Synchronizing the timestamps of concurrent events in traces of hybrid MPI/OpenMP applications

Daniel Becker [a1,b,c], Markus Geimer [c2], Rolf Rabenseifner [d3], and Felix Wolf [a4,b,c]

[a]*German Research School for Simulation Sciences, Laboratory for Parallel Programming, 52062 Aachen, Germany*
{[1]d.becker, [4]f.wolf}@grs-sim.de

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

[c]*Forschungszentrum Jülich, Jülich Supercomputing Centre, 52428 Jülich, Germany*
[2]m.geimer@fz-juelich.de

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

***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 that eliminates inconsistent inter-process timings postmortem in traces of pure MPI applications. Since hybrid programming, the combination of MPI and OpenMP in a single application, is becoming more popular on clusters in response to rising numbers of cores per chip and widening shared-memory nodes, we present an extended version of the algorithm that in addition to message-passing event semantics also preserves and restores shared-memory event semantics.**

I. Introduction

Due to the availability of inexpensive commodity components produced in large quantities, clusters now represent the majority of parallel computing systems, exhibiting a vast diversity in terms of architectures, interconnect technologies, and software environments. 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 such 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.

One technique widely used by cluster tools is event tracing with a broad spectrum of applications ranging from performance analysis [1], performance prediction [2] and modeling [3] to debugging [4]. 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, time-line browsers such as Vampir [1] allow these patterns to be visually explored, while the Scalasca toolset [5] 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. Usually, events are recorded along with the time of their occurrence to measure the temporal distance between them and/or to establish a total event order. Obviously, measuring the time between concurrent events necessitates either a global clock or well-synchronized processor-local clocks. While some custom-built clusters such as IBM Blue Gene offer sufficiently accurate global clocks, most commodity clusters provide only processor-local clocks that are either entirely non-synchronized or synchronized only within disjoint partitions (e.g., SMP node). Moreover, external software synchronization via NTP [6] is usually not accurate enough for the purpose of event tracing. 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 in reality 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 [7]. This may 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.

978-0-7695-4220-1/10 $26.00 © 2010 IEEE
DOI 10.1109/CLUSTER.2010.13

In our earlier work [8], we have introduced a scalable algorithm for synchronizing timestamps 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) [9], 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. However, the algorithm is unsuitable for hybrid applications because it neither restores nor preserves happened-before relations in shared-memory event semantics. Most important, the restoration of MPI event semantics may introduce violations of OpenMP event semantics even though those were not violated in the original trace.

To remove these limitations, we describe in this paper how the controlled logical clock is extended to correctly handle traces from hybrid applications. Our work makes the following specific contributions:

- Identification of happened-before relations in OpenMP constructs and library calls as well as their integration into the existing algorithmic framework
- Extension of the parallel replay mechanism so that it can replay traces from hybrid codes
- Integration of the enhanced algorithm and replay engine into the Scalasca performance analysis software

The remainder of this article is organized as follows: After reviewing related work in Section II, we introduce the pure MPI version of the CLC algorithm and describe its limitations in Section III. In Section IV, we present the extensions necessary to correctly synchronize the timestamps that occur during the execution of OpenMP constructs. Then, we describe the hybrid parallelization of the extended algorithm and its implementation within Scalasca in Section V. In Section VI, we evaluate the new scheme with respect to its accuracy, showing that the collaterally introduced deviations of local interval lengths remain within acceptable limits, and its scalability. Finally in Section VII, we summarize our results and outline future work.

II. 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 [6] 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 [10]. 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 [11] or at initialization and finalization [12], 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 [13]. While linear offset interpolation might prove satisfactory for short runs (or interpolation intervals), measurement errors and time-dependent drifts may create inaccuracies and violated happened-before relations during longer runs [7]. Additionally, repeated drift adjustments caused by NTP may impede linear interpolation, as they deliberately introduce non-constant drifts.

