Harel, Y., Heilper, A.: Clock synchronization in Cell BE traces. In: Proc. of the 14th Euro-Par Conference, Las Palmas de Gran Canaria, Spain. LNCS, vol. 5168, pp. 3–12. Springer, Berlin (2008)
9. Brown, P.N., Falgout, R.D., Jones, J.E.: Semicoarsening multigrid on distributed memory machines. SIAM J. Sci. Comput. **21**(5), 1823–1834 (2000)

10. Cell Broadband Engine resource center: (2011). www.ibm.com/developerworks/power/cell
11. Cristian, F.: Probabilistic clock synchronization. Distrib. Comput. **3**(3), 146–158 (1989)
12. Doleschal, J., Knüpfer, A., Müller, M.S., Nagel, W.: Internal timer synchronization for parallel event tracing. In: Proc. of the 15th European PVM/MPI Users' Group Meeting, Dublin, Ireland. LNCS, vol. 5205, pp. 202–209. Springer, Berlin (2008)
13. Dorta, A.J., Rodriguez, C., de Sande, F., Gonzalez-Escribano, A.: The OpenMP source code repository. In: Proc. of the 13th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Lugano, Switzerland, pp. 244–250. IEEE Press, New York (2005)
14. Drummond, R., Babaoğlu, O.: Low-cost clock synchronization. Distrib. Comput. **6**(4), 193–203 (1993)
15. Duda, A., Harrus, G., Haddad, Y., Bernard, G.: Estimating global time in distributed systems. In: Proc. of the 7th International Conference on Distributed Computing Systems, Berlin, Germany, pp. 299–306. IEEE Press, New York (1987)
16. Dunigan, T.H.: Hypercube clock synchronization. ORNL TM-11744 (1994). www.csm.ornl.gov/dunigan/clock.ps
17. Edwards, D., Kearns, P.: DTVS: A distributed trace visualization system. In: Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing, Dallas, TX, USA, pp. 281–288. IEEE Press, New York (1994)
18. Fidge, C.J.: Timestamps in message-passing systems that preserve partial ordering. Aust. Comput. Sci. Commun. **10**(1), 56–66 (1988)
19. Fidge, C.J.: Partial orders for parallel debugging. ACM SIGPLAN Not. **24**(1), 183–194 (1989)
20. Foster, I.T.: Globus toolkit version 4: Software for service-oriented systems. In: Proc. of the International Conference on Network and Parallel Computing, Tokyo, Japan. LNCS, vol. 3779, pp. 2–13. Springer, Berlin (2006)
21. Geimer, M., Wolf, F., Knüpfer, A., Mohr, B., Wylie, B.J.N.: A parallel trace-data interface for scalable performance analysis. In: Proc. of the Workshop on State-of-the-Art in Scientific and Parallel Computing, Umeå, Sweden. LNCS, vol. 4699, pp. 398–408. Springer, Berlin (2006)
22. Geimer, M., Wolf, F., Wylie, B.J.N., Mohr, B.: A scalable tool architecture for diagnosing wait states in massively parallel applications. Parallel Comput. **35**(7), 375–388 (2009)
23. Haban, D., Weigel, W.: Global events and global breakpoints in distributed systems. In: Proc. of the 21st Hawaii International Conference on System Sciences, Kailua-Kona, HI, USA, pp. 166–175. IEEE Press, New York (1988)
24. Hoeflinger, J.P.: Extending OpenMP to clusters (2005). cache-www.intel.com/cd/00/00/28/58/285865_285865.pdf
25. Hofmann, R.: Gemeinsame Zeitskala für lokale Ereignisspuren. In: Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, Aachen, Germany, pp. 333–345. Springer, Berlin (1993)
26. Hofmann, R., Hilgers, U.: Theory and tool for estimating global time in parallel and distributed systems. In: Proc. of the 6th Euromicro Workshop on Parallel and Distributed Processing, Madrid, Spain, pp. 173–179. IEEE Press, New York (1998)
27. Huband, S., McDonald, C.: A preliminary topological debugger for MPI programs. In: Proc. of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid, Brisbane, Australia, pp. 422–429. IEEE Press, New York (2001)
28. Jafri, H.: Measuring causal propagation of overhead of inefficiencies in parallel applications. In: Proc. of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, MA, pp. 237–243 (2007)
29. Janet: UK's Education and Research Network: (2011). www.ja.net
30. Jézéquel, J.M.: Building a global time on parallel machines. In: Proc. of the 3rd International Workshop on Distributed Algorithms, Nice, France. LNCS, vol. 392, pp. 136–147. Springer, Berlin (1989)
31. Karonis, N., Toonen, B., Foster, I.: MPICH-G2: a grid-enabled implementation of the message passing interface. J. Parallel Distrib. Comput. **63**(5), 551–563 (2003)
32. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: DiP: a parallel program development environment. In: Proc. of the European Conference on Parallel Computing, Lyon, France. LNCS, vol. 1124, pp. 665–674. Springer, Berlin (1996)
33. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
34. Lorenz, D., Mohr, B., Rössel, C., Schmidl, D., Wolf, F.: How to reconcile event-based performance analysis with tasking in OpenMP. In: Proc. of the 6th International Workshop on OpenMP, Tsukuba, Japan. LNCS, vol. 6132, pp. 109–121. Springer, Berlin (2010)
35. MacLaren, J.: HARC: the highly-available resource co-allocator. In: Proc. of On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, Vilamoura, Portugal. LNCS, vol. 4804, pp. 1385–1402. Springer, Berlin (2007)
36. Maillet, E., Tron, C.: On efficiently implementing global time for performance evaluation on multiprocessor systems. J. Parallel Distrib. Comput. **28**, 84–93 (1995)
37. Mattern, F.: Virtual time and global states of distributed systems. In: Proc. of the International Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France, pp. 215–226. Elsevier Science, Amsterdam (1989)
38. Mills, D.L.: Network Time Protocol (Version 3). The Internet Engineering Task Force—Network Working Group (1992). RFC 1305
39. Mohr, B., Malony, A., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for OpenMP. J. Supercomput. **23**(1), 105–128 (2002)
40. Nagel, W., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: Vampir: visualization and analysis of MPI resources. Supercomputer **12**(1), 69–80 (1996)
41. NGS: National Grid Service: (2011). www.grid-support.ac.uk
42. Pfalzner, S., Gibbon, P.: Many-Body Tree Methods in Physics. Cambridge University Press, Cambridge (1996)
43. Probert, R.L., Yu, H., Saleh, K.: Relative-clock-based specification and test result analysis of distributed systems. In: Proc. of the 11th Annual International Phoenix Conference on Computers and Communications, Scottsdale, AZ, USA, pp. 687–694. IEEE Press, New York (1992)
44. Rabenseifner, R.: The controlled logical clock—a global time for trace based software monitoring of parallel applications in workstation clusters. In: Proc. of the 5th Euromicro Workshop on Parallel and Distributed Processing, London, UK, pp. 477–484. IEEE Press, New York (1997)
45. Rabenseifner, R.: Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen. Ph.D. thesis, University of Stuttgart, Stuttgart (2000)
46. Rodriguez, G., Badia, R.M., Labarta, J.: Generation of simple analytical models for message passing applications. In: Proc. of the European Conference on Parallel Computing, Pisa, Italy. LNCS, vol. 3149, pp. 183–188. Springer, Berlin (2004)
47. Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: in search of the holy grail. Distrib. Comput. **7**(3), 149–174 (1994)
48. Smarr, L., Catlett, C.E.: Metacomputing. Commun. ACM **35**(6), 44–52 (1992)
49. van Dijk, G.J.V., van der Wal, J.V.D.: Partial ordering of synchronization events for distributed debugging in tightly-coupled multiprocessor systems. In: Proc. of the 2nd European Conference on

