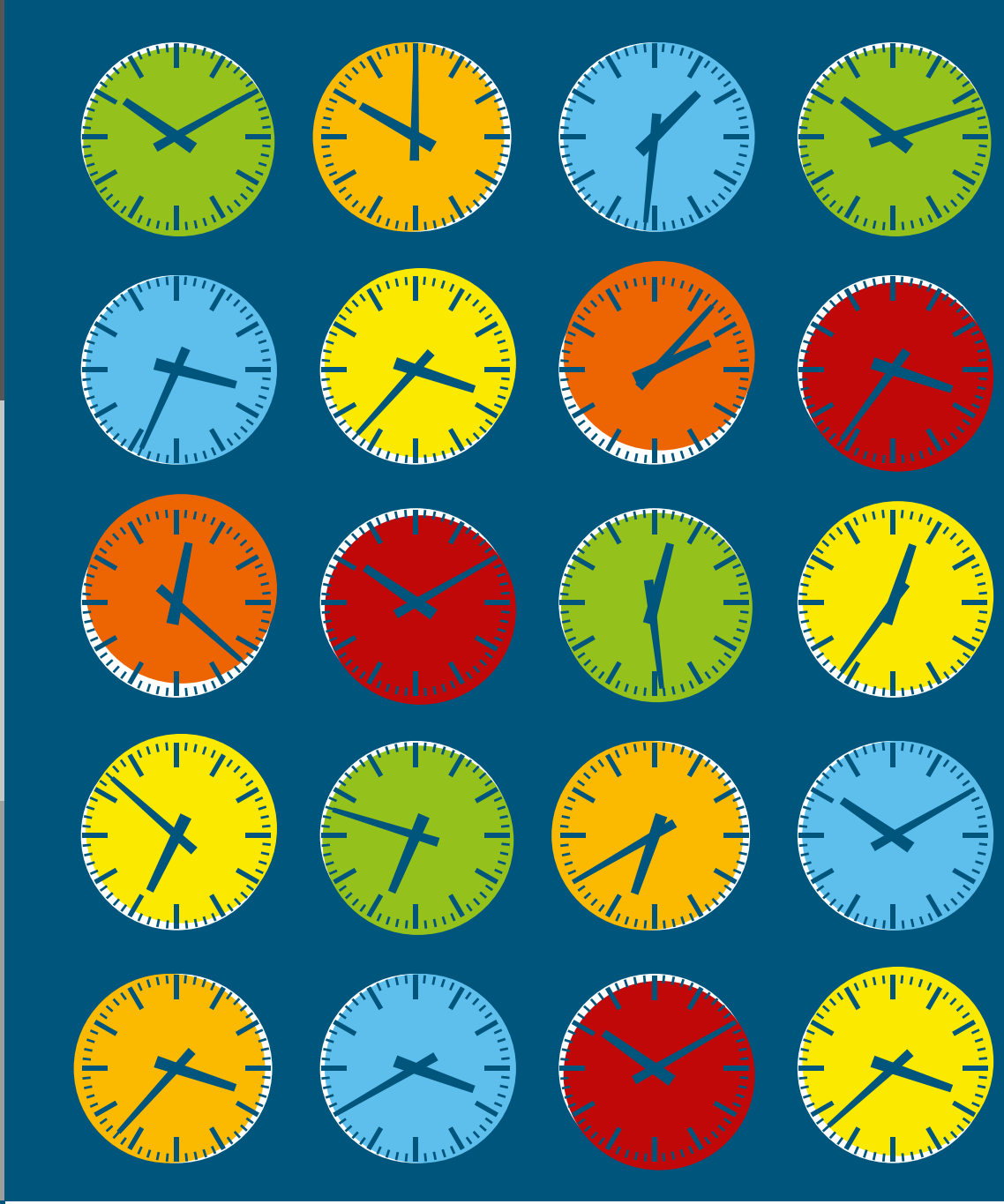


Timestamp Synchronization of Concurrent Events

Daniel Becker

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Schriften des Forschungszentrums Jülich
IAS Series Volume 4

Forschungszentrum Jülich GmbH
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Timestamp Synchronization of Concurrent Events

Daniel Becker

Schriften des Forschungszentrums Jülich
IAS Series Volume 4

ISSN 1868-8489 ISBN 978-3-89336-625-5

Bibliographic information published by the Deutsche Nationalbibliothek.
Die Deutsche Nationalbibliothek lists this publication in the
Deutsche Nationalbibliografie; detailed bibliographic data
are available in the Internet at <http://dnb.d-nb.de>

Cover Illustration: nemadesign GbR, Stuttgart

Publisher and Distributor: Forschungszentrum Jülich GmbH
Zentralbibliothek, Verlag
D-52425 Jülich
phone: +49 2461 61-5368 · fax: +49 2461 61-6103
e-mail: zb-publikation@fz-juelich.de
Internet: http://www.fz-juelich.de/zb

Cover Design: Grafische Medien, Forschungszentrum Jülich GmbH

Printer: Grafische Medien, Forschungszentrum Jülich GmbH

Schriften des Forschungszentrums Jülich
IAS Series Volume 4

D 82 (Diss., RWTH Aachen, Univ., 2009)

ISSN 1868-8489
ISBN 978-3-89336-625-5

The complete volume is freely available on the Internet on the Jülicher Open Access Server (JUWEL) at http://www.fz-juelich.de/zb/juwel

Persistent Identifier: urn:nbn:de:0001-2010051916
Resolving URL: http://www.persistent-identifier.de/?link=610

To my family.

Abstract

Supercomputing is a key technological pillar of modern science and engineering, indispensable for solving critical problems of high complexity. However, to effectively utilize the enormously complex large-scale computer systems available today, scientists and engineers need powerful and robust software development tools. One technique widely used by such tools is event tracing with a broad spectrum of applications ranging from performance analysis, performance prediction and modeling to debugging. In particular, event traces are helpful in understanding the performance behavior of parallel programs since they allow the in-depth analysis of communication and synchronization patterns. The accuracy of such analyses depends on the comparability of timestamps taken on different processors and may be adversely affected by non-synchronized clocks leading to inaccurate relative event timings. Such inaccuracies may cause a given interval to appear shorter or longer than it actually was, or introduce violations of the logical event order, which requires a message to be received only after it has been sent. Inconsistent trace data may not only lead to false conclusions, for instance, when the impact of communication patterns is quantified, but may also confuse the user of trace-visualization tools by causing message arrows to point backward in time-line views. Even more strikingly, trace-analysis tools may also cease to work in a satisfactory manner if they rely on the correct order to function properly. Although linear offset interpolation can restore the consistency of the trace data to some degree, time-dependent drifts and other inaccuracies may still disarrange the original sequence of events, as shown in a study conducted as a part of this Ph.D. thesis.

The already familiar controlled logical clock algorithm accounts for such violations in point-to-point communication by shifting message events in time as much as needed while trying to preserve the length of local intervals. This algorithm is, however, not suitable for realistic applications because (i) it ignores collective and shared-memory operations and (ii) as a serial algorithm it offers only limited scalability. This thesis addresses these shortcomings by extending the algorithm to restore event semantics related to collective and shared-memory operations and by parallelizing the extended version to make it suitable for large-scale systems including computational grids. The basic idea behind the semantic extension is to consider collective and shared-memory operations as being composed of multiple point-to-point messages, taking the semantics of the different flavors of these operations into account. In order to accomplish the correction in a scalable way, both distributed memory and parallel processing capabilities are exploited by processing separate local trace files in parallel and replaying the original communication on as many CPUs as were used to execute the target application itself. To employ the replay mechanism in computational grids, this work also defines the necessary infrastructure to accurately measure clock offsets in distributed environments with hierarchical networks.

The methodology was evaluated in practice by integrating the extended and parallelized algorithm into the Scalasca trace-analysis framework and applied to traces of realistic applications taken on single cluster systems and computational grids. The thesis shows that the algorithm eliminates inconsistent timings of concurrent events while only marginally changing the length of intervals between local events – even if wide-area communication is involved. Scalability is demonstrated with up to 4,096 application processes.

Kurzfassung

Supercomputing ist eine Schlüsseltechnologie moderner Wissenschaft und Technik, die zur Beantwortung schwieriger und komplexer Fragen unersetzlich ist. Zur effizienten Nutzung der neuesten Supercomputersysteme benötigen Wissenschaftler und Ingenieure mächtige und robuste Softwarewerkzeuge. Dabei ist das Aufzeichnen von Laufzeitereignissen in Ereignisspuren eine etablierte Technik sowohl zur Leistungsanalyse, Leistungsvorhersage und Modellierung als auch zum Debugging. Grund für die besondere Eignung der Ereignisspuren zur Untersuchung des Leistungsverhaltens paralleler Programme ist ihre Fähigkeit, die Analyse von Kommunikations- und Synchronisationsmustern zu ermöglichen. Die Genauigkeit dieser Untersuchungen hängt dabei von der Vergleichbarkeit der Zeitstempel einzelner nebenläufiger Ereignisse ab und kann daher durch nicht synchrone Prozessoruhren, die ungenaue Zeiten liefern, vermindert werden. Solche Ungenauigkeiten können sowohl die Länge eines gegebenen Intervalls kürzer oder länger erscheinen lassen, als auch zu Verletzungen der logischen Ereignisabfolge in Ereignisspuren führen, welche verlangt, dass eine Nachricht erst empfangen wird, nachdem sie gesendet wurde. Inkonsistente Ereignisspuren können somit nicht nur zu falschen Schlussfolgerungen, zum Beispiel während der Quantifizierung des Einflusses einzelner Kommunikationsmuster führen, sondern auch die Anwender von Visualisierungswerkzeugen verwirren, indem rückwärts gerichtete und somit in die Vergangenheit weisende Nachrichtenpfeile angezeigt werden. Besonders auffällig sind dabei diejenigen Analysewerkzeuge, die ihre Bearbeitung sogar gänzlich abbrechen, da sie die korrekte logische Abfolge von Ereignissen zur exakten Verarbeitung voraussetzen. Obwohl eine lineare Interpolation des Uhrenabstandes die Konsistenz der Ereignisspuren zu einem gewissen Grad wiederherstellen kann, können zeitabhängige Uhrenabweichungen und weitere Ungenauigkeiten die ursprüngliche Abfolge von Ereignissen ändern, wie in einer Studie als Teil dieser Doktorarbeit gezeigt wird.

Der schon bekannte Algorithmus der geregelten logischen Uhr korrigiert solche Verletzungen in Punkt-zu-Punkt Kommunikationen durch eine ausreichende zeitliche Verschiebung von Nachrichtenereignissen, wobei versucht wird, die Intervalllänge zwischen lokalen Ereignissen zu erhalten. Dieser Algorithmus ist jedoch für realistische parallele Programme nicht geeignet, weil er (i) kollektive Operationen sowie Operationen, die einen gemeinsam genutzten Speicherbereich ansprechen, ignoriert und (ii) als serieller Algorithmus nur eingeschränkte Skalierbarkeit aufweist. Diese Arbeit behebt die beschriebenen Limitierungen zum einen durch die Erweiterung des Algorithmus zur Wiederherstellung von Ereignissemantiken in den oben angeführten Operationen und zum anderen durch die Parallelisierung des erweiterten Algorithmus zur Anwendung auf Supercomputern und Grids. Der Semantikerweiterung liegt die Idee zugrunde, die genannten Operationen zusammengesetzt aus einzelnen Punkt-zu-Punkt Nachrichten aufzufassen und hierbei die verschiedenen Arten dieser Operationen zu berücksichtigen. Um die Korrektur skalierbar auszuführen, werden der verteilte Speicher und die parallelen Rechenressourcen zum einen durch eine parallele Verarbeitung einzelner lokaler Ereignisspuren und zum anderen durch das Nachspielen der ursprünglichen Kommunikation der Zielanwendung ausgenutzt. Um das beschriebene Verfahren auch in Grids einzusetzen, definiert die vorliegende Arbeit die notwendige Infrastruktur zur genauen Messung des Uhrenabstandes in verteilten Umgebungen mit hierarchischen Netzwerken.

Die Methodik wird evaluiert anhand der Integration des erweiterten und parallelisierten Algorithmus in die Leistungsanalyseumgebung Scalasca und deren Anwendung auf Ereignisspuren realistischer paralleler Programme, die auf verschiedenen Clustern und Grids aufgezeichnet wurden. Die vorliegende Arbeit zeigt dabei auf, dass der Algorithmus inkonsistente Intervalle beseitigt und zeitgleich die Länge von Intervallen zwischen lokalen Ereignissen nur marginal ändert – selbst wenn Weitverkehrsnetze eingesetzt werden. Zudem wird die Skalierbarkeit mit bis zu 4.096 Prozessoren gezeigt.

Acknowledgments

I would like to thank Prof. Dr. Dr. Thomas Lippert, Director of the Jülich Supercomputing Centre, for giving me the opportunity to carry out my PhD project in his excellent and supportive research environment. This project would have never been possible without the enthusiastic supervision and guidance of my advisor Prof. Dr. Felix Wolf, to whom I owe a deep debt of gratitude. Moreover, I thank Prof. Dr. Christian Bischof and Prof. Dr. Michael Resch for serving as second referees. I am also grateful to Prof. Dr. Jack Dongarra for hosting me at the University of Tennessee, Dr. David Klepacki for creating the opportunity of a visiting research studentship at the IBM T. J. Watson Research Center, Prof. Dr. Jesus Labarta for supporting me at the Barcelona Supercomputing Centre, Dr. Patrick Worley for supporting me at the Oak Ridge National Laboratory, and Dr. Rolf Rabenseifner for introducing me to the controlled logical clock algorithm.

Thanks are due to colleagues at the Jülich Supercomputing Centre for numerous stimulating discussions, help with experimental setup, and general advice. I would especially like to acknowledge the invaluable assistance of Dr. Markus Geimer, when technical help was necessary, and Wolfgang Frings along with Morris Riedel, without whom the Grid would have been much cloudier. Not to be forgotten, thanks are due to my PhD colleagues for their help and support in myriad situations.

Finally, I am grateful to all my friends for being my surrogate family for so many years in various phases of my life and for their continuing moral support. Although unfair to mention just a few names, I would like to specially thank Vanessa along with Marco, Thomas, Tina, and Tom for just being there. Last but by no means least, I shall be forever indebted to my parents and family for their unshakeable faith, absolute support, and unstinting encouragement when all else failed and it was most required.

Daniel Becker
December 2009

Causarum enim cognitio cognitionem eventorum facit.

CICERO, TOPICA 67

Contents

List of Figures

List of Tables

List of Tables

Chapter 1

Introduction

Supercomputing is a key technological pillar of modern science and engineering, indispensable for solving critical problems of high complexity. World-wide efforts to build machines with performance levels in the petaflops range acknowledge that the requirements of many key applications can only be met by the most advanced custom-designed large-scale computer systems. However, as a prerequisite for their productive use, the HPC community needs powerful and robust software tools that make the development of parallel applications both more effective and more efficient. Such tools not only help to improve the scalability characteristics of scientific codes and thus expand their potential, but also allow domain experts to concentrate on the science underneath rather than to spend a major fraction of their time debugging their code and tuning it for a particular machine.

1.1 Parallel Computers

Todays supercomputers are most commonly *parallel computers*, executing one program on multiple processors in parallel and so jointly computing the solution of a large scientific or technical problem. Often, such parallel computers are classified based on the memory architecture and referred to as *shared-memory* or *distributed-memory* architectures [48]. On shared-memory machines, all processors share a common address space, whereas distributed-memory machines exhibit multiple private address spaces. Shared-memory architectures are classified as either symmetric shared-memory or distributed shared-memory multiprocessors. *Symmetric shared-memory multiprocessors* (SMP) have a symmetric relationship to memory and include systems like SUN Sunfire or IBM eserver nodes [48]. This style of architecture is also called uniform memory access (UMA), arising from the fact that all processors have a uniform latency to memory. Architectures that support shared memory in a distributed fashion are called *distributed shared-memory* (DSM) multiprocessors. Such distributed shared-memory architectures have variable access times to a memory address and are called non-uniform memory access (NUMA). In addition, cache coherent non-uniform memory access (ccNUMA) architectures, such as SGI Origin/Altix, are similar to NUMA architectures but use protocols to guarantee cache coherence across the machine.

In contrast, distributed-memory architectures, often referred to as *massively parallel processors* (MPP), do not provide a common address space but provide interconnection networks to exchange data among processors. The data exchange is typically done by sending and

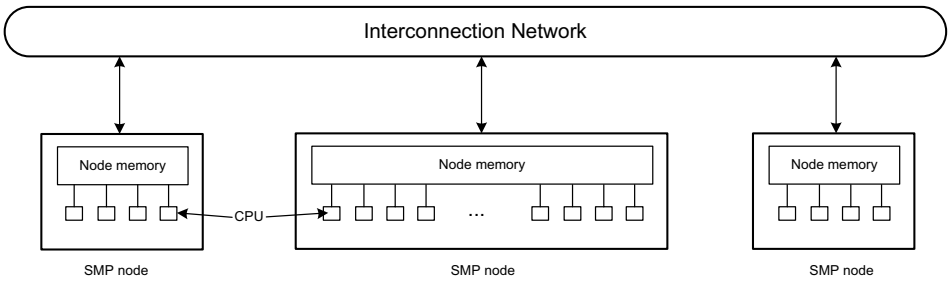


Figure 1.1: Schematic view of a cluster with multiple nodes and its interconnection network.

receiving messages between communication peers, since each network node has its own local memory which is not accessible from another processor. Parallel computers that use this *message-passing* approach are called *clusters*. The individual nodes of a cluster are either commodities or customized, likewise the interconnection network. The nodes and networks of a commodity cluster are usually standard components, whereas the nodes and the interconnect of a custom cluster are customized and more tightly integrated than in a commodity cluster. Typical representatives of custom clusters are IBM Blue Gene and Cray XT systems. Driven by the availability of inexpensive commodity components produced in large quantities, commodity clusters now represent the majority of supercomputing systems, exhibiting a vast diversity in terms of architecture, interconnect technology, and software environment. Commodity clusters include, but are not restricted to, Beowulf-class PC clusters which are composed of commodity hardware, a dedicated interconnection network, and an open-source software stack [30]. Often, these systems are also referred to as Linux clusters or PC clusters. A Beowulf that is built entirely using commodity components is referred to as Class I, whereas Class II clusters may use commodity components along with specialized hardware [87]. In addition, commodity clusters also include other “homemade” clusters [48], for instance, consisting of locally dispersed workstations (e.g., in different offices) linked by a local-area network. Finally, a single cluster node usually consists of many processors sharing a common address space and so clusters may be regarded as coupled SMP systems. Figure 1.1 shows the schematic view of a cluster with multiple nodes and its interconnection network.