If linear interpolation alone turns out to be inadequate to achieve the desired level of accuracy, error estimation allows the retroactive correction of clock values in event traces after assessing synchronization errors among all distributed clock pairs. First, difference functions among clock values are calculated from the differences between clock values of receive events and clock values of send events (plus the minimum message latency). Second, a medial smoothing function can be found and used to correct local clock values because for each clock pair two difference functions exist. Regression analysis and convex hull algorithms have been proposed by Duda [14] to determine the smoothing function. Using a minimal spanning tree algorithm, Jézéquel [15] adopted Duda's algorithm for arbitrary processor topologies. In addition, Hofmann [16] improved Duda's algorithm using a simple minimum/maximum strategy and further proposed that the execution time should be divided into several intervals to compensate for different clock drifts in long running applications. Using a graph-theory algorithm to calculate the shortest paths, Hofmann and Hilgers [17] simplified Jézéquel's algorithm for handling multiprocessor topologies. Biberstein et al. [18] rewrote Hofmann's and Hilgers's algorithm for use on the Cell BE architecture using a short and intelligible notation. Their version solves the clock-condition problem only for short intervals (i.e., without splitting into sub-intervals for handling nonlinear drifts of the physical clocks). Babaoğlu and Drummond [19], [20] 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.

In contrast, logical synchronization uses happened-before relations among send and receive pairs to synchronize distributed clocks. Lamport introduced a discrete logical clock [21] with each clock being represented by a monotonically increasing software counter. As local clocks are incremented after every local event and the updated values are exchanged at synchronization points, happened-before relations can be exploited to further validate and synchronize distributed clocks. If a receive event appears before its corresponding send event, that is, if a clock condition violation occurs, the receive event is shifted forward in time according to the clock value exchanged. As an enhancement of Lamport's discrete logical clock, Fidge [22], [23] and Mattern [24] proposed a vector clock. In their scheme, each processor maintains a vector representing all processor-local clocks. While the local clock is advanced after each local event as before, the vector is updated after receiving a message using an element-wise maximum operation between the local vector and the remote vector that has been sent along with the message.

Finally, Rabenseifner's controlled logical clock (CLC) algorithm [9], [25] , recently extended and parallelized by Becker et al. [8], retroactively corrects clock condition violations in event traces of message-passing applications by shifting message events in time while trying to preserve the length of intervals between local events. The algorithm restores the clock condition using happened-before relations derived from both point-to-point and collective MPI event semantics. Starting from the parallel MPI version of the algorithm, this paper describes its hybridization, retaining its good accuracy and scalability characteristics.

III. Controlled Logical Clock

Because we focus on a timestamp synchronization method to be used within the Scalasca trace-analysis framework, we first describe the Scalasca event model 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 [26], 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 [27]. Event sequences recorded for typical MPI and OpenMP operations are given in Table I. In preparation of its hybridization, 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.

TABLE I
Event sequences recorded for typical MPI and OpenMP operations.

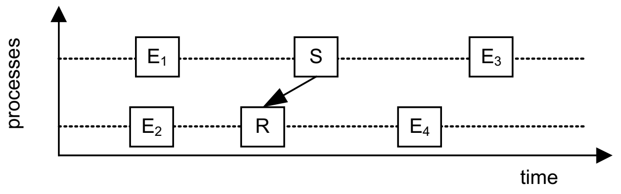
Operation name	**Event sequence**
MPI	
`MPI_Send()`	(enter, send, exit)
`MPI_Recv()`	(enter, receive, exit)
`MPI_Allreduce()`	(enter, MPI collective exit) for each participating process
OpenMP	
`parallel` construct	(fork, enter, OpenMP collective exit, join) for the participating master thread
	(enter, 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)

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 [7], 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* [21]:

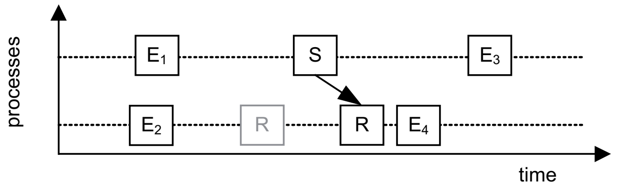
$$\forall\ e, e' : e \rightarrow e' \Longrightarrow C(e) < C(e'). \qquad (1)$$

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 violated 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.