Distributed Memory Computing, Munich, Germany. LNCS, vol. 487, pp. 100–109. Springer, Berlin (1991)
50. Warren, M.S., Salmon, J.K.: A parallel hashed oct-tree n-body algorithm. In: Proc. of the Conference on High Performance Networking and Computing, Portland, OR, USA, pp. 12–21. ACM Press, New York (1993). doi:10.1145/169627.169640
51. Wong, A.K.L., Goscinski, A.M.: Using an enterprise grid for execution of MPI parallel applications—a case study. In: Proc. of the 13th European PVM/MPI Users' Group Meeting, Bonn, Germany. LNCS, vol. 4192. Springer, Berlin (2006)
52. Yang, Z., Marsland, T.A.: Annotated bibliography on global states and times in distributed systems. Oper. Syst. Rev. **27**(3), 55–74 (1993)

Daniel Becker received his Ph.D. degree from RWTH Aachen University in 2009. He completed his Ph.D. project at the Jülich Supercomputing Centre in the area of scalable performance analysis tools. His career path also includes research stays at academic and industrial organizations including Porsche (Germany), Nokia (Germany), the University of Tennessee, and the IBM T.J. Watson Research Center. Today, he is a postdoctoral researcher at the German Research School for Simulation Sciences, where he works on performance tools and middleware for heterogeneous parallel systems.

Markus Geimer After earning his Ph.D. degree in computer science from the University of Koblenz-Landau (Germany) in 2005, Markus Geimer joined the Jülich Supercomputing Centre as a research scientist beginning of 2006. Since then he is working in the Helmholtz-University Young Investigators Group "Performance Analysis of Parallel Programs" on research in the context of the Scalasca performance-analysis toolset. Moreover, Geimer is the lead developer of Scalasca's parallel trace analysis component. He has published more than a dozen refereed articles in journals and conference or workshop proceedings.

Rolf Rabenseifner studied mathematics and physics at the University of Stuttgart. Since 1984, he has worked at the High-Performance Computing-Center Stuttgart (HLRS). In his dissertation, he developed a controlled logical clock as global time for trace-based profiling of parallel and distributed applications. Since 1996, he has been a member of the MPI-2 Forum and since Dec. 2007, he is in the steering committee of the MPI-3 Forum. From January to April 1999, he was an invited researcher at the Center for High-Performance Computing at Dresden University of Technology. Currently, he is head of Parallel Computing—Training and Application Services at HLRS. He is involved in the HPC Challenge Benchmark Suite. In recent projects, he studied parallel programming models for clusters of SMP nodes, and optimization of MPI collective routines. He teaches MPI, OpenMP, PGAS languages, and hybrid programming in courses, tutorials, and seasonal schools.

Felix Wolf After receiving his Ph.D. degree from RWTH Aachen University in 2003, Felix Wolf spent more than two years as a postdoctoral researcher at the University of Tennessee. Since 2009, he is head of the Laboratory for Parallel Programming at the German Research School for Simulation Sciences in Aachen. At the same time, he holds an appointment as a computer-science professor at RWTH Aachen University, where he teaches parallel programming in science and engineering. Wolf specializes in software and tools for large-scale parallel computers. He is a principal designer of the Scalasca performance-analysis tool. Moreover, Wolf is founder and spokesman of the Virtual Institute—High Productivity Supercomputing, an international initiative of academic HPC programming-tool builders aimed at the enhancement, integration, and deployment of their products.