Nowadays, clusters represent the majority of supercomputer systems because a low development effort and cheap standard components make their use popular [48]. Given that the processors of a cluster use the interconnection network for exchanging data among processors, the network performance has a major influence on the overall performance of the system. Therefore, different types of networks are used on a cluster in order to increase its performance. On-chip networks are used for interconnecting functional units, caches, and processors within chips or multichip devices. System-area networks are used for interprocessor and processor-memory interconnections within parallel computers. In contrast to custom-built solutions (e.g., IBM Blue Gene and Cray XT), commodity clusters often leverage InfiniBand and Myrinet networks [55, 70]. Finally, local-area networks (LAN) usually connect computer

systems within a single building, whereas wide-area networks (WAN) connect geographically dispersed computers.

Operating systems are used to manage the resources of a parallel computer such as the CPUs, main memory, and network [86]. On shared-memory architectures, one operating system image manages all hardware resources including all the CPUs and the shared memory. In contrast, on distributed memory architectures, each processor typically runs its own OS kernel managing local resources such as the local CPU and memory. In addition, each processor runs further software modules supporting distributed operating system services such as the handling of interprocessor communication. Note that the processor-local operating system may also use globally accessible modules responsible for centralized services (e.g., file-system services or batch system).

1.2 Programming Models

In order to parallelize programs, different parallel programming models are available. Similar to the classification of parallel computers, parallel programming models are classified either as multithreading or as message-passing models [23, 90]. In a multithreading model, one parallel program is concurrently executed by many threads representing execution states that are able to process an instruction stream. As all threads can access the same memory, data exchange among threads is done via shared-memory variables. Synchronization mechanisms, such as locking specific variables or barrier constructs, are used to avoid race conditions. For instance, OpenMP (Open specifications for Multi Processing) [75] is a widespread programming interface realizing a multithreading programming model. It assumes that one master thread creates a team of worker threads once a parallel region has been entered and terminates it after the parallel region has been left. OpenMP provides directives and library calls to coordinate the accesses to shared data, ensuring that certain operations are performed by only one thread at a time.

The programming model on distributed-memory systems is referred to as message passing. This model assumes that programs are executed by one or more processes, each of which has its own private address space. For instance, the MPI (Message Passing Interface) [66] communication library defines a de-facto standard for message passing and is available on most parallel computers. MPI provides means to execute multiple processes in parallel, along with operations for sending and receiving messages, and for performing collective operations across data distributed among different processes. The latest version, MPI 2.1, additionally supports parallel I/O and one-sided communication assuming that a process may interact directly with remote memory across a network to read and write data anywhere on a machine.

Parallel programs can also use message passing and multithreading in combination. Such programs are often referred to as *hybrid* programs. In particular, MPI and OpenMP may be used together in a program, typically on MPPs that consist of multiple SMPs. In such a hybrid programming model, each thread can issue MPI calls. However, MPI may be implemented in environments where threads are not supported or perform poorly, and therefore, it is not required that an MPI implementation fulfills the above requirement. For this reason, MPI

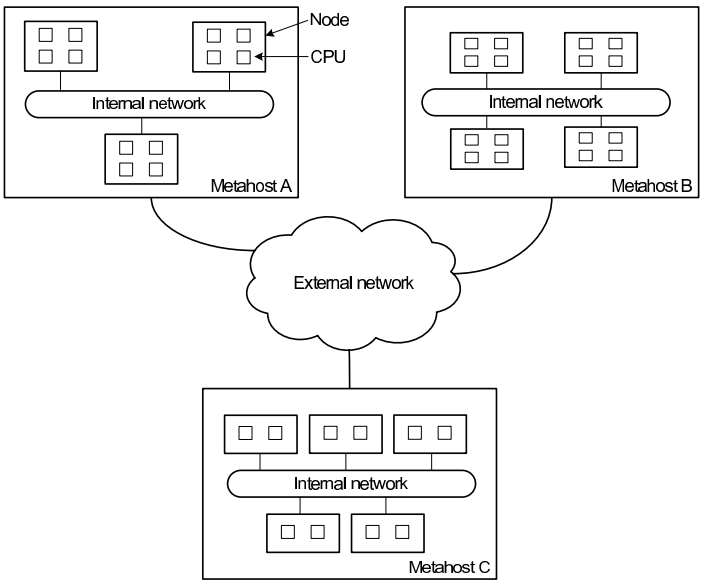


Figure 1.2: Schematic view of a metacomputer including its external and internal networks.

provides software functions to query the respective level of thread support. If only one thread is allowed to be executed at a time, the thread level is *single*. Once multiple threads per process are allowed, the thread level may be *funneled*, *serialized*, or *multiple*. In funneled mode, only the master thread makes MPI calls, whereas in serialized mode all threads make MPI calls but never concurrently. Finally, if multiple threads can make MPI calls without restrictions, the thread level is called multiple.

1.3 Metacomputing Environments

Often, the solution of compute-intensive problems requires more processing power than is available on a single cluster because the problem cannot be solved within a reasonable time frame on a single machine or because the solution must be calculated under real-time conditions. For this reason, the processing power and memory capacity of multiple clusters can be combined to form a more powerful *metacomputer* [84] that appears to its users as a single coherent system. Such a metacomputer usually consists of several independent and potentially heterogeneous and geographically dispersed clusters (metahosts), which are connected by network links to a single unit. Figure 1.2 shows the schematic view of a metacomputer, in which metahosts are internally connected via local area networks, whereas distant metahosts, which often belong to different organizations, are linked by a wide-area interconnection. In this sense, a metacomputing environment can be regarded as a special type

of computational grid. Due to their distributed nature, the predominant programming model for metacomputers is message passing, which may be combined with multithreading used within the nodes of a metahost. Apart from being a pure aggregation of computational power, such a metacomputer can also provide a suitable platform for multi-physics simulations, where the different submodels may be optimized for different architectures.

However, although applications can benefit from the increased parallelism offered by a metacomputer, as supported by a recent study by Wong and Goscinski [95], achieving satisfactory application performance is difficult. Often, applications have to deal with a hierarchy of latencies and bandwidths. In general, the heterogeneity of metahost hardware including differences in networks and architectures complicate load balancing. Given the fact that performance optimization for a single cluster is already a non-trivial task that requires substantial tool support, we can argue that this is even more important for metacomputing environments.

1.4 Event Tracing

Software tools are necessary to investigate the runtime behavior of parallel programs. Given that a computer changes its state in discrete intervals, it is possible to model the runtime behavior of a single program as a sequence of state changes. A single state change can be regarded as an *event* happening at a given time and location. *Event tracing* regards the execution of a program as a sequence of events, each with an associated *event type* (e.g., entering a code region). Such an event type is defined by a set of attributes (e.g., timestamp, location) that may be shared by multiple event types depending on the level of specialization. An *event model* defines the event types with their related attributes and constraints, for example, regarding the correct event order. Obviously, the selection of event types observed determines the expressiveness and granularity an event trace can provide.

Event tracing is a widely used technique by software tools with a broad spectrum of applications ranging from debugging, performance modeling and prediction to performance analysis. For instance, as programming user-defined process topologies is often error-prone, Huband et al. [53] describe a trace-based topology debugger that exploits topological information to abstract, identify, and report patterns of expected and unexpected communication behavior. In addition, performance models can be derived from event traces and subsequently used for performance prediction. Labarta et al. [60] determine such performance models as functions of specific parameters such as the processor count or speed and the network latency or bandwidth. Based on event traces taken on a small number of processors, Rodriguez et al. [82] use these performance models to predict the program performance when running on a large number of processors, enabling the tuning of message-passing programs before actually running them on those large configurations.

Moreover, event traces can be searched for potential performance bottlenecks either manually or automatically [90]. Manual trace analysis transforms event traces into a visual representation of the runtime behavior, which can be interactively explored in graphical trace browsers such as Vampir [71] and Paraver [60]. These trace browsers allow the fine-grained investigation of an application's runtime behavior and provide statistical summaries, translating a

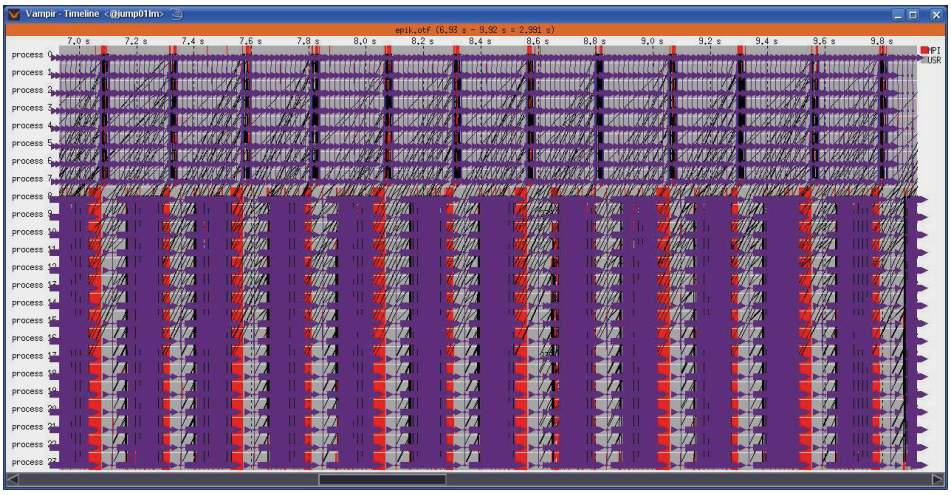


Figure 1.3: Graphical trace browser Vampir: Time-line visualization of an application's runtime behavior.

given event trace into a variety of graphical views including state diagrams, activity charts, and zoomable time-line displays. Figure 1.3 shows a time-line visualization of an application running with 24 processes. Such a time-line visualization consists of boxes indicating the execution of different code sections, arrows indicating point-to-point communication, and dashed lines indicating collective communication. Here, the time spent in MPI calls is visualized through red boxes, whereas the time spent in user regions is visualized through grey boxes. In addition, point-to-point messages are shown with black arrows, whereas collective communication is shown with purple dashed lines. As a consequence, developers can easily identify different execution and communication phases of the application. Using the zooming functionality, they can subsequently investigate the runtime behavior at any level of granularity in or between those phases. However, in view of the large amounts of data generated on contemporary parallel computers, performance bottlenecks can be identified more efficiently by automatically searching the trace data for their occurrence. In addition to usually being faster than a manual analysis performed using a trace browser, this approach is also guaranteed to cover the entire event trace and not to miss any instances.

Automatic trace analysis transforms the event trace into a compact representation of the performance behavior in terms of inefficiency patterns. For instance, the KOJAK [93] and Scalasca [92] toolset automatically search event traces of parallel programs for patterns of inefficient behavior, classify detected instances by category, and quantify the associated performance penalty. This allows developers to study the performance of their applications on a higher level of abstraction, while requiring significantly less time and expertise than a manual analysis. A distinctive feature of both tools is their ability to identify wait states that potentially occur as a result of unevenly distributed workloads. Especially when trying to scale communication intensive applications to large processor counts, such wait states can

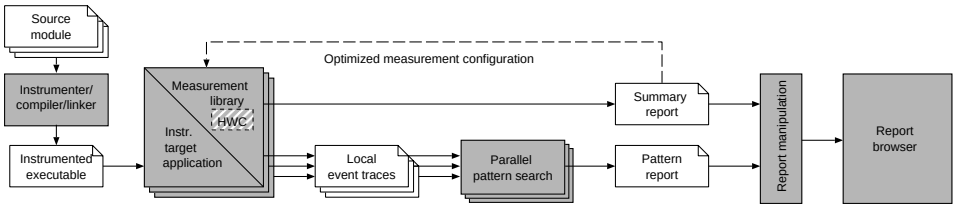


Figure 1.4: Schematic overview of the performance data flow in Scalasca. Grey rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs, files, or data objects running or being processed in parallel.

present severe challenges to achieving good performance. Given that KOJAK analyzes a single global trace file sequentially, its processing scheme is limited as soon as it targets traces taken on such large processor counts. In view of rapidly increasing parallelism, it is crucial that the trace analysis scales to large numbers of application processes [19, 22, 44]. In addition, exponentially rising numbers of cores and increased concurrency levels place even higher scalability demands on this trace analysis [73, 88]. As KOJAK's successor, Scalasca has been specifically designed for use on large-scale systems including IBM Blue Gene and Cray XT, but is also well-suited for small- and medium-scale clusters. Instead of sequentially processing a single global trace file, Scalasca implements a scalable trace-analysis approach [44] by processing separate process-local trace files in parallel and *replaying* the original communication on as many CPUs as were used to execute the target application itself. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of application processes, this approach guarantees good scalability at very large scales.

Scalasca supports an incremental performance analysis process that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. Figure 1.4 shows the basic analysis workflow supported by Scalasca. Before any performance data can be collected, the target application must be instrumented. When running the instrumented code on the parallel machine, the user can choose to generate a summary report ("profile") with aggregate performance metrics for individual function call paths, and/or event traces recording individual runtime events from which a profile can later be produced. Summarization is particularly useful to obtain an overview of the performance behavior and for local metrics such as those derived from hardware counters [18]. Since traces tend to rapidly become very large, scoring of a summary report is usually recommended, as this allows instrumentation and measurement to be optimized. When tracing is enabled, each process generates a trace file containing records for its process-local events. After program termination, Scalasca loads the trace files into main memory and searches them in parallel for patterns of inefficient performance behavior using the above-mentioned replay mechanism. The result is a trace-analysis report similar in structure to the summary report but enriched with higher-level communication and synchronization inefficiency metrics. Both summary and trace-analysis reports contain performance metrics for every function call path and process/thread which can be interactively

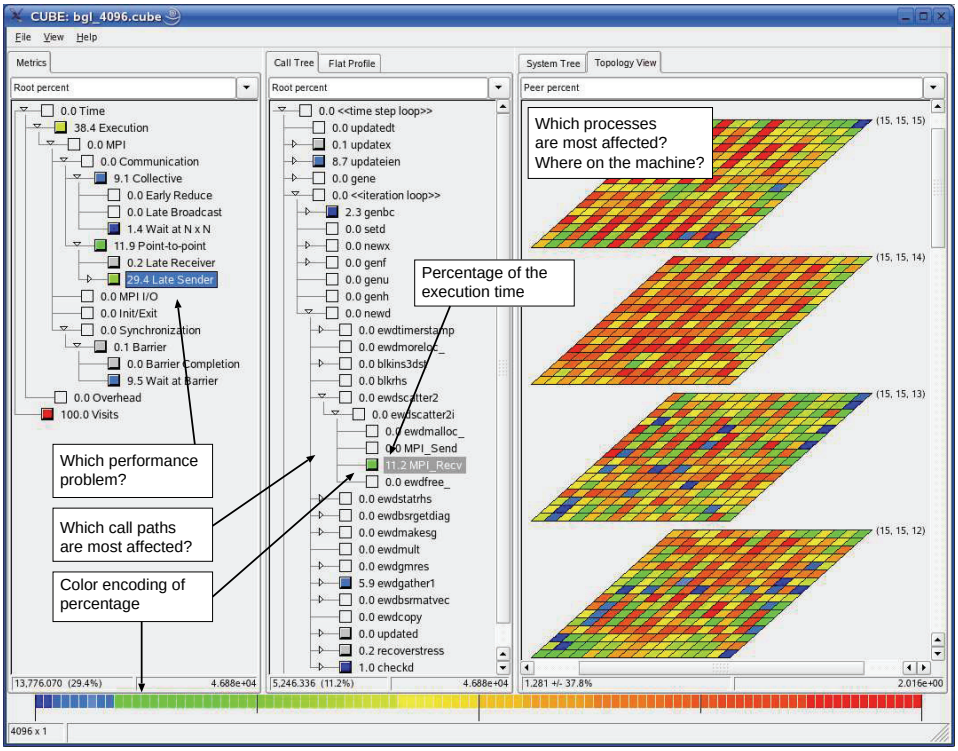


Figure 1.5: Scalasca's trace-analysis report.

examined in the provided analysis-report browser shown in Figure 1.5. The tree in the left window pane displays patterns of inefficient performance behavior arranged in a specialization hierarchy. In addition, the middle window pane shows the distribution of the selected pattern's severity across the call tree. Finally, the right window pane shows the distribution of the pattern's severity at the selected call path across the machine topology.

When an application is traced, runtime events critical to communication and computation activities are intercepted and temporarily stored in main memory. These performance-relevant events are subsequently written to a trace file according to the Scalasca event model. Performance-relevant events include entering and leaving functions or other code regions as well as sending and receiving point-to-point messages or participation in collective communication. Whereas the communication-related event types are crucial to study the interactions among different processors and to identify wait states, function entries and exits are needed to understand the computational requirements and the context in which the most demanding communication operations occur.

Table 1.1: Event types of the Scalasca event model as used in this thesis.

Name	Description	Abbr.
Programming-model independent event types		
Enter	Region entered	E
Exit	Region left	X
MPI-related event types		
Send	Message sent	S
Receive	Message received	R
MPI Collective Exit	MPI collective code region left	MX
OpenMP-related event types		
Fork	Master thread creates a team of threads	F
Join	Worker threads finish their execution	J
OpenMP Collective Exit	OpenMP collective code region left	OX
Lock-Acquisition	Lock variable acquired	L
Lock-Release	Lock variable released	U

Given that this thesis focuses on a synchronization method to be used within Scalasca, the Scalasca event model is described in more detail. The Scalasca event model has been designed to provide a uniform data representation suitable for MPI, OpenMP, and hybrid applications that use MPI and OpenMP in combination. The model maps events onto their location within the hierarchical hardware (i.e., machine and node) and to their process and thread of execution. It supports the storage of all necessary source code and call-site information, recording of performance metrics, such as hardware counters [18], and marking of collectively executed operations for both MPI and OpenMP. For the tracing of OpenMP-related events, Scalasca uses the POMP performance interface [69] and assumes that the same team of threads is used throughout the entire execution in funneled mode. As tracing of OpenMP ordered, task, and taskwait sections is not supported within Scalasca, it does also not account for OpenMP nested and task parallelism. In addition to clusters, target systems can also be metacomputing environments [12].