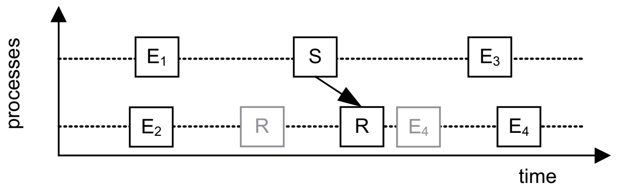
Figure 1 illustrates the different steps of the CLC algo-



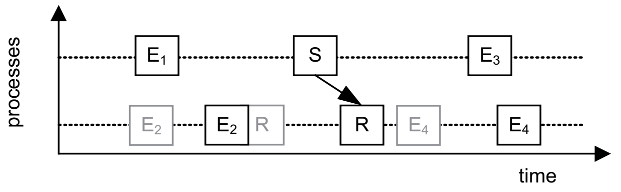
(a) Inconsistent event trace: Clock condition violation in point-to-point communication pair.



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



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



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

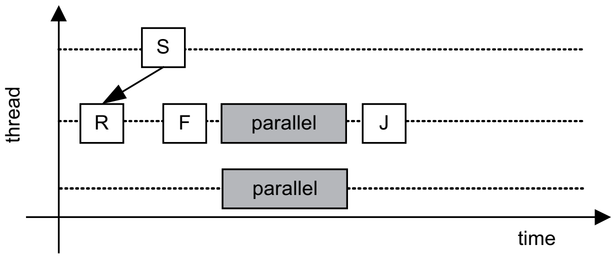
Fig. 1. Backward and forward amortization in the controlled logical clock algorithm.

rithm 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 1(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 (Figure 1(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 (Figure 1(c)). However, the jump discontinuity introduced by adjusting R affects not only events later than R but also events earlier than R. This is corrected during the backward amortization, which shifts E_2 closer to the new position of R (Figure 1(d)). As can be seen in this example, the algorithm only moves events forward in time.

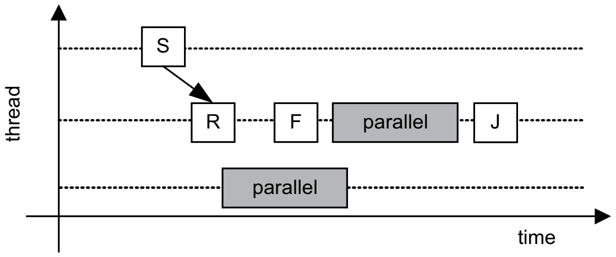
Moreover, happened-before relations also exist among the constituent events of collective MPI operations. The algorithm considers a single collective message-passing operations as being composed of multiple point-to-point operations, taking the semantics of the different flavors of such operations into 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.

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 two factors. First, it does not account for direct violations of shared-memory event semantics in the original trace. Although rare, instances of such violations have been reported [7]. Second and more important, the algorithm does not preserve happened-before relations in shared-memory operations, because the constituent events of such constructs are currently treated as internal 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 this reason, the current CLC algorithm is not suitable for hybrid cluster applications that use MPI and OpenMP in combination.

The potential implications of isolated corrections based on MPI event semantics for the semantics of OpenMP events are exemplified in Figure 2 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 Figure 2(a) the point-to-point event order is violated, the parallel regions appear clearly after the worker has been forked. However, while in Figure 2(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



(a) Inconsistent point-to-point event semantics followed by consistent shared-memory fork semantics: All threads enter the parallel region after they have been forked.



(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. 2. Violations of OpenMP event semantics in the wake of restoring MPI event semantics.

introduces a new violation as a result of the algorithm not accounting for event semantics in shared-memory operations.

To address these limitations, Section IV 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, Section V shows how the hybrid version is parallelized and integrated into the scalable Scalasca trace-analysis framework.

IV. Extensions for OpenMP

Like in the case of MPI, happened-before relations exist among the constituent events of OpenMP regions (i.e., executed constructs). To enforce also these relations 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 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 [8] 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 II 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 Figure 3. 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 [28]) because in those cases additional communication may be used, introducing further constraints which are currently ignored by the algorithm.

1) Team creation and termination: Figure 3(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 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 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.

2) 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 Figure 3(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.

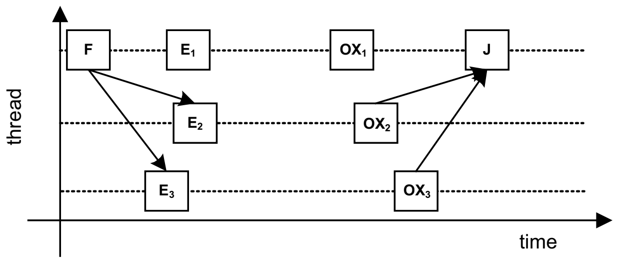
3) Locking: Figure 3(c) visualizes two threads competing for a lock variable. First, one thread acquires (LA_1) and releases (LR) the lock variable. Then, the other thread locks

TABLE II
Classification of OpenMP regions.

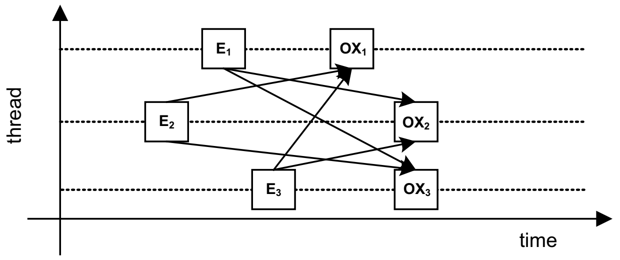
Category	**OpenMP region**
Team creation	begin of `parallel` region
Team termination	end of `parallel` region
Barrier	explicit `barrier` region implicit barrier (if executed) at the end of `parallel` region loop region (i.e., `for`, `do`) `single` region `workshare` region `sections` region

Locking	`omp_set_lock`
	`omp_unset_lock`
	`critical` region

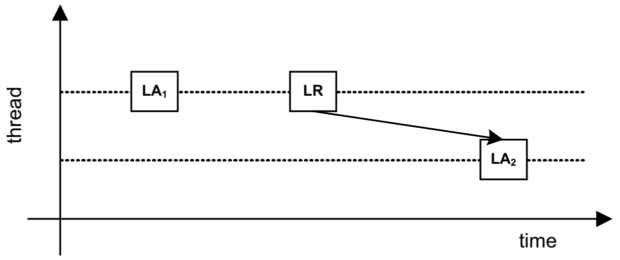
Tasking	`task` region `taskwait` region



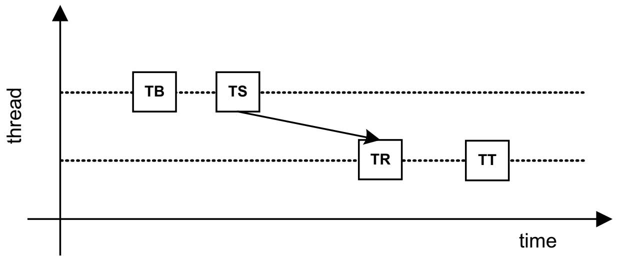
(a) Team creation and termination: The execution of a parallel region is chronologically enclosed by fork and join events.



(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. 3. Happened-before relations in OpenMP regions visualized as arrows representing logical messages.

(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 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 [26], 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.