The information Scalasca records for an individual event includes at least the timestamp, the location (i.e., 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 and events related to MPI operations and OpenMP constructs. Table 1.1 lists all programming-model independent as well as MPI- and OpenMP-related event types of the Scalasca event model [94] along with a brief explanation and their abbreviation as used in this thesis. In addition, Table 1.2 lists the event attributes for each event type.

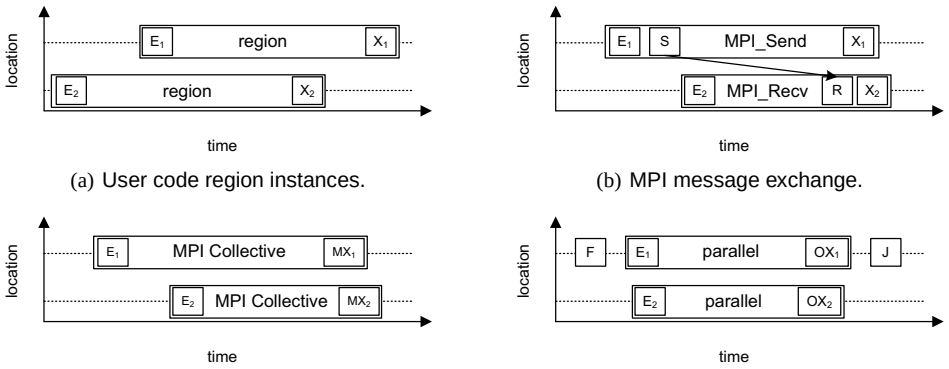
Programming-model independent events indicate that a program enters or leaves a code region. Such a code region of a parallel program may be a function, a loop, or just a basic block [90]. One execution of a region forms a region instance. As a consistency requirement, regions must be left in the reverse order they are entered. That is, the region that has been entered last at a location must be left first on that location. Figure 1.6 illustrates typical event sequences

Table 1.2: Event attributes of Scalasca's event types.

Event type	Event type attributes
General attributes for all event types	
	Location Timestamp
Programming-model independent event types	
Enter	Region entered Hardware counter values (optional)
Exit	Region exited Hardware counter values (optional)
MPI-related event types	
Send	Destination location of message Communicator Message tag Message length in bytes
Recv	Source location of message Communicator Message tag Message length in bytes
MPI Collective Exit	Collective region left Root location of the operation Communicator Bytes sent Bytes received Hardware counter values (optional)
OpenMP-related event types	
Join	
Fork	
Lock-Acquisition	Lock variable acquired
Lock-Release	Lock variable released
OpenMP Collective Exit	Collective region left Hardware counter values (optional)

according to the Scalasca event model exemplified with the time lines of two locations (i.e., processes or threads). For example, Figure 1.6(a) shows the event sequences of user code region instances consisting of an enter (E_i) and exit (X_i) event record on each location.

Moreover, MPI-related events include events representing point-to-point operations, such as sending and receiving messages, and events representing the completion of MPI collective operations. Figure 1.6(b) shows a point-to-point message exchange between two locations. The respective send (S) and receive (R) event records are enclosed by enter (E_i) and exit (X_i) event records indicating that a sending or receiving region has been entered or left. Here, the MPI message semantics in combination with event attributes (e.g., destination location, source location) enable the determination of the correct logical event sequence. Note that the send



(a) User code region instances. (b) MPI message exchange.

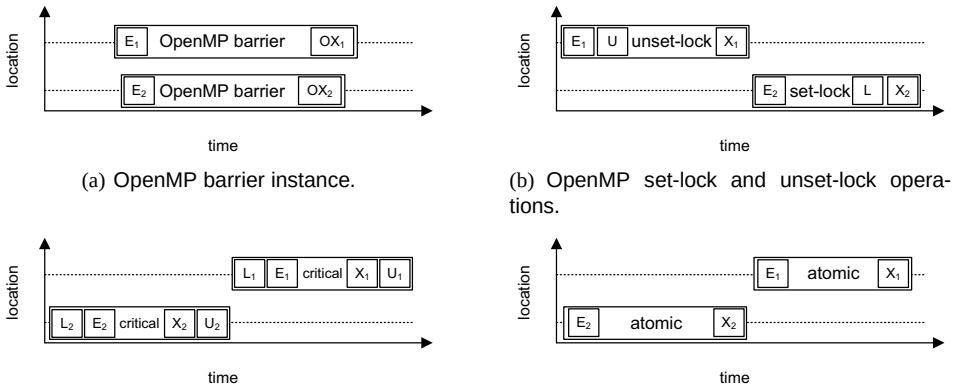
(c) MPI collective operation instance. (d) OpenMP parallel region instance.

Figure 1.6: Typical event sequences according to the Scalasca event model.

event always marks the beginning of a send operation, whereas a receive event marks the end of a receive operation. Assuming globally synchronized clocks, this scheme guarantees that the timestamp of the receive event is larger than the timestamp of the send event. In addition, Figure 1.6(c) presents a collective operation instance, on each location marked by enter (E_i) and collective exit (MX_i) event records. These collective exit events are specializations of normal exit events carrying, among other attributes, information on the communicator. This information allows identifying concurrent collective exits belonging to the same collective operation instance.

Finally, OpenMP-related events include events that represent the creation and termination of a team of threads, leaving parallel regions or barriers executed in parallel, and acquiring and releasing lock variables. As can be seen in Figure 1.6(d), a fork (F) event record indicates that the master thread creates a team of threads, whereas a join (J) event record indicates that this team of threads is terminated. The location of both events is always the location of the master thread. In this situation, the enter (E_i) event records indicate that a program enters a parallel region, whereas the OpenMP-related collective exit (OX_i) event records indicate that the program leaves the parallel region.

Figure 1.7 shows typical event sequences inside OpenMP parallel regions with the time lines of two locations (i.e., threads). For the sake of simplicity, fork and join event records are left out. Figure 1.7(a) presents an OpenMP barrier instance, on each location enclosed by an enter (E_i) and OpenMP collective exit (OX_i) event record. In addition, lock event semantics are illustrated in Figure 1.7(b). The respective lock-acquisition (L) and lock-release (U) event records are enclosed by enter (E_i) and exit (X_i) event records indicating that a thread entered or left an `omp_set_lock` or `omp_unset_lock` region. Whereas the lock-acquisition (L) event record indicates that a lock variable is acquired by a thread (i.e., shared variable locked), the lock-release (U) event record indicates that this lock is released (i.e., shared variable unlocked). In the Scalasca event model, the correct sequence of lock events is only given by the timestamp of the respective events. Assuming that the thread-local clocks are synchronized,



(a) OpenMP barrier instance.

(b) OpenMP set-lock and unset-lock operations.

(c) OpenMP critical region instance.

(d) OpenMP atomic region instance.

Figure 1.7: Typical event sequences inside OpenMP parallel regions according to the Scalasca event model.

the timestamp could be used to determine the logical sequence of locks during the program run. However, on some systems this assumption cannot be maintained and so a further global event attribute, such as a sequence count which is incremented once a lock is acquired, is necessary to decide which thread acquires a lock after another thread has released it. So far, the Scalasca event model does not provide such an attribute. Note that the lock-acquisition (L) event record always marks the end of an set-lock operation, whereas the lock-release (U) event record marks the beginning of the unset-lock operation. Given that a critical construct restricts the execution of a structured block to a single thread at a time, the Scalasca event model uses lock-acquisition and lock-release event records to indicate when the associated structured block was locked or unlocked. Both events are recorded inside the critical section: the lock-acquisition event record at the beginning and the lock-release event record at the end. More specifically, as can be seen in Figure 1.7(c), before the structured block of the critical region is entered (E_i event records) it is locked (L_i event records), and is again unlocked (U_i) after the structured block is left (X_i event records). Finally, only one thread is allowed to execute an OpenMP atomic construct at a time (see Figure 1.7(d)). In general, an atomic region is only described by enter (E_i) event records and exit (X_i) event records. However, these events are recorded before the region was entered and after the region was left because tracing inside such a region is not possible. To determine which thread executed the atomic region before another thread, it would be necessary to record when a thread executes the atomic region. Unfortunately, the execution of an atomic region is restricted to statements that can be calculated atomically and so it is not possible to insert event tracing calls. Note that tracing inside atomic regions would be inefficient, because it would impose a large and non-negligible overhead.

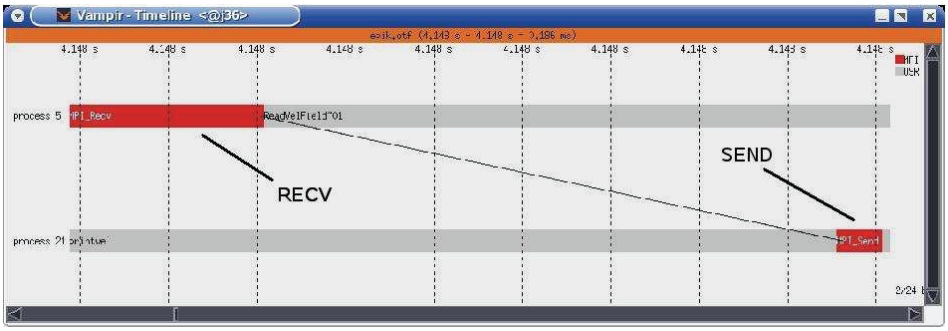


Figure 1.8: Time-line visualization of a message exchange in backward direction.

1.5 Timestamp Synchronization

In general, the accuracy of trace analysis depends on the comparability of timestamps taken on different processors. Inaccurate timestamps may cause a given interval to appear shorter or longer than it actually was, or change the logical event order, which requires a message to be received only after it has been sent. This is also referred to as the *clock condition*. Inaccurate timestamps may lead to false conclusions during performance analysis, for example, when the impact of certain behaviors is quantified, or – even more strikingly – may confuse the user of trace-visualization tools such as Vampir by causing arrows representing messages to point backward in time-line views (see Figure 1.8). Moreover, tools such as KOJAK may also cease to work in a satisfactory manner if they rely on message event orders imposed by the communication substrate to which an operation belongs.

To avoid clock condition violations, the error of timestamps should ideally be smaller than one half of the message latency μ. For instance, let us assume that a send event appears $\frac{1}{2}\mu$ too early while the matching receive event appears $\frac{1}{2}\mu$ too late in the trace. If we now consider a message delay of exactly the message latency, the send event appears at the same time as the matching receive event does, which is impossible and considered a clock condition violation. While some custom-built clusters such as IBM Blue Gene offer relatively 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). Clock synchronization protocols such as NTP [68] can align the clocks to a certain degree, but are often not accurate enough for the purposes of program observation. Assuming that every local clock on a parallel machine runs at a different but constant speed (i.e., drift), the (global) time of an arbitrarily chosen master clock can be calculated locally as a linear function of the local time. For this purpose, offset measurements may be performed between all local clocks and the master clock once at program initialization and once at program finalization. However, as the assumption of constant drifts is only an approximation, violations of the clock condition may still occur – especially when the offset measurements are taken with long intervals in between. Figure 1.9 shows clock deviations after linear offset interpolation measured using a simple benchmark program that was executed for 3600 seconds on an Intel Xeon cluster.

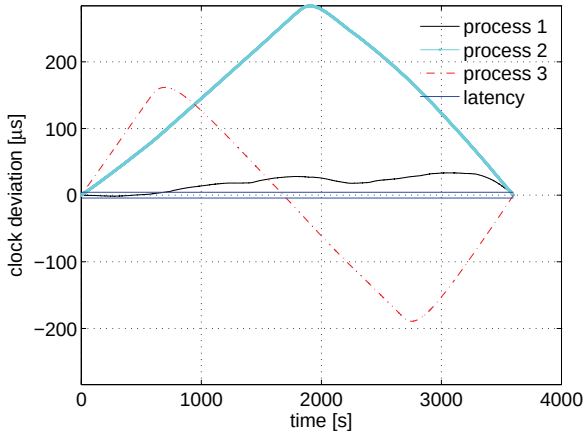


Figure 1.9: Non-linear offsets of physical clocks measured on a Xeon cluster in comparison to the send-receive latency.

As can be seen, the non-linearity of local clocks caused clock errors much larger than the send-receive latency.

While the errors of single timestamps are hard to assess, clock condition violations can be easily detected and offer a toehold to increase the fidelity of inter-process and inter-thread timings. The *controlled logical clock* (CLC) [79] is a method to retroactively correct timestamps violating the clock condition. As the modification of individual event timestamps might change the length of local intervals and even introduce new violations, the correction takes the context of the modified event into account by carefully adjusting the local time axis from the immediate past of the affected event to the end of the local trace. This algorithm is, however, not suitable for realistic parallel programs because (i) it ignores collective and shared-memory communication and (ii) as a serial algorithm it offers only limited scalability. The original CLC algorithm cannot be used to correct clock condition violations among MPI collective events and OpenMP events. Even more strikingly, the algorithm may introduce new violations because it ignores happened-before relations among MPI collective events and OpenMP events while correcting point-to-point event semantics.

Focusing onto cluster systems but not losing sight of the more general metacomputing case, the contribution of this thesis is fourfold:

1. The thesis investigates the robustness of linear offset interpolation across a range of timer technologies available on different platforms and shows that the error of timestamps derived in this way can easily compromise the consistency of the logical event order imposed by the event semantics.

2. This thesis extends the algorithm to enable the correction of realistic traces taken from MPI and hybrid applications. The basic idea behind the extension is to consider

collective and shared-memory operations as being composed of multiple point-to-point messages, taking the semantics of the different flavors of operations into account.

3. To accomplish this correction in a scalable way, both distributed memory and parallel processing capabilities are exploited by processing separate local trace files in parallel and replaying the original communication on as many CPUs as were used to execute the target application itself.
4. To employ the replay mechanism in computational grids, this work also defines the necessary infrastructure to accurately measure clock offsets in distributed environments based on hierarchical networks.

The remainder of this thesis is structured as follows: Chapter 2 starts with a description of the most common clock types and their accessibility, followed by the investigation of the robustness of linear offset interpolation including the influence on timestamps of concurrent events. While Chapter 3 reviews related work in general, Chapter 4 introduces the original serial version of the CLC algorithm including its limitations in more detail. In Chapter 5, the extensions necessary to correctly synchronize collective and shared-memory operations are introduced. Then, Chapter 6 presents the parallel algorithm design and describes its implementation within Scalasca. Chapter 7 evaluates the scalability of the parallel version, and also shows that the collaterally introduced deviations of local interval lengths remain within acceptable limits. Finally, Chapter 8 summarizes the thesis research and outlines future work.

1. INTRODUCTION

Chapter 2

Processor Clocks

Measuring the time between concurrent events requires a global clock, which is often unavailable on clusters. Assuming that the potentially different drifts of local clocks remain constant over time, linear offset interpolation can be applied postmortem to map local timestamps onto global timestamps. In this chapter, the robustness of the above assumption is investigated using different types of timers. It is shown that 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. This indicates that linear offset interpolation alone may be insufficient for many applications of event tracing.

2.1 Classification

Processor clocks are used to obtain event timestamps and can be characterized in terms of their relative offset and drift. The clock offset is the time difference between two clocks at a given time, whereas the clock drift is the rate at which a clock progresses over time, which may also be different for two clocks. Figure 2.1 shows two clocks with both an initial offset and different but constant drifts. Assuming that clocks have different but constant drifts, the (global) time of an arbitrarily chosen master clock can be calculated locally as a linear function of the local time. However, the rate at which the offset changes over time (i.e., clock drift) is usually time dependent. Given that different clock types may exhibit different offset and drift characteristics, this section reviews the most common clock types and explains how they can be accessed [9].

2.1.1 Clock Types

Different types of clocks are used to measure and maintain the processor time. Clocks based on cycle counters use the processor clock signal to increment an internal counter on each tick. The step size, which depends on the clock rate, may change over time, as state-of-the-art power management may dynamically slow down or accelerate the signal. As a consequence, remote cycle counters are very hard to synchronize and therefore only useful to compare events happening on the same CPU.

In contrast, hardware clocks, often called timestamp counters, use specialized hardware counters. Based on separate oscillators, their values are incremented on each tick of the

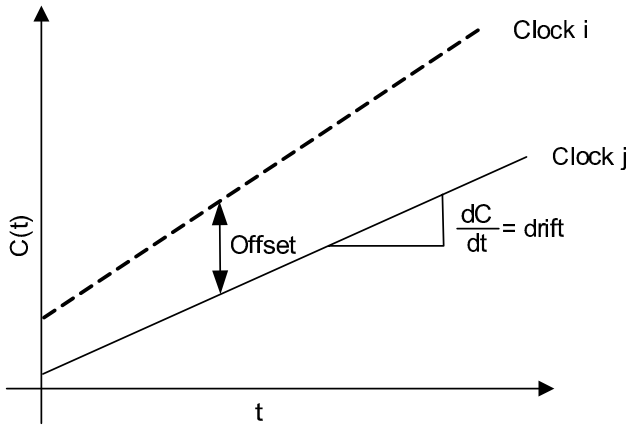


Figure 2.1: Two clocks with both an initial offset and different but constant drifts.

oscillator, and thus, their step size does not depend on a potentially unstable processor clock rate. Although a single hardware clock can provide accurate relative timings, synchronization among multiple remote hardware clocks is usually not provided.

As another alternative, software clocks are realized in the form of user or library functions. Often, those clocks are implemented entirely in software without any dedicated hardware support. Software-based synchronization among different software clocks (e.g., via NTP) may guarantee synchronized time values to a certain degree [67].

Finally, system clocks are specializations of software clocks and managed by the operating system (e.g., `gettimeofday()`). Such system clocks are usually based on cycle counters, hardware clocks, or software clocks and maintain the system-local time.