4) Tasking: Figure 3(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

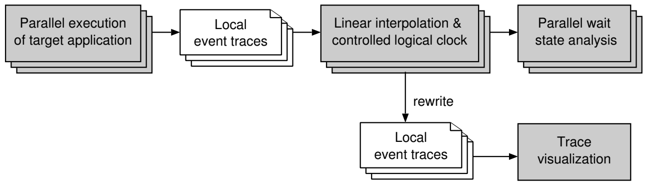

Parallel execution of target application
Local event traces
Linear interpolation & controlled logical clock
Parallel wait state analysis
rewrite
Local event traces
Trace visualization

Fig. 4. 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.

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 the taskwait region, which can be handled accordingly.

V. Hybrid Parallelization

Scalasca, the performance analysis toolset for which our algorithm has been primarily designed, uses event traces to identify wait states that occur, for example, as a result of an unevenly distributed workload and that may harm performance especially at larger scales [5]. The basic analysis workflow is depicted in Figure 4. 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 an third-party time-line browser.

Just like the 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 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 the proper identification of logical senders and receivers among OpenMP-related events, and
- facilitated the exchange of timestamps between the 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 [29], 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.

VI. 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 $4.5\ \mu s$, the measured MPI intra-node latency was $1.5\ \mu s$. Unless stated otherwise, all numbers presented in this section represent the average across at least three measurements.

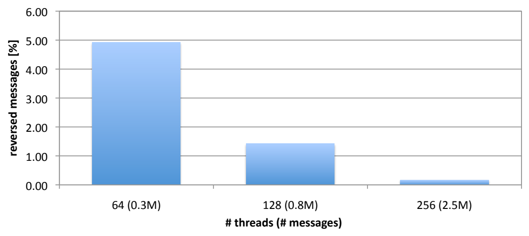
As a first test case served the application PEPC, a parallel tree-code for rapid computation of long-range Coulomb forces in n-body particle systems [30]. In the course of this evaluation study, the original parallel processing scheme, an MPI implementation of the Barnes-Hut tree algorithm [31] according to the Warren-Salmon hashed oct-tree structure [32], 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 [33]. In our test configurations, the runtime was approximately 30, 15 or 7.5 min 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. This mimics the common practice of tracing only pivotal points that warrant 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 [34]. 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.

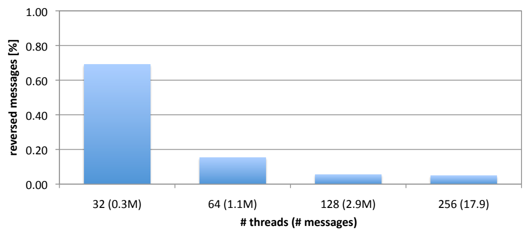
TABLE III
Execution configurations and the distribution of reversed messages in the original trace and violated messages detected during the timestamp synchronization with respect to the programming-model semantics they violate.

	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

Table III 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.

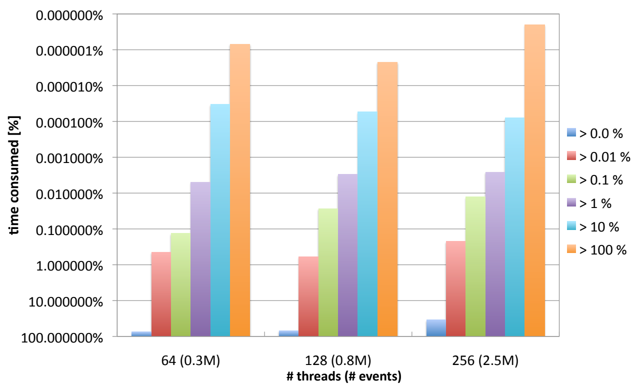


(a) PEPC on Nicole.