As examples of hardware clocks, we consider IBM's real-time clock (RTC), IBM's time base register (TB), and Intel's timestamp counter register (TSC). All these clocks are 64-bit special-purpose registers. RTC counts seconds and nanoseconds, while TB and TSC return the number of ticks counted since processor reset. In contrast, `MPI_Wtime()` must be classified as a software clock that can be used to transparently query clock values on cluster systems. Open MPI [74], a widely used open-source MPI library, chooses among a rich set of implementations for `MPI_Wtime()` at configuration time. The default is `gettimeofday()`, which often relies on network-based synchronization via NTP [68]. The general idea behind NTP is to synchronize distributed clocks before reading them. The distributed clocks query the global time from reference clocks, which are often organized in a hierarchy of servers. NTP uses widely accessible and already synchronized primary time servers. In addition, secondary time servers and clients can query time information via both private networks and the Internet. To reduce network traffic, the time servers are accessed only in regular intervals to adjust the local clock. Jumps are avoided by changing the drift while leaving the actual time unmodified. Unfortunately, varying network latencies limit the accuracy of NTP to about one millisecond

compared to a few microseconds required to accurately trace MPI applications running on clusters equipped with modern interconnect technology [67].

2.1.2 Clock Accessibility

Access to processor clocks is provided either locally or globally. Global accessibility implies that each processor has access to the same clock over an interconnection network within either the entire machine or only within a single partition (e.g., SMP-node or multicore-chip). Because every access either takes exactly the same amount of time regardless of the origin of the request or the exact amount of time is always known and can be used for local correction, global accessibility usually guarantees high accuracy. Even though each access introduces a certain, and usually not negligible overhead, no further synchronization is required, which can even be counted against the initial overhead. More precisely, a global clock request consumes more time than a local clock request, but necessitates no postprocessing, thus, reducing the overall time needed for such a clock access. As a typical representative of a global clock, the IBM Blue Gene/P system [54] offers a hardware clock that is globally accessible across the entire machine. In comparison, local accessibility means that each processor has only access to its own local clock. Of course, querying local clocks incurs less overhead because no data transfer over interconnection networks is required. On the other hand, the synchronization of remote clocks may create new overhead. Note that, in general, it cannot be assumed that processor-local clocks within the same SMP node are perfectly synchronized, as individual chips may provide their own timestamp counters, such as Intel Xeon multi-core chips [56].

Clock accesses can be further classified as either non-transparent or transparent. Non-transparent access means a clock is queried directly, with all necessary calculations to yield the final time value left to the user. These calculations may include the multiplication with a scaling factor or the mapping onto a predefined start time. During a transparent clock access, in comparison, the user queries appropriate software functions that already incorporate these functionalities.

2.2 Requirements of Event Tracing

In this section, we formulate the requirements distributed clocks must satisfy to allow the generation of event traces that are suitable for analyzing parallel applications. After explaining the basic scenario of event trace generation, the actual accuracy requirement is presented and potential implications of inaccuracies on the timestamps of concurrent events are discussed. A detailed introduction to the technique of event tracing can be found in Chapter 1.

2.2.1 Event Trace Generation

In order to observe the runtime behavior of an application, it is typically necessary to insert additional code fragments into the application. The process of adding this extra code into the application is called *instrumentation*. Usually, the extra code is a function call (i.e.,

tracing call) to a *tracing library* to which the application needs to be linked. The tracing library is responsible for recording necessary data enabling the observation of the application's runtime behavior. The locations within the application where the code is added are referred to as *instrumentation points*. Most commonly, these points are function entries and exists, statements, loop entries and exits, or instructions.

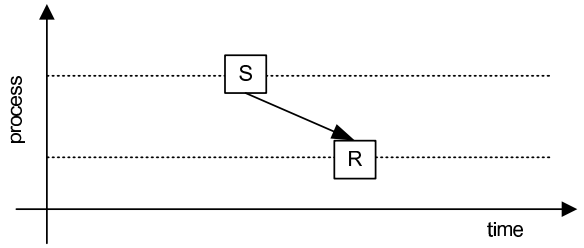
A simple way of inserting instrumentation code into an application is specified by the MPI standard [66]. All MPI library calls also exist with a second entry point name in the profiling message-passing interface (PMPI), allowing a user or tool developer to provide an interposed wrapper library intercepting MPI calls issued by the user code. A complementary approach applicable to user code is to leverage the capabilities provided by many of today's compilers to automatically instrument the entry and exit points of functions [41]. Moreover, instrumentation can take place on the source-code level [42]. Here, a source-code preprocessor parses source files and adds tracing calls at the entry and exit points of functions. For instance, the OPARI [69] source-code preprocessing tool specifically focuses on instrumenting OpenMP directives, but is also well suited to instrument user functions through user-defined directives. In contrast to source- and compiler-level instrumentation, the binary instrumentation technique [20, 27] inserts tracing calls after the program's binary code is generated. In this case, the additional instrumentation code is injected either at runtime by patching the program's binary image in memory, or through rewriting the program executable prior to execution. Finally, users can also manually instrument their application by directly adding tracing calls to the program.

Whenever the running application executes a tracing call, an event is generated in the tracing library, which takes the current time and writes an event record with a corresponding timestamp to a memory buffer. The buffer contents are flushed to disk when necessary. Events typically recorded by MPI and/or OpenMP applications include entering and leaving code regions, sending and receiving point-to-point messages, and events related to collective communication or synchronization. The clock is read locally to minimize the intrusion overhead associated with timestamp creation, as querying a remote clock across the network would consume too much time and introduce inaccuracies caused by varying clock-reading latencies. As a consequence, the timestamps taken on most cluster nodes stem from insufficiently synchronized local clocks.

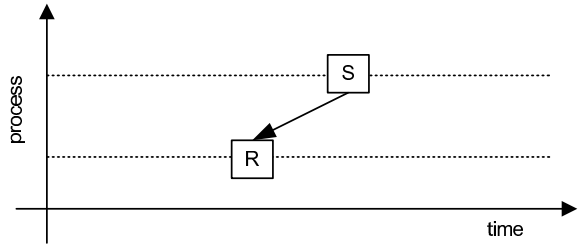
2.2.2 Accuracy Requirements

The accuracy of most trace analyses depends on the comparability of timestamps taken on different processors. Inaccurate timestamps can not only cause a given interval to appear shorter or longer than it actually was, but also change the logical event order, which requires that a message can only be received after it has been sent. This is also referred to as the *clock condition* [62]. The clock condition, which is given in Equation 2.1, requires that a receive event occurs at the earliest l_{min} after the matching send event, with l_{min} being the minimum message latency.

$$t_{recv} \geq t_{send} + l_{min} \tag{2.1}$$



(a) Consistent message-passing event trace: The message is received after it has been sent.



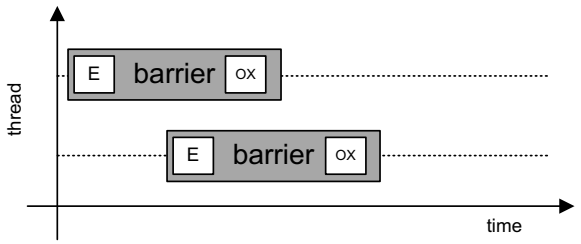
(b) Inconsistent message-passing event trace: The message is received before it has been sent.

Figure 2.2: Implications of inaccurate timestamps for message-passing (MPI) event semantics.

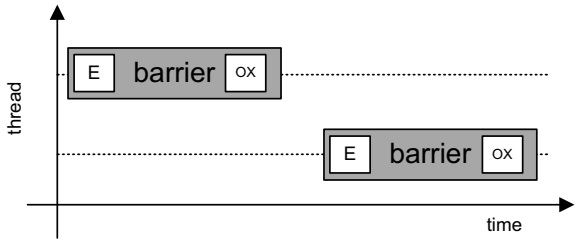
To avoid violations of this condition, the timestamp error should ideally be smaller than one half of the message latency μ. Once a send event appears $\frac{1}{2}\mu$ too early, while the matching receive event appears $\frac{1}{2}\mu$ too late in the trace and we further assume a message delay of exactly the message latency, the send event appears at the same time in the event trace as the matching receive event does, which is impossible and considered a clock condition violation. A typical clock quartz with a drift of only 1 min/year will already cause a deviation of 2 μs after 1 s, roughly corresponding to the latency of many modern interconnection networks. Analogous requirements can be derived for alternative communication mechanisms, such as collective communication or synchronization, by mapping their semantics onto point-to-point communication.

2.2.3 Implications of Inaccuracies

The potential implications of inaccurate timestamps for the semantics of message-passing and shared-memory events are exemplified in Figures 2.2 and 2.3. The correct message-passing event order shown in Figure 2.2(a) is violated in Figure 2.2(b). The two diagrams show the time lines of two processes exchanging a message via a send (S) and a receive (R) event. In the second picture, the measurement suggests that the message has been received before it has



(a) Consistent shared-memory event trace: The execution of the barrier by both threads overlaps.



(b) Inconsistent shared-memory event trace: The execution of the barrier by both threads does not overlap.

Figure 2.3: Implications of inaccurate timestamps for shared-memory (OpenMP) event semantics.

been sent, which, of course, is impossible. Obviously, such inconsistencies may also occur during collective message-passing operations.

In Figure 2.3, the two diagrams present a similar case that may occur in OpenMP programs when writing traces according to the POMP event model [69]. Shown is the execution of an OpenMP barrier by two threads involving two different event types: entering (E) and exiting the barrier (OX). Whereas in Figure 2.3(a) the event order is consistent, in Figure 2.3(b) one thread leaves the barrier before the other one has entered it, constituting a clear violation of barrier semantics.

As the screenshot in Figure 2.4 demonstrates, such violations can indeed occur in practice. The figure shows the time-line visualization of an event trace taken from an OpenMP benchmark program executed with four threads on an Intel Itanium node with four chips and four cores per chip. As can be seen in the encircled area, thread 1:2 seems to have left the barrier (red bars or medium dark) before thread 1:3 had a chance to enter it. Besides violations of barrier semantics, OpenMP applications may also suffer from misrepresentations of other happened-before relationships specified in the POMP event model, such as the rule that all events belonging a parallel region must be temporally enclosed by fork and join events.

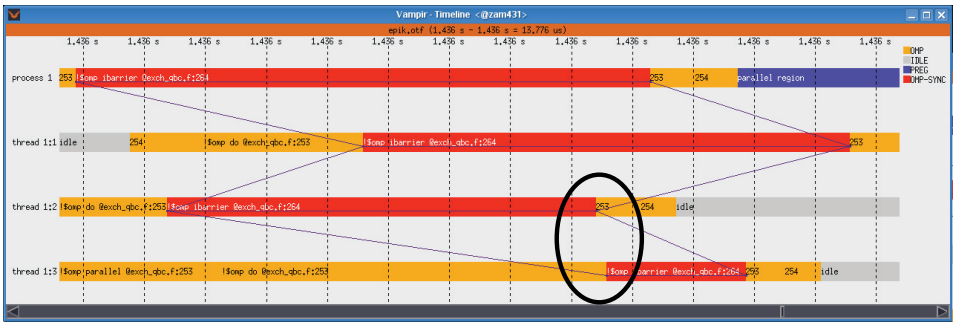


Figure 2.4: A violation of OpenMP barrier semantics observed on an Itanium SMP node.

2.3 Linear Offset Interpolation

The most significant deviations between non-synchronized clocks result from differences in offset and drift. In a simple model assuming different but constant drifts, the local (i.e., worker) time can be mapped onto the global time of an arbitrarily chosen master postmortem via linear offset interpolation based on prior offset measurements. Offset values among participating clocks are measured either at program initialization [34] or at initialization and finalization, and subsequently used as parameters of the linear correction function [51, 64].

Offsets between master and workers can be determined using Cristian's probabilistic remote clock reading technique [25]. Figure 2.5 illustrates this technique with time lines of a master and worker process exchanging two messages. As can be seen in this figure, the master process sends a request to a remote worker process at time m_1, the worker responds by sending back its current local time w_0, which is received by the master at time m_2. Assuming that the two message delays have equal length, the offset can be calculated according to Equation 2.2.

$$o = m_1 + \frac{m_2 - m_1}{2} - w_0 \tag{2.2}$$

Since, contrary to our assumption, real message communication is prone to irregular delays, the process must be repeated several times to minimize the measurement error. In order to prevent the program from being perturbed, offset measurements are usually avoided while a program is running. Hence, we use offset values among participating clocks which are measured either at program initialization (i.e., offset alignment) or at initialization and finalization (i.e., linear offset interpolation). Nonetheless, a recent approach by Doleschal et al. [29] proposes periodic offset measurements during global synchronization operations while limiting the effort required in each step by resorting to indirect measurements across several hops (see Chapter 3).

A single offset measurement to an arbitrary chosen master clock at program initialization can be used to align different clocks by simply adding the measured offset value to the local

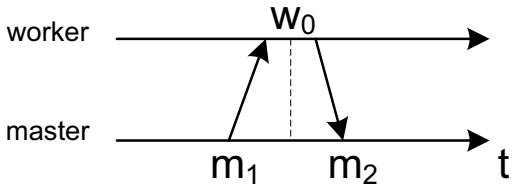


Figure 2.5: Cristian's probabilistic remote clock reading technique.

clock value. Assuming two measurements, one at the beginning and one at the end of the program run, every remote worker eventually has two pairs (w_1, o_1) and (w_2, o_2) that contain its local worker time together with the offset to the matching master time when the two measurements were taken. The master time m can now be calculated from the worker time w using Equation 2.3.

$$m(w) = w + \frac{o_2 - o_1}{w_2 - w_1} \times (t - w_1) + o_1 \tag{2.3}$$

2.4 Sources of Inaccuracy

While the above scheme might prove satisfactory for short runs, measurement errors and time-dependent drifts may create inaccuracies and clock condition violations during longer runs. Additionally, repeated drift adjustments caused by NTP may impede linear offset interpolation, as they deliberately introduce non-constant drifts.

Inaccurate timestamps are often the result of either unstable clock drifts or measurement errors. Varying temperature and flexible power management provided by modern microprocessors may alter oscillation frequencies, causing clocks to gradually diverge as time progresses [67]. Temperature variations may cause drift variations of more than 10^{-8} resulting in synchronization errors of more than 1 μs after $100\ s$. Moreover, insufficient timer resolution may introduce measurement errors, an effect exacerbated by OS jitter. Jitter interference is primarily caused by scheduling daemon processes or handling asynchronous events such as interrupts on the side of the operating system.

Although all of the above influences are predictable to some degree, modeling them correctly will require intimate knowledge of the underlying hardware and software infrastructure, which is usually not available to developers of generic cluster tools. From that perspective, this behavior can therefore be classified as non-deterministic.

Finally, network topology and load may adversely affect the predictability of message latencies, an important prerequisite for network-based synchronization. As messages travel through various stages of the network, the processing time in each stage may vary depending on the current network load. Since messages exchanged between the same pair of locations may require different amounts of time, error correction based on assumptions about the message latency remains challenging.

2.5 Clock Evaluation

The primary objective of this chapter is to evaluate the effectiveness of linear offset interpolation as an instrument for the postmortem synchronizations of timestamps in event traces of parallel applications. For this purpose, this section now reviews measurements conducted using different timers on a selection of typical cluster architectures:

Xeon cluster: Located at the Center for Computing and Communication of RWTH Aachen University, this cluster consists of 60 compute nodes, each with 2 quad-core Intel Xeon processors running at 3.0 GHz. The compute nodes communicate primarily through an InfiniBand network.

PowerPC cluster: Located at the Barcelona Supercomputing Center, this cluster (aka MareNostrum) consists of 2,560 IBM JS21 blade compute nodes, each with 2 dual-core IBM 64-bit PowerPC 970MP processors running at 2.3 GHz. The compute nodes communicate primarily through a Myrinet network with Myrinet adapters integrated on each server blade.

Opteron cluster: Located at the National Center for Computational Sciences at Oak Ridge National Laboratory, this cluster (aka Jaguarcnl) consists of 3,744 XT3 compute nodes, each with one dual-core AMD Opteron processor running at 2.6 GHz. Each node is connected to a distinct Cray SeaStar router through HyperTransport with all the SeaStars arranged in a 3-D-torus network topology.

In a first step, residual clock deviations were measured after applying

(i) offset alignment only at program initialization so that all clocks started from zero and

(ii) linear offset interpolation based on offset measurements both at program initialization and finalization, as described in the previous section.

To reflect varying application runtimes, short (300 s), medium (1800 s), and long (3600 s) measurement runs were performed. All processes were located on different SMP nodes. In a next step, the actual frequency of clock condition violations was measured in event traces

Table 2.1: Xeon cluster: Process pinning for measurements among SMP nodes, chips, and cores.

	Process pinning
Inter node	4 nodes 1 process per node
Inter chip	1 node 2 chips per node 1 process per chip
Inter core	1 node 1 chip per node 4 processes per chip

Table 2.2: Xeon cluster: Measured message and collective latencies for different measurement setups.

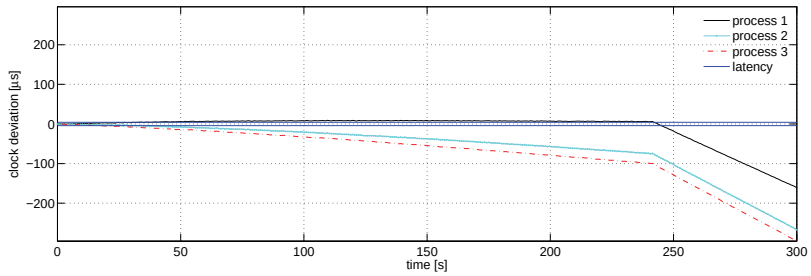
	mean [μs]	std. dev. [μs]
Inter node message latency	4.29	9.80E-04
Inter chip message latency	0.86	4.77E-05
Inter core message latency	0.47	6.94E-06
Inter node collective latency	12.86	1.68E-02

of two MPI applications. The first application was the Parallel Ocean Program (POP), which is shipped with the SPEC MPI2007 1.0 benchmark suite [85]. The second application was the MPI version of the ASC SMG2000 benchmark [17], a parallel semi-coarsening multigrid solver that uses a complex communication pattern and performs a large number of non-nearest-neighbor point-to-point communication operations. Finally, taking the hierarchical structure of modern multicore-based cluster architectures into account, we attempted to assess the chance of clock condition violations occurring in MPI or OpenMP codes when processes are placed on the same SMP node but on different chips or on the same chip, as shown in Table 2.1 for the Xeon cluster.