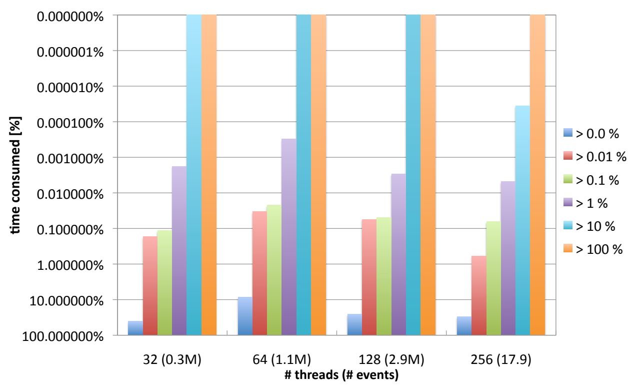


(b) Jacobi on Nicole.

Fig. 5. Percentage of (logical) MPI messages with the order of send and receive events being reversed in the original trace.



(a) PEPC on Nicole.



(b) Jacobi on Nicole.

Fig. 6. Relative deviation of the event distance: Percentage of execution time consumed by intervals with deviation above threshold

Moreover, Figure 5 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 [8]. 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. The PEPC traces exhibit an average error of 21.7μs and a maximum error of 531.0μs, whereas the Jacobi exhibit an average error of 3.5μs and a maximum error of 98.0μs. The numbers demonstrate that clock-condition violations may appear frequently and that individual violations can be large in absolute terms.

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 location, which is referred to as the *event position*, and
- intervals between adjacent thread-local events (i.e., intervals between an event and its immediate successor), which is referred to as the *event distance*.

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, Figure 6 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. All numbers represent the maximum across three measurements. 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.

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, Figure 7 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.

VII. Conclusion

Event traces of parallel applications on 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 time-line visualizations with messages flowing backward in time. Because linear offset interpolation

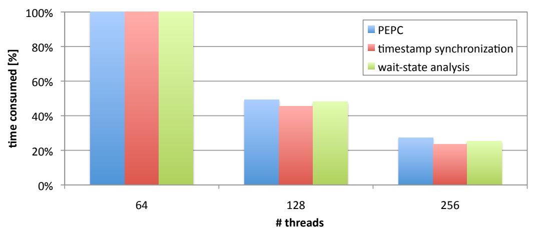


Fig. 7. Normalized execution time of the parallel timestamp synchronization on Nicole.

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 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. [13], 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 timestamps.

REFERENCES

[1] W. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, January 1996.

[2] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, "DiP: A parallel program development environment," in *Proc. of the European Conference on Parallel Computing (Lyon, France)*, ser. LNCS 1124. Springer, August 1996, pp. 665–674.

[3] G. Rodriguez, R. M. Badia, and J. Labarta, "Generation of simple analytical models for message passing applications," in *Proc. of the European Conference on Parallel Computing (Pisa, Italy)*, ser. LNCS 3149. Springer, August - September 2004, pp. 183–188.

[4] S. Huband and C. McDonald, "A Preliminary Topological Debugger for MPI Programs," in *Proc. of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (Brisbane, Australia)*. IEEE, 2001, pp. 422–429.

[5] M. Geimer, F. Wolf, B. J. Wylie, and B. Mohr, "A scalable tool architecture for diagnosing wait states in massively parallel applications," *Parallel Computing*, vol. 35, no. 7, pp. 375–388, Jul. 2009.

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

[7] D. Becker, R. Rabenseifner, and F. Wolf, "Implications of non-constant clock drifts for the timestamps of concurrent events," in *Proc. of the IEEE Cluster Conference (Tsukuba, Japan)*. IEEE, 2008, pp. 59–68.

[8] D. Becker, R. Rabenseifner, F. Wolf, and J. C. Linford, "Scalable timestamp synchronization for event traces of message-passing applications," *Parallel Computing*, vol. 35, no. 12, pp. 595–607, December 2009.

[9] R. Rabenseifner, "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 (London, UK)*. IEEE, January 1997, pp. 477–484.

[10] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, no. 3, pp. 146–158, 1989.

[11] T. H. Dunigan, "Hypercube clock synchronization," ORNL TM-11744, September 1994, www.epm.ornl.gov/~dunigan/clock.ps.