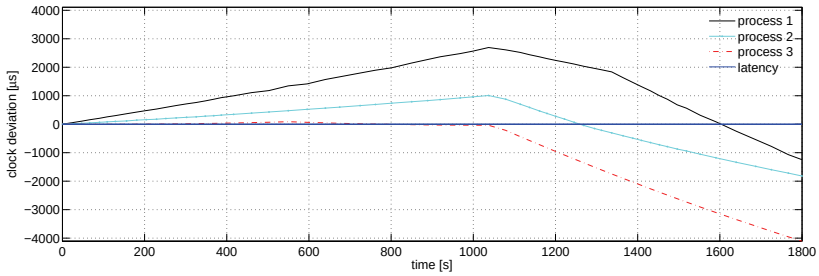
2.5.1 Clock Deviations

As described in Section 2.2, the error of timestamps should ideally be smaller than one half of the message latency to generate traces suitable for parallel-program analysis. Since messages latencies between cores on a single chip, between chips on a single SMP node, and between different SMP nodes usually differ, the message latency was measured for all these cases. For the first case, the collective all-reduce latency was also measured. As Table 2.2 shows for the Xeon cluster, the latency exhibits significant variations depending on the relative location of processes.

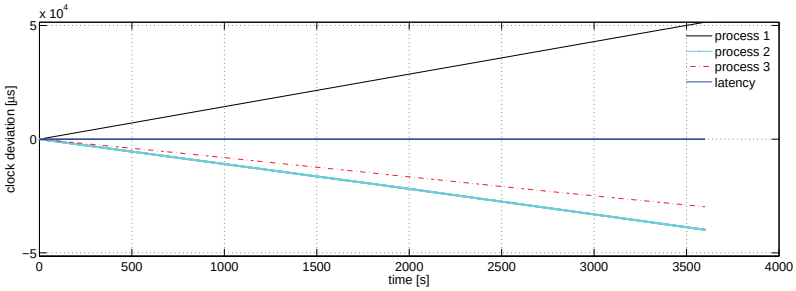
To evaluate the effectiveness of timers for linear offset interpolation, clock deviations of `MPI_Wtime()`, `gettimeofday()`, and the Intel timestamp counter were measured on the Xeon cluster during runs of increasing duration after an initial alignment of offsets. The results are shown in Figure 2.6. Obviously, `MPI_Wtime()` (Figure 2.6(a)) produces severe clock deviations of more than 200 μs already after a relatively short period. Interestingly, the deviation seems to grow roughly at a constant rate up to a turning point at which the slope abruptly changes. After this point, the affected processes continue striding away linearly but at a much higher rate. `gettimeofday()` (Figure 2.6(b)) exhibits a very similar drift pattern, again showing phases of roughly constant drift interrupted by sudden drift adjustments – albeit a little bit more curvy at least in one instance. The changes are presumably caused by the underlying NTP synchronization, which periodically corrects the drift to prevent the clocks from diverging too far. This, of course, is detrimental to linear offset interpolation, as it deliberately introduces non-constant drifts. In sharp contrast to the previous two measurements, however, the Intel timestamp counter (Figure 2.6(c)) appears to maintain an approximately constant clock drift rate even across a very long period of time. Obviously,



(a) MPI_Wtime(): Clock deviations during a short run.

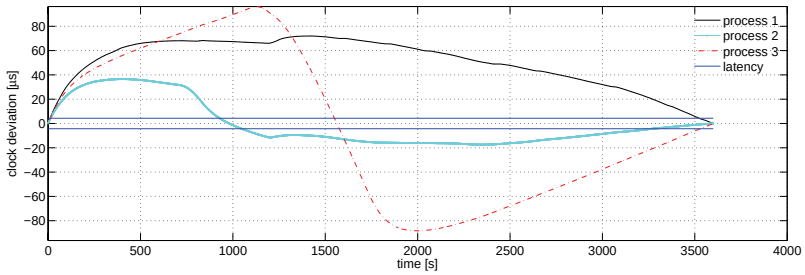


(b) gettimeofday(): Clock deviations during a medium run.

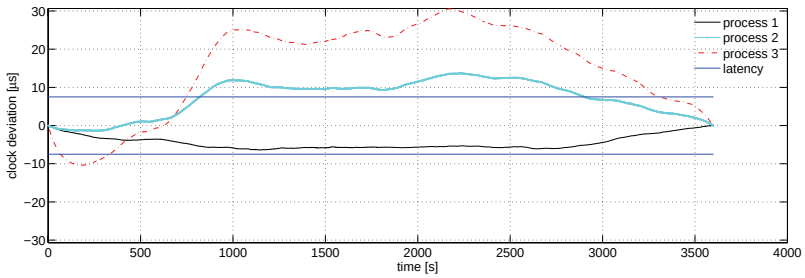


(c) Intel timestamp counter: Clock deviations during a long run.

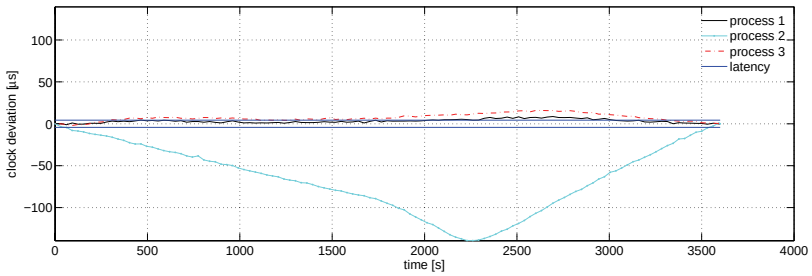
Figure 2.6: Xeon cluster: Measured clock deviations of different timers during short, medium, and long measurement runs after an initial offset alignment.



(a) Xeon cluster: Clock deviations using the Intel timestamp counter.



(b) PowerPC cluster: Clock deviations using the IBM time base register.



(c) Opteron cluster: Clock deviations using gettimeofday().

Figure 2.7: Measured clock deviations of two different hardware clocks and gettimeofday() during long runs after linear offset interpolation.

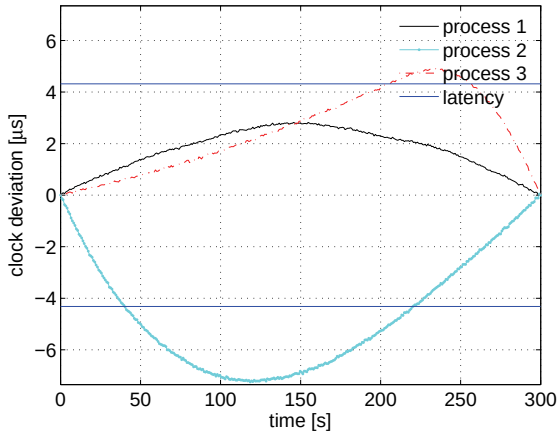


Figure 2.8: Measured clock deviations after linear interpolation during a short run on the Xeon cluster using the Intel timestamp counter. The deviations slightly exceed the latency.

hardware clocks seem much more appropriate when it comes to taking the timestamps of concurrent events.

Having reached this conclusion, subsequent experiments therefore focused on the evaluation of hardware clocks on different cluster platforms. Further tests were conducted on the three clusters with Intel's timestamp counter, with IBM's time base register, and for comparison with `gettimeofday()` always using a duration of 3600 s as this mimics a longer application run. Figure 2.7 shows residual clock deviations after performing linear offset interpolation (between initialization and finalization) with an expected convergence of offsets at the end of the run. As can easily be seen, linear interpolation already accounts for the most severe differences in offset and drift, although significant deviations can still be observed, the highest occurring when using `gettimeofday()` on the Opteron system. In fact, measured deviations exceeded the message latency already after a few minutes or even earlier, rendering linear interpolation alone insufficient to guarantee the absence of clock condition violations during longer runs. Since shorter runs also use a shorter interpolation interval, linear interpolation may still be adequate in those cases, although the results of an experiment running for 300 s on the Xeon cluster suggest that even then violations may occur (Figure 2.8).

2.5.2 Clock Condition Violations

To quantify the extent of clock condition violations in traces of real applications, experiments were performed with POP and SMG2000 on the Xeon cluster, each time using 32 processes. To emulate a realistic scenario, we refrained from using a specific process pinning. Instead, the default setting was kept so the scheduler chose the pinning automatically. Traces were

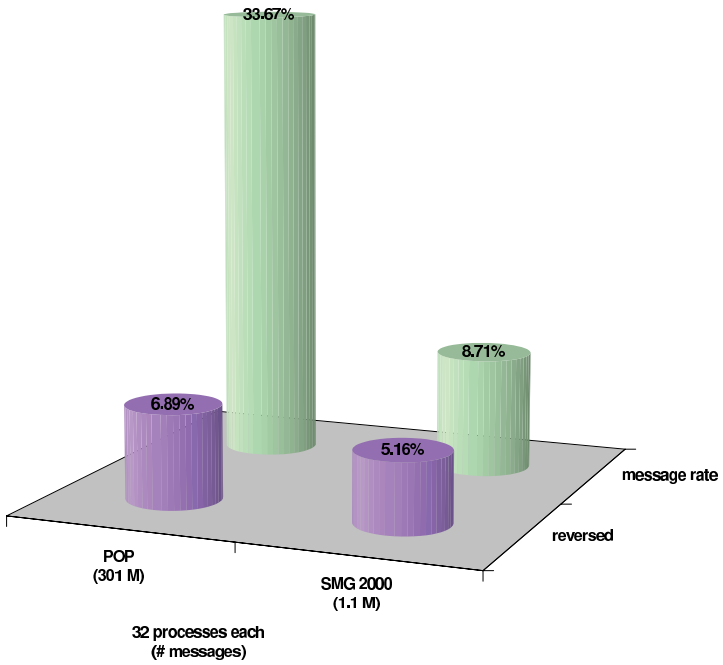


Figure 2.9: Xeon cluster: Percentage of messages with the order of send and receive events being reversed and of message transfer events in relation to the total number of events for SMG2000 and POP.

obtained using the Scalasca toolset [92], which performs linear offset interpolation based on offset measurements taken during `MPI_Init()` and `MPI_Finalize()`. The POP application ran with the mref input data set, causing it to execute 9000 iterations in roughly 25 min. Since tracing the full run would consume a prohibitively large amount of storage space, only iterations 3500 to 5500 were traced. This "partial" tracing corresponds to the recommended practice of tracing only pivotal points of long-running applications that warrant a more detailed analysis. For SMG2000, a problem size of $16 \times 16 \times 8$ per process with five solver iterations was configured. A longer run of SMG2000 was emulated by inserting sleep statements immediately before and after the main computational phase so that it was carried out ten minutes after initialization and ten minutes before finalization. This corresponds to a scenario similar to POP, in which only distinct intervals of a longer run are traced with tracing being switched off in between. For this purpose, 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, the execution times of both POP and SMG2000 in these experiments can still be regarded as an optimistic assumption.

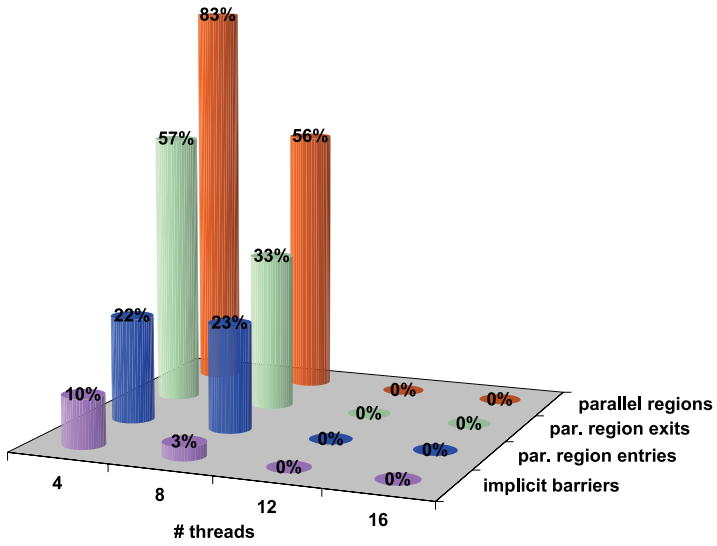


Figure 2.10: Intel Itanium SMP node: Percentages of parallel regions in an OpenMP benchmark program exhibiting clock condition violations across a range of thread counts.

Figure 2.9 shows the frequency of clock condition violations for both applications on the Xeon cluster. The numbers represent averages across three measurements for each application because the number of violations varied between runs. The front row shows the percentage of messages with the order of send and receive events being reversed, while the back row shows the fraction of message transfer events in relation to the total number of events in the trace. The numbers also include logical messages that can be derived by mapping collective communication onto point-to-point semantics. For POP, around 6% of the messages flew backward in time, while for SMG2000 the percentage was roughly 5%. The results underline the hypothesis that linear interpolation alone is insufficient to produce traces free of clock condition violations and that such violations may adversely affect a significant percentage of message events.

Finally, relative deviations of clocks co-located on the same SMP node of the Xeon cluster were examined (i) without any correction, (ii) after aligning only initial offsets, and (iii) after applying linear offset interpolation. These measurements distinguished between processes located on different chips and processes located on the same chip. In all cases, the measured deviations essentially constituted "noise" oscillating around zero with a maximum difference of roughly 0.1 μs between any two clocks in this ensemble. One further conclusion is that on this system MPI message semantics can be easily preserved without further postprocessing of timestamps. A study by Etsion et al. [36] confirmed on similar platforms that the overhead accessing processor clocks through timestamp counters roughly corresponds to such noise.

However, as experiments on an Intel Itanium SMP node with four chips and four 4 cores suggest, on some systems the semantics of OpenMP constructs can be violated: Traces were taken from a simple OpenMP benchmark program that executes a loop whose body contains a single parallel-for construct, which is a short cut for an OpenMP for construct enclosed by a parallel region. The tests were conducted with varying numbers of threads, ranging from 4 to 16. All events were recorded according to the POMP event model. Neither offset alignment nor linear offset interpolation was applied to the timestamps, which were taken using the Intel timestep counter. Figure 2.10 shows the fraction of parallel regions exhibiting clock condition violations. Again, the numbers represent averages across three measurements for each configuration. The row in the back gives the percentage of parallel regions with violations of any kind, while the three rows in the front give the percentages of parallel regions with specific violations: at the region entry (i.e., fork event not the first event), at the region exit (i.e., join event not the last event), and during the implicit barrier. Notably, when using only four threads, more than three quarters of the regions (83%) were affected, with violations at the region exit occurring most frequently. However, the fraction of affected regions drops significantly as the number of threads is increased, with 12 threads causing only very few violations and 16 threads none at all. OpenMP synchronization latencies rising with an increasing number of threads offer a potential explanation. Interestingly, some of the traces showed violations at the region entry but not at the exit and vice versa, which may back the assumption that a more systematic clock deviation can be held responsible. Unfortunately, the test system did not support the pinning of individual OpenMP threads to specific cores so that we were unable to distinguish between inter- and intra-chip effects. Whether offset alignment or interpolation can alleviate the errors remains to be evaluated and also depends on the question to which extent the mapping of threads onto cores remains stable during the execution of longer programs.

2.6 Summary

In this chapter, we have evaluated different options for obtaining event timings when tracing parallel applications on cluster systems. Because the danger of perturbation complicates offset measurements in the middle of the run, linear offset interpolation between offset measurements at the beginning and the end of the run has been introduced as an established instrument used by tracing tools such as Scalasca for an initial correction of timestamps and as a yardstick to assess the appropriateness of timer technologies.

Since software clocks such as `MPI_Wtime()` or `gettimeofday()` often leverage network synchronization via NTP, hardware clocks such as IBM's time base register (TB) or Intel's timestamp counter register (TSC) have been identified as alternatives with at least approximately constant clock drifts. However, as a more detailed analysis revealed, even these alternatives suffer from drift deviations that may compromise the accuracy of linear offset interpolation - especially when the application runs longer than a few minutes. As a consequence, many traces of MPI applications spanning multiple SMP nodes of a cluster system will exhibit violations of the clock condition, potentially misrepresenting the logical

event order imposed by message semantics and therefore harming further analyses. Evidence of frequent violations in real codes has been presented.

Moreover, inaccuracies of timestamps within single SMP nodes in combination with the low latency of shared-memory synchronization in OpenMP may lead to infringements of semantics on some systems. As the experiments further indicate, smaller numbers of threads tend to be more easily affected than larger numbers – potentially due to lower OpenMP synchronization latencies when using only a few threads.

Summarizing the insights we have gained so far, we can state that linear offset interpolation is insufficient at least for message-passing and realistic parallel programs spanning more than one SMP node. As a consequence, the logical event order imposed by the semantics of the underlying communication substrate may be misrepresented. Lacking more appropriate timer technologies, we now look for alternatives in Chapter 3, where we review several approaches aiming at correcting such inconsistencies.

Chapter 3

Clock Synchronization

This thesis describes an approach for retroactively synchronizing timestamps in event traces of realistic parallel programs for the purpose of accurate program observation. Before discussing the underlying method of the proposed synchronization approach in Chapter 4, this chapter reviews work related to the topic of clock and timestamp synchronization.

3.1 Network-based Synchronization

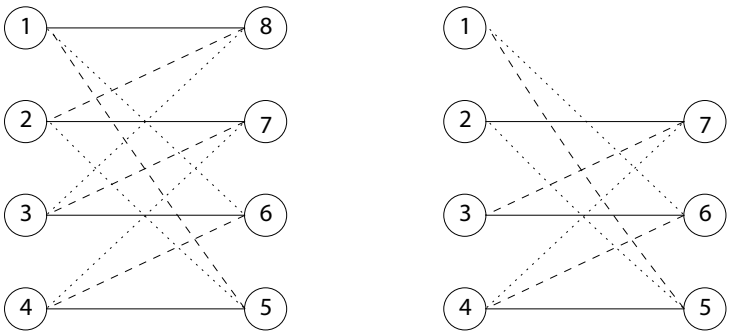
Network-based synchronization protocols aim at synchronizing distributed clocks before reading them. Clocks distributed across the network query the global time from reference clocks, which are often organized in a hierarchy of servers. For instance, NTP [68] uses widely accessible pre-synchronized primary 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. Jumps are avoided by changing the drift while leaving the actual time unmodified. Unfortunately, varying network latencies limit the accuracy of NTP to about one millisecond compared to a few microseconds required to satisfy the clock condition for message-passing applications running on clusters equipped with modern interconnect technology.

3.2 Offset Interpolation

Time differences among distributed clocks can be characterized in terms of their relative offset and drift, as discussed in Chapter 2. 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 [25]. After estimating the drift, the local time can be mapped onto the global (i.e., master) time via linear offset interpolation. Given that a detailed description of the linear offset interpolation technique can be found in Chapter 2, the account of linear offset interpolation is limited to the basics.

Offset values among participating clocks are measured either at program initialization [34] or at initialization and finalization, and are subsequently used as parameters of the linear correction function [51, 64]. So as not to perturb the program, offset measurements in between



(a) Even number of clocks. (b) Odd number of clocks.

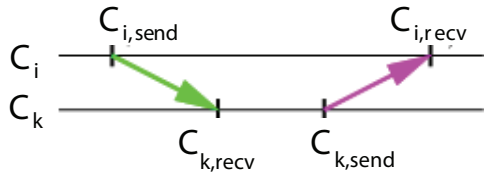
Figure 3.1: Communication scheme including edge-coloring for pair-wise offset measurements among even and odd numbers of clocks [29].

are usually avoided. While linear offset interpolation might prove satisfactory for "short" runs (or interpolation intervals), measurement errors and time-dependent drifts may create inaccuracies and clock condition violations during longer runs (see Chapter 2). Additionally, drift adjustments caused by NTP may impede linear interpolation, as they deliberately introduce non-constant drifts.

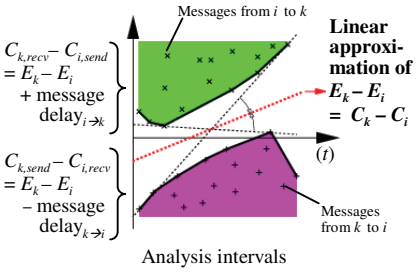
Doleschal et al. [29] propose periodic offset measurements during global synchronization operations while limiting the effort required in each step by resorting to indirect measurements across several hops. More precisely, clock offsets are periodically measured at global synchronization points while the target application is running. To reduce the measurement overhead, clock offsets are measured pair-wise using a communication scheme given as a color-edged graph. As illustrated in Figure 3.1 for even and odd numbers of clocks, this graph is a bipartite, regular graph containing a Hamilton cycle that allows all clock offsets to be determined – either directly or indirectly. For instance, in Figure 3.1(a) the offset between clock 1 and 2 can be determined by combining the offset between clock 1 and 8 and the offset between clock 8 and 2. Note that for odd numbers of clocks, a solution can be derived from the solution with $n + 1$ vertices by deleting one vertex and all adjacent edges. The effort of this communication scheme uses $\mathcal{O}(n\ log\ n)$ synchronization messages and requires $\mathcal{O}(log\ n)$ time for n clocks. After program termination, the offsets are used as parameters of a piece-wise linear interpolation function. Apparently, larger temporal distances between offset measurements decrease the measurement overhead but also the accuracy of the piece-wise linear interpolation.

3.3 Error Estimation

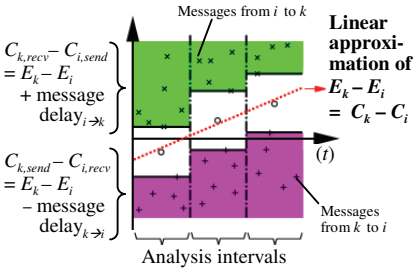
If linear interpolation alone turns out to be inadequate to achieve the desired level of accuracy on a specific cluster system, error estimation allows the retroactive correction of clock values in event traces after assessing synchronization errors among all distributed clock pairs.



(a) Time lines of two processes exchanging messages.



(b) Duda et al.



(c) Hofmann.

Figure 3.2: Algorithms that calculate the clock errors through the differences of message transfer times in both directions between two processes.

Difference functions among clock values are calculated from the difference between clock values of receive and send events (plus the minimum message latency). Then, 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. [33] to determine the smoothing function. Using a minimal spanning tree algorithm, Jezequel [58] adopted Duda's algorithm for arbitrary processor topologies. In addition, Hofmann [50] 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 3.2 shows the principles underlying Duda's and Hofmann's algorithms with two processes exchanging messages. Figure 3.2(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, messages arrows from process i to k are shown in green, whereas messages arrows from process k to i are shown in purple. In Figures 3.2(b) and 3.2(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 the 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 Figure 3.2(b), these time differences are enclosed by their convex hulls (i.e., green and purple encolored areas). In addition, the dotted lines show those lines with maximum and minimum slope possible between both convex hulls. Duda et al. calculate the approximation function as the interior bisector of the angle between both linear functions. As can be seen in Figure 3.2(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 [52] 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 [80]. Biberstein et al. [13] rewrote Hofmann and Hilgers' algorithm for use on the Cell BE architecture [21] 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 [4, 32] 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 [96].

3.4 Logical Synchronization

In contrast, logical synchronization uses "happened-before" relations among send and receive pairs to synchronize distributed clocks. Lamport introduced a discrete logical clock [62] with each clock being represented as 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. Lamport's discrete logical clock [62] can be used directly for monitoring [26]. Moreover, an algorithm to prevent the drift between the logical clocks has been proposed by Raynal [81].

Figure 3.3 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 communication and internal events (opposed to communication-related events). While events are depicted as small squares, messages are shown as arrows pointing in the direction of the communication. Given that the process-local clock value is represented as a software counter, 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 and the updated values are sent along with the message. Those values are used to calculate the new clock value at the receiver side. Actually, the clock value of the receive event is given

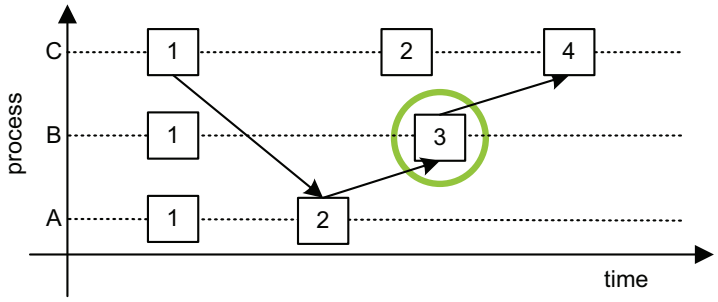


Figure 3.3: Lamport's discrete logical clock: The clock value of the green encircled event on process B is updated based on the maximum of both the incremented local clock value on process B and the incremented clock value of the sending event on process A.

as the maximum of the incremented local clock value and the incremented clock value of the corresponding send event. For instance, the green encircled event on process B (Figure 3.3) 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 sending event on process A is 3. Given that the maximum of both is 3, the new value is set to 3. Finally, Lamport's logical clock preserves the relative order of events, but does not account for the temporal distance between adjacent events. Therefore, Lamport's logical clock cannot be used for certain performance-analysis applications such as measuring wait states. Moreover, events happening at different wall-clock times may have the same logical clock value, as can be seen in Figure 3.3 for the second events of processes A and C.

As an enhancement of Lamport's discrete logical clock, Fidge [37, 38] and Mattern [65] 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 [28, 35] and, in a modified form, to distinguish in event traces between primary wait states and secondary ones that are merely caused by propagation. Furthermore, global events are introduced in [46], while in [78] spontaneous events (e.g., collisions on a network) are taken into account. Finally, limits of the logical clock and the vector clock are illustrated in [83].

As a further enhancement, the *controlled logical clock* (CLC) algorithm [79, 80] developed by Rolf Rabenseifner retroactively corrects clock condition violations in event traces of message-passing applications. The CLC algorithm operates on wall-clock time values and is therefore able to preserve interval lengths. Those are preserved by shifting events in time as much as needed. Similar to Lamport's discrete logical clock, the CLC algorithm restores the clock condition using happened-before relations derived from point-to-point message semantics.

If the clock condition is violated for a send-receive event pair, the receive event is moved forward in time. As the modification of individual timestamps might change the length of local intervals and even introduce new violations, the correction preserves the context of the modified event by collaterally moving affected events forward in time, compressing the local time axis in the future and carefully stretching the local time axis in the immediate past of the affected event. These adjustments are called “forward” and “backward amortization”, respectively. Note that the accuracy of the adjustment depends on the accuracy of the original timestamps, which can be increased through a pre-synchronization step using the linear offset interpolation technique discussed earlier.

3.5 Summary

In this chapter, approaches to synchronizing timestamps in event traces of parallel programs were reviewed. Focusing our description of related work on the topic of clock and timestamp synchronization, we have identified network based synchronization, offset interpolation, error estimation, and logical synchronization as common mechanisms.

Although all discussed techniques are suitable options to increase the fidelity of event timings, all are inappropriate for our purposes. Even though NTP can align processor clocks to a certain degree, the achieved precision of about one millisecond is too low to avoid violations of the logical event order as imposed by the underlying communication substrate. Furthermore, offset interpolation - especially the popular linear offset interpolation - and error estimation cannot handle arbitrary clock drifts and thus fail to guarantee consistent process timings at the desired level of accuracy. While logical synchronization guarantees the logical correctness of event timings, this technique modifies the actual time once the logical event order has been violated and may introduce jumps at synchronization points. In order to retain the length of intervals between local events, the CLC algorithm accounts for this limitation, making it a promising target for further studies.

However, the current CLC algorithm is limited by two factors. First, it covers only point-to-point event semantics, which makes it only partially suitable for realistic message-passing applications that perform both point-to-point and collective communication. This limitation also includes the non-observance of shared-memory clock conditions related to OpenMP constructs. Second, it is a serial algorithm designed for a single global trace file and cannot be efficiently applied to traces from large numbers of processes. Nevertheless, the CLC algorithm can form the basis of a more comprehensive and scalable method for retroactively synchronizing timestamps in event traces of parallel programs for the purpose of accurate program observation. However, before we discuss the extensions of the CLC algorithm necessary to satisfy the above-mentioned requirements, the details of the original CLC algorithm are described in Chapter 4.

Chapter 4

Controlled Logical Clock

Analyzing the performance of parallel programs, for example, by identifying wait states in event traces, requires measuring temporal displacements between concurrent events. In the absence of synchronized hardware clocks, linear interpolation techniques can already account for differences in offset and drift, assuming that the drift of an individual processor is not time dependent. However, inaccuracies and drifts varying in time may cause violations of the logical event order. The *controlled logical clock* (CLC) algorithm accounts for such violations in point-to-point communication by shifting message events in time as much as needed while trying to preserve the length of intervals between local events. In this chapter, the CLC algorithm is described as an extensible option for removing remaining inconsistencies in event traces postmortem.

4.1 Rationale

Given that this thesis focuses on a synchronization method for the use within Scalasca, the CLC algorithm is described it in terms of the Scalasca event model, which is similar to the Vampir event model [71] for which the algorithm was originally designed. As far as message-passing is concerned, the two models differ only in the way they express collective communication, which the original algorithm ignores anyway. Since the Scalasca event model has already been introduced in Chapter 1, this section only briefly recapitulates the basics needed to understand the algorithm.

The information Scalasca records for an individual event includes at least the timestamp, the location (e.g., the process) causing the event, and the event type. Depending on the type, additional information may be supplied. The event model distinguishes between programming-model independent events, such as entering and exiting code regions, and events related to MPI and OpenMP operations. MPI-related events include events representing point-to-point operations, such as sending and receiving messages, and events representing the completion of collective MPI operations. Event sequences recorded for typical MPI operations are given in Table 4.1. OpenMP-related events 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 record 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 OpenMP-related collective exit record indicates that the program leaves either a parallel or a barrier

Table 4.1: Event sequences recorded for typical MPI operations.

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

region. Furthermore, an lock-acquisition event record indicates that a lock variable is set, whereas a lock-release event record indicates that this variable is unset. Event sequences recorded for typical OpenMP constructs are given in Table 4.2.

As shown in Chapter 2, clock errors may cause both quantitative and qualitative effects. Quantitative effects may occur as a change of the length of intervals, whereas qualitative effects may occur as a change of the logical event order, which requires a message to be received only after it has been sent. In general, 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* [62], which is given in Equation 4.1. While the errors of single timestamps are hard to assess, violations of the clock condition can be easily detected and offer a toehold to increase the fidelity of inter-process timings.

$$\forall\, e, e' : e \rightarrow e' \Longrightarrow C(e) < C(e'). \tag{4.1}$$

The CLC algorithm [79, 80] is an enhancement of Lamport's logical clock [62] and requires timestamps with limited errors, which can be achieved through linear offset interpolation between program start and end. Specifically, the algorithm retroactively corrects clock condition violations in event traces of message-passing applications by shifting message events in time while trying to preserve the length of intervals between local events. In fact, the algorithm restores the clock condition using happened-before relations derived from point-to-point message semantics. Since messages need time to travel to their destination, we can reformulate the above condition, as given in Equation 4.2, with l_{min} being the minimum message latency.

$$\forall\, e, e' : e \rightarrow e' \Longrightarrow C(e) + l_{min} \leq C(e'). \tag{4.2}$$

Note that the hierarchical structure of many parallel systems allows the definition of multiple l_{min} per system, for example, depending on whether messages are exchanged within the same node or across different nodes. Usually, different latency values can be measured between (i) processors on the same node, (ii) processors on different nodes, and (iii) processors on distributed metahosts of a metacomputer.

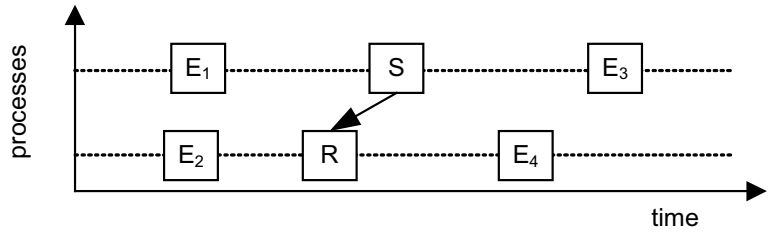
If the condition is violated for a send-receive event pair, the receive event is corrected (i.e., moved forward in time). To preserve the length of intervals between local events, events following or immediately preceding the corrected event are also adjusted. These adjustments are called "forward" and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

Table 4.2: Event sequences recorded for typical OpenMP constructs.

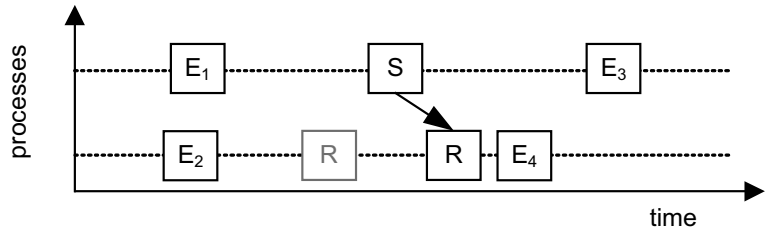
Region name	Event sequence
`omp_parallel`	(fork, enter, OpenMP collective exit, join) for the participating master thread
`omp_parallel`	(enter, OpenMP collective exit) for each participating worker thread
`omp_barrier`	(enter, OpenMP collective exit) for each participating thread
`omp_set_lock`	(enter, lock-acquisition, exit)
`omp_unset_lock`	(enter, lock-release, exit)
`omp_critical`	(lock-acquisition, enter, exit, lock-release)
`omp_atomic`	(enter, exit)

Figure 4.1 illustrates the different steps of the CLC algorithm using a simple example consisting of two processes exchanging a single message. The subfigures show the time lines of the two processes along with their send (S) or receive (R) event, each of them enclosed by two other events (E_i). Figure 4.1(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 4.1(b)). 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 4.1(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 4.1(d)). As can be seen in this example, the algorithm only moves events forward in time.

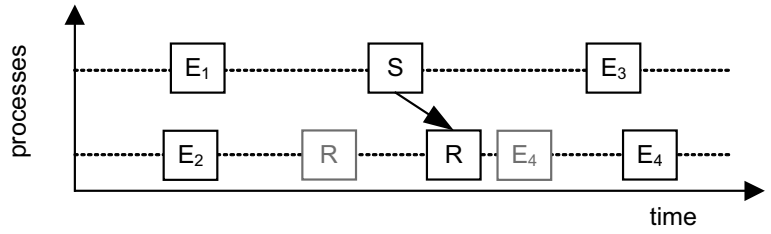
The logical clock scans the event trace for clock condition violations and applies the forward amortization to all events following a violated receive event. To prevent an increase of the overall time represented by the trace that may occur as a result of a domino-style propagation of forward amortizations, the algorithm applies scaling factors (i.e., control variables) to ensure that the overall error remains within predefined boundaries. The CLC algorithm always tries to advance all processor clocks to the fastest clock when correcting the non-linearity of the clocks. Given that the original timestamps may be logically wrong, this correction leads to logically correct timestamps with marginal local inaccuracies. As a result, timestamp differences between events on different processes normally become more accurate than the original ones because the clocks are advanced to a global clock represented by the fastest clock among all participating clocks at a time. In comparison, the above-mentioned algorithms of Duda, Hofmann, and colleagues align the timestamps with the average of the local clocks. However, for monitoring purpose this difference is not significant because it is in the range of the drift rates among local clocks (i.e., in the range of about $10^{-6} - 10^{-4}$). Combined with linear offset interpolation between program start and end, the expected differences are in the range of 10^{-8}.



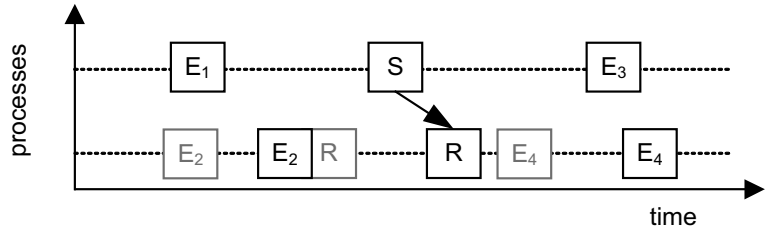
(a) Inconsistent event trace: Clock condition violation in point-to-point communication pair.



(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.

Figure 4.1: Backward and forward amortization in the controlled logical clock algorithm.

The backward amortization applies a linearly increasing correction to a limited amortization interval before a formerly violated receive event. However, in order to avoid new violations of the clock condition, the correction must not advance any send event located in this interval farther than the matching receive event (minus the minimum message latency). In such a case, linear correction is applied piecewise, advancing the send events as far as possible and calculating a different slope for each subinterval before, after, or between those sends [8, 80]. The sections below give a formal explanation of logical clock with forward and backward amortization.