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

[13] J. Doleschal, A. Knüpfer, M. S. Müller, and W. Nagel, "Internal timer synchronization for parallel event tracing," in *Proc. of the 15th European PVM/MPI Users' Group Meeting (Dublin, Ireland)*, ser. LNCS 5205. Springer, September 2008, pp. 202–209.

[14] A. Duda, G. Harrus, Y. Haddad, and G. Bernard, "Estimating global time in distributed systems," in *Proc. of the 7th International Conference on Distributed Computing Systems*. IEEE, September 1987, pp. 299–306.

[15] J.-M. Jézéquel, "Building a global time on parallel machines," in *Proc. of the 3rd International Workshop on Distributed Algorithms (Nice, France)*, ser. LNCS 392. Springer, 1989, pp. 136–147.

[16] R. Hofmann, "Gemeinsame Zeitskala für lokale Ereignisspuren," in *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen (Aachen, Germany)*. Springer, 1993, pp. 333–345.

[17] R. Hofmann and U. Hilgers, "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)*. IEEE, January 1998, pp. 173–179.

[18] M. Biberstein, Y. Harel, and A. Heilper, "Clock synchronization in Cell BE traces," in *Proc. of the 14th Euro-Par Conference (Las Palmas de Gran Canaria, Spain)*, ser. LNCS 5168. Springer, 2008, pp. 3–12.

[19] O. Babaoğlu and R. Drummond, "(Almost) no cost clock synchronization," Cornell University, Technical Report TR86-791, 1986.

[20] R. Drummond and O. Babaoğlu, "Low-cost clock synchronization," *Distributed Computing*, vol. 6, no. 4, pp. 193–203, July 1993.

[21] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[22] C. J. Fidge, "Timestamps in message-passing systems that preserve partial ordering," *Australian Computer Science Communications*, vol. 10, no. 1, pp. 56–66, February 1988.

[23] ——, "Partial orders for parallel debugging," *ACM SIGPLAN Notices*, vol. 24, no. 1, pp. 183–194, January 1989.

[24] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. of the International Workshop on Parallel and Distributed Algorithms (Chateau de Bonas, France)*. Elsevier Science Publishers B. V., Amsterdam, Oct. 1989, pp. 215–226.

[25] R. Rabenseifner, "Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen," Ph.D. dissertation, University of Stuttgart, Stuttgart, March 2000.

[26] B. Mohr, A. Malony, S. Shende, and F. Wolf, "Design and prototype of a performance tool interface for OpenMP," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 105–128, August 2002.

[27] D. Lorenz, B. Mohr, C. Rössel, D. Schmidl, and F. Wolf, "How to reconcile event-based performance analysis with tasking in OpenMP," in *Proc. of the 6th International Workshop on OpenMP (Tsukuba, Japan)*, ser. LNCS 6132. Springer, June 2010, pp. 109–121.

[28] J. P. Hoeflinger, "Extending OpenMP to clusters," Intel Corporation, 2005, cache-www.intel.com/cd/00/00/28/58/285865_285865.pdf.

[29] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, and B. J. N. Wylie, "A parallel trace-data interface for scalable performance analysis," in *Proc. of the Workshop on State-of-the-Art in Scientific and Parallel Computing (Umeå, Sweden)*, ser. LNCS 4699. Springer, June 2006, pp. 398–408.

[30] S. Pfalzner and P. Gibbon, *Many-Body Tree Methods in Physics*. Cambridge University Press, Oct. 1996.

[31] J. E. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, Dec. 1986.

[32] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proc. of the Conference on High Performance Networking and Computing (Portland, OR, USA)*. ACM, 1993, pp. 12–21.

[33] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. of the AFIPS Joint Computer Conferences (Atlantic City, NJ, USA)*. ACM, 1967, pp. 483–485.

[34] A. J. Dorta, C. Rodriguez, F. de Sande, and A. Gonzalez-Escribano, "The OpenMP source code repository," in *Proc. of the 13th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (Lugano, Switzerland)*. IEEE, February 2005, pp. 244–250.