4.2 Logical Clock with Forward Amortization

In the following, the symbol LC' is used to denote timestamps computed by the CLC algorithm. LC' is modeled with t as the wall-clock time and $T(t)$ as the synthetic global time to which the process clocks $C_i(t)$ $(i = 0, \ldots, n-1)$ will be synchronized. Moreover, let n be the number of processes, e_i^j the j^{th} event on process i, and

$$E = \{e_i^j | i = 0, \ldots, n-1 \wedge j = 0, \ldots, j_{\max}(i)\}$$

the set of all events in the trace. In addition, the set of matching send and receive pairs is defined in Equation 4.3. In this version of the algorithm, M only consists of the set of point-to-point messages.

$$M = \{(e_k^l, e_m^n) | e_k^l \ = \ \text{send event} \ \wedge e_m^n \ = \ \text{matching receive event}\}. \tag{4.3}$$

As discussed in Chapter 1, the send event always marks the beginning of a send operation, whereas a receive event marks the end of a receive operation. Furthermore, e_i^j is called an *internal* event if it is neither a send nor a receive event. Furthermore, δ_i is the minimal difference between two events on process i and $\mu_{k,i}$ is the minimum message delay of messages from process k to process i. Finally, γ_i^j is a control variable with $\gamma_i^j \in [0, 1]$. For each process i, LC_i' is now defined as

$$LC_i'(e_i^j) := \begin{cases} \max(LC_k'(e_k^l) + \mu_{k,i}, & \\ \quad LC_i'(e_i^{j-1}) + \delta_i, & \\ \quad LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), & \\ \quad C_i(t(e_i^j))) & \text{if } \exists e_k^l : (e_k^l, e_i^j) \in M \qquad (4.4) \\ \max(LC_i'(e_i^{j-1}) + \delta_i, & \\ \quad LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), & \\ \quad C_i(t(e_i^j))) & \text{otherwise.} \qquad (4.5) \end{cases}$$

As can be seen, the algorithm consists of two equations. Equation 4.4 adjusts the timestamps of receive events while Equation 4.5 adjusts timestamps of internal and send events. Note that

for each process, the terms $LC'_i(e_i^{j-1}) + \delta_i$ and $LC'_i(e_i^{j-1}) + \gamma_i^j(C_i(t(e_i^j)) - C_i(t(e_i^{j-1})))$ must be omitted for the first event $(j = 0)$.

The terms $C_i(t(e_i^j))$, $LC'_k(e_k^l) + \mu_{k,i}$, and $LC'_i(e_i^{j-1}) + \delta_i$ implement the logical clock, whereas the term $LC'_i(e_i^{j-1}) + \gamma_i^j(C_i(t(e_i^j)) - C_i(t(e_i^{j-1})))$ implements the forward amortization. More precisely, through the term

$$C_i(t(e_i^j))$$

in Equations 4.4 and 4.5, the algorithm ensures that a correction is only applied if the trace violates the clock condition. The new timestamps satisfy the clock condition because the term

$$LC'_k(e_k^l) + \mu_{k,i}$$

in Equation 4.4 guarantees that $LC'(e_i^j)$ is put forward compared to $C_i(t(e_i^j))$ if required in the case of a clock condition violation. To make sure that the clock does not stop after a clock condition violation, the term

$$LC'_i(e_i^{j-1}) + \gamma_i^j(C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))$$

in Equations 4.4 and 4.5 approximates the duration of the original communication after a clock condition violation and so implements the forward amortization. That is, the clock LC'_i for subsequent events of process i runs with the speed of C_i slowed down by the factor γ_i^j. As a consequence, the overall time represented by the trace is preserved while only marginally changing the length of intervals between local events following the clock condition violation.

Moreover, Rabenseifner has shown that γ_i^j with a constant value can cause LC' to be faster than the fastest clock among all process-local clocks C_i [80]. Cyclic changes of physical clock drifts may cause an avalanche effect that enlarges the value of clock corrections and propagates until the end. To avoid this effect, a control loop is used to find the optimal value of γ_i^j. The controller tries to limit the differences between LC' and T, that is, the controller estimates the output error indirectly because $T(t(e_i^j))$ is unknown. If $1 - \gamma$ is chosen smaller than the maximal drift differences, the controller will enlarge $1 - \gamma$ (e.g., to 1%) to guarantee that any propagation is bounded by this factor. To calculate γ_i^j for each event, the controller requires a global view of the event data. However, if γ_i^j is kept less than 1 minus the maximum drift of the processor clocks, a fixed $\gamma = 0.99$ or 0.999 is usually good enough to avoid an avalanche effect because physical clock drifts are normally much less than 10^{-4}. For subsequent events of the same process, the term

$$LC'_i(e_i^{j-1}) + \delta_i$$

in Equations 4.4 and 4.5 causes LC' to advance at least a small number of ticks δ_i if the physical clocks return the same clock value for different events or the controller has reduced γ_i^j to nearly zero. Rabenseifner described the control mechanism in more detail in [80].

4.3 Backward Amortization

Backward amortization is applied to smooth jump discontinuities caused by the jump Δt of the logical clock. The value of the jump $\Delta t(e_i^j)$ at a violated receive event e_i^j is shown in Equation 4.6 with $(e_k^l, e_i^j) \in M$.

$$\Delta t(e_i^j) = t_2(e_i^j) - t_1(e_i^j) \tag{4.6}$$

The respective definitions of $t_1(e_i^j)$ and $t_2(e_i^j)$ are shown in Equations 4.7 and 4.8 also with $(e_k^l, e_i^j) \in M$.

$$t_1(e_i^j) = \begin{cases} \max(LC_i'(e_i^{j-1}) + \delta_i, \\ \quad LC_i'(e_i^{j-1}) + \gamma_i^j(C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), \\ \quad C_i(t(e_i^j))) \end{cases} \tag{4.7}$$

$$t_2(e_i^j) = \begin{cases} \max(LC_k'(e_k^l) + \mu_{k,i}, \\ \quad LC_i'(e_i^{j-1}) + \delta_i, \\ \quad LC_i'(e_i^{j-1}) + \gamma_i^j(C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), \\ \quad C_i(t(e_i^j))) \end{cases} \tag{4.8}$$

The amortization of the jump Δt is achieved by slowly building up the ascension using a piecewise process-local linear correction in an amortization interval L_A of appropriate size before the violating receive event [80]. The compensation is realized by setting the timestamps forward. In Figure 4.2, the horizontal axis represents LC_i^b, which is equal to LC_i' (i.e., the state after forward amortization) but without the jump Δt at the corrected receive event r (shown on the right). The vertical axis shows offsets to LC_i^b after applying different stages of backward amortization. Naturally, the offset at r corresponds to the jump Δt. Note that the smaller the gradient of a clock in this figure, the better the correction and the smaller the perturbation of preceding events. Therefore, the ratio $\Delta t / L_A$ should be only a few percent. Clearly, adjacent clock condition violations cause a larger perturbation.

In order to avoid new violations of the clock condition, the correction must not advance the timestamps of send events farther than $LC_m' - \mu_{i,m}$ of the corresponding receive event e_m^n of a remote process m. These upper limits are shown as circled values above the locations of the send events. If a linear interpolation (see dash-dotted line in Figure 4.2) does not advance send events farther than their upper limits (here only at event s_1), it can be directly applied. However, if these limits are smaller than the dashed-dotted line (here at events s_2, s_3, s_4, and s_5), then a reduced piecewise linear interpolation function must be used (see the dotted line). In our example, the clock error rate is higher than the desired $\Delta t / L_A$ in all intervals. For each receive event with a jump, the backward amortization algorithm is applied independently. If there are additional receive events inside the amortization interval during such a calculation step, then these events can be treated like internal events, because advancing the timestamp of a receive event further cannot violate the clock condition.

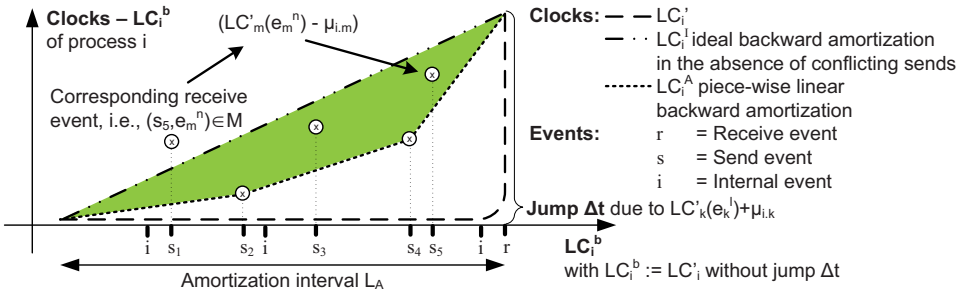


Figure 4.2: Piecewise process-local linear correction as backward amortization.

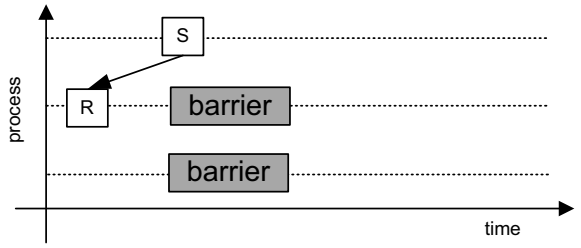
To preserve the length of local intervals and so to avoid overcorrection, the piecewise process-local linear correction must be calculated once the above-mentioned upper limits are smaller than the linear correction function. The piecewise process-local linear correction function (see the dotted line in Figure 4.2) is given as the lower border of the green-colored area which represents a convex area including all upper limits smaller than the linear correction function. This area is, in fact, described by two independent convex graphs. Whereas the linear correction function (see dash-dotted line) is the upper boundary of the marked area, the lower boundary (see dotted line) results in the piecewise process-local linear correction function.

4.4 Limitations

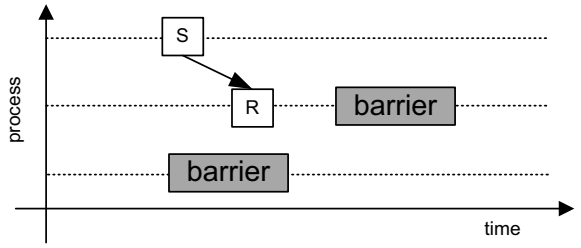
Although the original CLC algorithm removes remaining inconsistencies in event traces postmortem, it is limited by two factors. First, it only accounts for point-to-point event semantics and ignores others such as those imposed by collective message-passing or shared-memory operations. Second, it is a serial algorithm designed for a single global trace file and so it cannot be efficiently applied to traces from large numbers of processes.

The former limitation is twofold. Obviously, the current CLC algorithm does not account for direct violations of collective message-passing and shared-memory event semantics in the original trace (see Chapter 2). In fact, the algorithm neither restores nor preserves happened-before relations in collective message-passing and shared-memory operations, because the constituent events of such constructs are currently treated as internal events. Thus, the correction may introduce violations of the underlying event semantic even though the event semantic was not violated in the original trace. For this reason, the current CLC algorithm is not suitable for many parallel programs that perform not only point-to-point but also collective communication and shared-memory operations.

The potential implications of isolated corrections based on point-to-point message events for the semantics of collective message-passing and shared-memory events are exemplified in Figures 4.3 and 4.4. The two diagrams in Figure 4.3 show the time lines of three



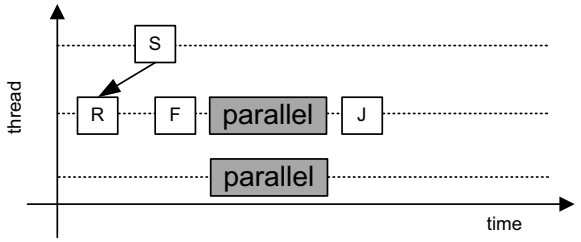
(a) Inconsistent point-to-point event semantics followed by consistent message-passing barrier semantics: The execution of the barrier by both processes does overlap.



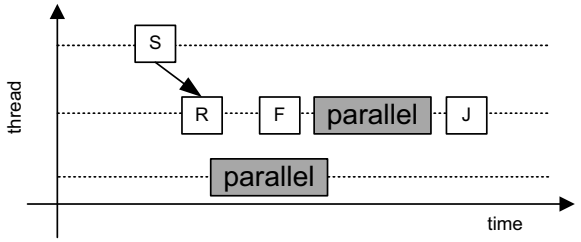
(b) Correction of inconsistent point-to-point event semantics may lead to inconsistent message-passing barrier semantics: Here, the execution of the barrier by both processes does no longer overlap.

Figure 4.3: Implications of corrections based on point-to-point event semantics for collective message-passing (MPI) event semantics.

processes. Two of these processes exchange a message via a send (S) and a receive (R) event. Subsequently, the second process executes together with the third process an MPI barrier. The violated point-to-point event order shown in Figure 4.3(a) is corrected in Figure 4.3(b). Here, the algorithm detects the clock condition violation in the point-to-point message exchange and immediately corrects this by moving the receive event (R) forward in time. In a next step, events following the exchange are also moved forward (as part of the forward amortization) without respecting the collective barrier event semantics. Given that no process is allowed to exit the barrier before the last process has entered it, the described correction introduces a clock condition violation. The next two diagrams in Figure 4.4 present a similar case that may occur in hybrid MPI/OpenMP programs. Shown is a message exchange analogous to Figure 4.3 followed by the execution of an OpenMP parallel region according to the POMP event model [69]. Here, the execution of an OpenMP parallel region by two threads is depicted, enclosed by a fork (F) and a join (J) event of the master thread. Whereas in Figure 4.4(a) the point-to-point event order is violated, the parallel regions appear clearly after the worker has been forked. However, while in Figure 4.4(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. More precisely, the algorithm detects and corrects the clock condition



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

Figure 4.4: Implications of restoring point-to-point event semantics for shared-memory (OpenMP) event semantics.

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.

Moreover, the current version is a serial algorithm designed for a single global trace file. In view of rapidly increasing parallelism, it is crucial that the timestamp synchronization scales to large numbers of application processes and so a parallel processing scheme would be desirable.

In order to address these limitations, Chapter 5 describes the algorithmic extensions aimed at restoring and preserving not only point-to-point but also collective message-passing and shared-memory event semantics. Chapter 6 presents the parallel version and its integration into the Scalasca trace-analysis framework.

Chapter 5

Algorithmic Extensions

The original CLC algorithm considers only point-to-point message semantics and, therefore, it is not suitable for parallel programs that also perform collective communication and shared-memory operations. However, these additional operations need to be taken into account in order to enable a more complete correction of timestamps in event traces of realistic parallel programs. To address the limitations of the original CLC algorithm, this chapter describes extensions to the original CLC algorithm aiming at restoring and preserving the logical event order in collective message-passing communication and shared-memory operations. These extensions allow the correction of realistic parallel programs by mapping collective message-passing and shared-memory event semantics onto point-to-point event semantics.

5.1 Basic Principle

This thesis focuses on a synchronization method for the use within Scalasca and thus the necessary extensions of the CLC algorithm are described in terms of the Scalasca event model. In this event model, a collective operation instance consists of multiple pairs of enter and MPI collective exit events (i.e., one pair for each participating process). Similar to a collective MPI operation, an OpenMP barrier is represented by multiple pairs of enter and OpenMP collective exit events. In addition, a parallel shared-memory region instance also consists of multiple pairs of enter and OpenMP collective exit events, enclosed by fork and join events only on the master thread to indicate that the team of threads was logically created or terminated. Moreover, OpenMP lock operations are also covered by the event model. A lock-acquisition event indicates that a lock variable – specified by a lock identifier – was acquired, whereas a lock-release event indicates that this lock was released.

In the following, a happened-before relation between two events is modeled as the exchange of a logical message between both events. Given that such happened-before relations exist among the constituent events of collective MPI and OpenMP operations, the basic idea behind the extension is now to map the above-mentioned events onto point-to-point communication events [9]. For this purpose, single collective message-passing and shared-memory operations are considered 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*, etc.) [8, 11]. Because the CLC algorithm synchronizes the timestamps of concurrent events using happened-before relations, the respective “receive” event is put forward in time whenever the matching

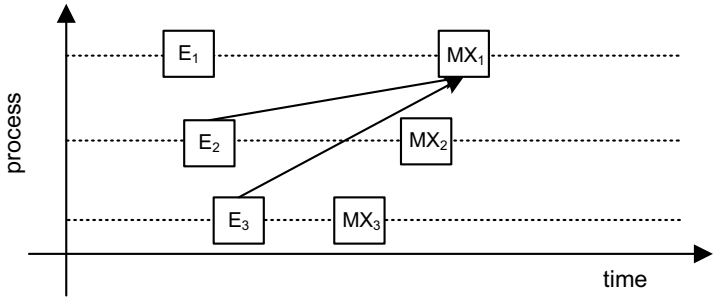
Table 5.1: Classification of MPI collective communication.

Category	MPI collective function
N-to-1	`MPI_Gather`, `MPI_Gatherv`
	`MPI_Reduce`
1-to-N	`MPI_Bcast`
	`MPI_Scatter`, `MPI_Scatterv`
N-to-N'	`MPI_Allgather`, `MPI_Allgatherv`
	`MPI_Alltoall`, `MPI_Alltoallv`, `MPI_Alltoallw`
	`MPI_Allrecuce`, `MPI_Reduce_scatter`
	`MPI_Barrier`
Special cases	`MPI_Scan`
	`MPI_Exscan`

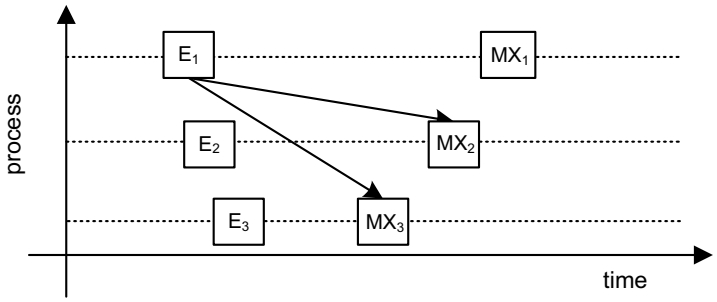
"send" event appears too late in the trace to satisfy the clock condition. In reference to the fact that this method is based on logical clocks, the send and receive event types assigned during this mapping are called the *logical event types* as opposed to the actual event types (e.g., enter, collective exit, etc.) specified in the event trace. The logical event type can usually be derived from the region name of the respective operation and the role a process or thread plays in the operation.

Common happened-before relations between events in collective message-passing operations are exemplified in Figure 5.1 using time-line views of three processes. Note that a happened-before relation between two events is shown by message arrows between these events. For the sake of simplicity and to keep the figures well-structured and understandable, happened-before relations on the same location are not shown. For instance, Figure 5.1(a) shows an MPI collective *N-to-1* operation, where one root process receives data from N other processes. Given that the root process is not allowed to exit the operation (see collective exit event MX_1) before it has received data from the last process to enter the operation (see enter events E_i), the clock condition must be observed between the enter events of all sending processes and the exit event of the receiving root process. In addition, Figures 5.1(b) and 5.1(c) present a *1-to-N* and a *N-to-N* situation, respectively. Whereas in the former figure only the root process sends messages to all other processes, in the latter figure every process sends a message to every other process. To fulfill the clock condition, none of the messages is allowed to flow backward in time.

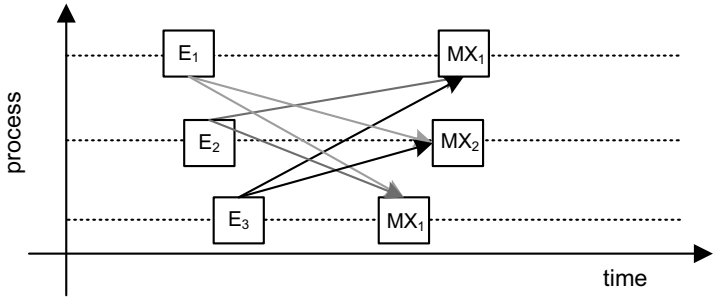
The CLC algorithm can use these happened-before relations in collective message-passing communication to synchronize event timestamps. Depending on the flavor of the collective operation, different enter and exit events are mapped onto logical send and receive events, respectively. Table 5.1 lists all MPI collective functions along with their category. Note that in *N-to-N'* situations, the number of processes contributing input data may be different from the number of processes receiving output data. For example, an `MPI_Allgather` operation can be used to collect data from all members of one group with the result appearing in all members of a different group. Moreover, the communication patterns of `MPI_Scan` and `MPI_Exscan` do not fit this taxonomy either. For instance, `MPI_Scan` returns in the receive buffer of process i the reduction of the values from the send buffers of processes $0, ..., i$ (inclusive).



(a) N-to-1: All processes send a logical message to the root process.

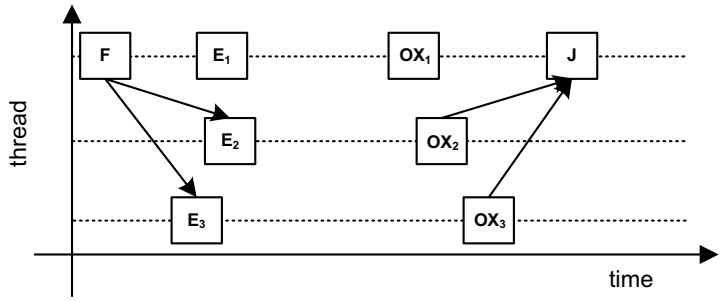


(b) 1-to-N: The root process sends logical messages to all other processes.

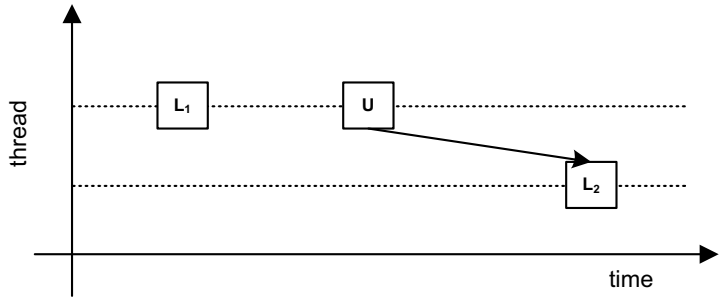


(c) N-to-N: Every process sends a logical message to every other process.

Figure 5.1: MPI collective operation event semantics.



(a) Team creation and termination: The master thread sends a logical message to create a team of threads and receives logical messages once the team has terminated.



(b) OpenMP lock sequence: A thread can only acquire a lock once the lock has been released.

Figure 5.2: OpenMP operation event semantics.

Although the MPI 2.1 standard [66] classifies the `MPI_Barrier` operation as a special case, because it does not move any data, for the purpose of this thesis, the happened-before relations found in this function allow us to classify this function as an *N-to-N'* situation with $N' = N$.

Recall that, even though clocks on an SMP node are usually well synchronized, the original CLC algorithm may violate shared-memory event semantics while correcting message-passing event semantics in a hybrid code. To enable a more complete correction of realistic parallel programs, the algorithm should preserve shared-memory event semantics and undo potential violations. As a consequence, happened-before relations need to be identified. Common happened-before relations in shared-memory event semantics are exemplified in Figure 5.2. Again, a happened-before relation between two events is shown by (logical) message arrows between these events, while happened-before relations on the same location remain hidden. None of the messages is allowed to flow backward in time and thus the CLC algorithm must use these happened-before relationships not only to restore and but also to retain the clock condition.

Table 5.2: Classification of OpenMP regions.

Category	OpenMP region
Team creation	begin of `parallel` region
Team termination	end of `parallel` region
Barrier	explicite `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

Figure 5.2(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) which subsequently enters (at enter events E_i) the parallel region. Such a situation is referred to as *team creation*. 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, which is referred to as *team termination*. OpenMP barrier constructs are similar to MPI barrier operations and therefore exhibit the same communication semantics that was discussed earlier. In addition, Figure 5.2(b) shows the time-line visualization of two threads acquiring and releasing a lock variable. First, one thread acquires (L_1) and releases (U) the lock variable. Second, the other thread locks (L_2) the same variable after the lock was released by the first thread. In such a situation, which is referred to as *locking*, two different happened-before relations exist. Of course, a thread is only allowed to release a lock after it was acquired ($L_1 \rightarrow U$). Additionally, the second thread can only acquire the lock once it is released by the first thread ($U \rightarrow L_2$). As the event model describes critical constructs with lock events, the described 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 when a thread has entered the critical region, whereas a lock-release event is recorded before the thread leaves the critical region. As discussed earlier, the event sequence of locks is only given by the timestamp of the respective events. Assuming that the thread-local clocks are synchronized, the timestamp could be used to determine the correct logical sequence of locks during the program run. However, on some systems this assumption cannot be maintained and so the timestamp is insufficient to determine the correct logical sequence of locks. In general, the algorithm can therefore not detect clock condition violations in lock sequences. Nonetheless, the algorithm can preserve the event order as found in the original trace. In addition, similar happened-before relations are also found in atomic and flush constructs, but the tracing library does not allow us to record events inside those regions. However, this is crucial to determine when a thread entered or left such a region. For instance, an atomic construct ensures that a specific storage location is updated atomically. Similar to

critical constructs, it would be necessary to record events when a thread executes the atomic construct. However, the execution of an atomic construct is restricted to statements that can be calculated atomically and so it is not possible to insert event tracing calls. Since we cannot identify happened-before relations in such regions, these constructs are currently ignored by the algorithm. Table 5.2 lists all OpenMP regions along with their category where we can identify happened-before relations.

Given that the original CLC algorithm neither restores nor preserves these happened-before relations, the algorithm must be extended to cover the above-mentioned event semantics. The following sections give a formal explanation of the extended version of the logical clock with forward and backward amortization for both cases: collective message-passing and shared-memory event semantics.

5.2 Collective Message-Passing Event Semantics

Given that the original CLC algorithm has been designed to correct clock condition violations only related to point-to-point communication, collective message-passing semantics are completely ignored. This section explains how the algorithm can be made suitable for realistic MPI applications that also use collective operations, starting with a discussion of the logical clock with forward amortization, followed by a discussion of backward amortization.

5.2.1 Logical Clock with Forward Amortization for Collectives

The logical clock scans the event trace for clock condition violations and applies the forward amortization to all events following a violated receive event. Below, different types of collective message-passing operations are reviewed to identify happened-before relationships based on the decomposition of collective operations into send and receive event pairs. Let S and R denote the set of logical send and receive events in a collective operation instance, respectively. As defined in Chapter 4, the set of matching send and receive pairs (i.e., messages) is called M. For each call to a collective operation, M is now enlarged by adding $S \times R$ with two exceptions, which are discussed later.

1-to-N:

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

N-to-1:

One root process receives its data from N processes. Examples are `MPI_Reduce()`, `MPI_Gather()`, and `MPI_Gatherv()`. R only contains the collective exit event of the root process as logical receive event. S contains the set of all the enter events, one for each processes in the communicator, with a data length greater zero as logical send events. Given that the root process is not allowed to exit the operation until the last process has entered it, the latest enter event is the relevant send event to fulfill the collective clock condition. Hence, if S contains more than one element, the term $LC'_k(e_k^l) + \mu_{k,i}$ in Equation 4.4 must be replaced by the maximum of

$$LC'_k(e_k^l) + \mu_{k,i}$$

over all $e_k^l \in S$. Given that M is enlarged by $S \times R$, Equation 4.4 must be rewritten as

$$LC'_i(e_i^j) := \begin{cases} \max(\ \max_{\{e_k^l | (e_k^l, e_i^j) \in M\}}(LC'_k(e_k^l) + \mu_{k,i}), \\ \qquad LC'_i(e_i^{j-1}) + \delta_i, \\ \qquad LC'_i(e_i^{j-1}) + \gamma_i^j(C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), \\ \qquad C_i(t(e_i^j))) \quad \text{if } \exists e_k^l : (e_k^l, e_i^j) \in M & (4.4') \\ \max(LC'_i(e_i^{j-1}) + \delta_i, \\ \qquad LC'_i(e_i^{j-1}) + \gamma_i^j(C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), \\ \qquad C_i(t(e_i^j))) \qquad \text{otherwise.} & (4.5) \end{cases}$$

N-to-N':

All processes of the communicator are at the same time sender and receiver. Examples here are `MPI_Allreduce()`, `MPI_Allgather()`, `MPI_Alltoall()`, and `MPI_Barrier()` with N'=N, and the variable length operations `MPI_Alltoallv()`, `MPI_Alltoallw()`, `MPI_Allgatherv()`, and `MPI_Reduce_scatter()` with potentially $N' \neq N$. S and R are defined by all those enter and collective exit events whose processes contribute input data or receive output data, respectively. To identify happened-before relations in the mentioned variable length operations, detailed information is needed on which process sent data where and from where a process received data. However, space requirements only allow us to record the aggregate amount of data sent and received for these routines. These aggregated data allow only the identification of logical send and receive events in situations where (i) all processes must have the same amount of input and/or output data or (ii) only one process either sends or receives data to or from other processes. As a consequence, the above-mentioned `MPI_Alltoallv()` and `MPI_Alltoallw()` operations cannot be synchronized with the event trace data currently available. Finally, for a call to `MPI_Barrier()`, all processes in the communicator contribute to S and R.

Special Cases:

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

$$\{(e_k^l, e_i^j) \mid k = 0, \ldots, N\text{–}1,\ i = 0, \ldots, k\text{–}x\}$$

with e_k^l referring to the enter event and e_i^j to the collective exit event of such a collective operation instance, and with $x = 0$ for `MPI_Scan()` and $x = 1$ for `MPI_Exscan()`.

Regardless of the collective operation type, it is important to determine $S \times R$ in Equation 4.4' efficiently. As described in Chapter 6, the parallelized version of the CLC algorithm achieves this by taking advantage of the way collectives are usually implemented, typically reducing the effort to $\mathcal{O}(\log N)$.

5.2.2 Backward Amortization for Collectives

The backward amortization applies a linearly increasing correction to a limited amortization interval before a formerly violated receive event. To preserve the clock condition, the new timestamp of a send event in the amortization interval still has to be smaller than or equal to the corresponding receive event timestamp minus the minimal message delay. Hence, the corresponding receive event timestamp needs to be determined for a send event. If there are multiple corresponding receive events for a send event, the backward amortization is not allowed to advance the send event farther than the earliest corresponding receive event. Depending on the type of operation, the earliest logical receive event must therefore be determined for each send event in the amortization interval.

More precisely, to extend the backward amortization algorithm for collective routines, the upper bounds for the send events (see Figure 4.2) must be adapted to collective events: If e_i^{j-m} is the send event of a collective routine, the upper bound for the piecewise linear interpolation at e_i^{j-m} is defined as

$$\min_{e_k^l \in R} LC'_k(e_k^l) - \mu_{i,k}$$

with R being the set of receive events defined in Section 5.2.1.

5.3 Shared-Memory Event Semantics

So far, the original CLC algorithm was extended to support the synchronization of collective message-passing event timestamps. Nonetheless, the algorithm does not yet account for violations of shared-memory event semantics in the original trace. Hence, the algorithm is currently not suitable for shared-memory application and hybrid applications that use message-passing and shared-memory parallelism in combination. Discussing the logical clock with forward amortization followed by backward amortization, this section explains how the algorithm can identify happened-before relations in shared-memory event semantics in order to synchronize timestamps in event traces of OpenMP and (hybrid) MPI/OpenMP applications. Again, the necessary enhancements are described in terms of the Scalasca

event model by mapping shared-memory events onto point-to-point communication events. Since the mapping depends on the information the event model provides, it is limited by the event model's constraints. As tracing of OpenMP ordered, task, and taskwait sections is not supported within Scalasca, it does also not account for OpenMP nested and task parallelism as well as dynamically changing thread counts. As discussed earlier, atomic and flush constructs are currently ignored because the tracing library does not support the observation of happened-before relations in these constructs. Furthermore, the algorithmic extensions do not cover shared-memory event semantics imposed by cluster-wide OpenMP implementations (e.g., Intel Cluster OpenMP [24, 49]), because such implementations may use additional communication introducing further event semantics which are currently ignored by the algorithm.

5.3.1 Logical Clock with Forward Amortization

In order to identify happened-before relationships derived from shared-memory event semantics, different types of shared-memory constructs are examined. In the following, a happened-before relation between two events is modeled with the exchange of a logical message between these events. Again, S and R denote the set of logical send and receive events in a shared-memory construct instance, respectively. For each such instance, the set of all send-receive pairs M is enlarged by adding $S \times R$.

Team creation:

Such a situation occurs when the master creates a team of threads. The master thread sends a logical message to N worker threads. S only contains the fork event of the master thread as logical send event, whereas R contains all the enter events of the corresponding parallel region, one from each thread in the team, as logical receive events.

Team termination:

Such a situation occurs when the master thread terminates the team of threads. The master thread receives logical messages from N worker threads. R only contains the join event of the master thread as logical receive event. S is the set of all OpenMP collective exit events of the corresponding parallel region, one from each thread of the team. Given that the master thread is not allowed to terminate the team of threads until the last thread has exited the parallel region, the latest OpenMP collective exit event is the relevant send event to fulfill the shared-memory clock condition. As already discussed in Section 5.2.1, if S contains more than one element, the term $LC'_k(e^l_k) + \mu_{k,i}$ in Equation 4.4 must be replaced by the maximum of $LC'_k(e^l_k) + \mu_{k,i}$ over all $e^l_k \in S$ and thus Equation 4.4' must be used accordingly.

Barrier:

All threads in a team of threads are at the same time sender and receiver. This situation shows up in explicit and implicit OpenMP barrier construct. S and R are defined by all those enter and OpenMP collective exit events.

Locking:

For OpenMP lock semantics, the set of messages added to M can also be expressed as the Cartesian product $S \times R$. At a time, a lock variable can only be acquired by one thread. Once a look is released, the releasing thread sends a logical "message" to the next thread acquiring the lock. R contains the lock-acquisition event as logical receive event, whereas S only contains the lock-release event of the thread releasing the lock as logical send event. At program start, no lock is assigned to a thread and so the first thread acquiring a lock does not need to wait for a preceeding unlock event because all locks are unlocked at program start. In order to satisfy the happened-before relations for these lock-acquisition events, S needs to be enlarged by an "theoretical" lock-release event (e.g., first event in the trace) matching the first lock-acquisition event in the trace.

As discussed in Chapter 1, the sequence of these events is only given by the timestamp of the respective events and so the algorithm cannot detect clock condition violations in the original trace. More precisely, event attributes are necessary to accurately decide which thread acquires a lock after another thread has released it and to identify sending and receiving thread pairs. However, the current event model does not provide such attributes (e.g., sequence count) and therefore the CLC algorithm can only preserve the event order as found in the original trace. Hence, the original order of lock events is determined once at program start and subsequently used when the roles of sender and receiver are determined. As the event model describes critical constructs with lock events, this technique also accounts for critical regions. Given that the same unspecified name or a user-defined name is used to identify a critical region, the event model provides lock identifiers referring to the name of a critical region.

5.3.2 Backward Amortization

To extend the backward amortization algorithm for shared-memory constructs, the upper bounds for the send events (see Figure 4.2) must be adjusted to shared-memory events: If e_i^{j-m} is the send event of a shared-memory construct, the upper bound for the piecewise linear interpolation at e_i^{j-m} is defined as

$$\min_{e_k^l \in R} LC'_k(e_k^l) - \mu_{i,k}$$

with R being the set of receive events defined in Section 5.3.1.

5.4 Summary

This chapter described algorithmic extensions to the original CLC algorithm aiming at restoring and preserving the logical event order in collective message-passing communication and shared-memory operations. Because the algorithm synchronizes the timestamps of concurrent events based on happened-before relations, they are determined for these operations. A happened-before relation between two events is modeled as the exchange of a logical message between the two events. The basic idea behind the extension is to consider collective and shared-memory operations as being composed of multiple logical point-to-point messages, taking the semantics of the different flavors of operations into account. 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, etc.) specified in the event trace.

The extensions cover the different flavors of message-passing communication and shared-memory operations. The extensions to identify happened-before relations in collective message-passing event semantics allow the correction of clock condition violations found in *1-to-N*, *N-to-1*, *N-to-N*', and `MPI_Scan`/`MPI_Exscan` communication patterns. Moreover, even though clocks on a SMP node are usually well synchronized, the original CLC algorithm may violate shared-memory event semantics while restoring message-passing event semantics in event traces of hybrid codes. The extended algorithm takes happened-before relations during *team creation*, *team termination*, *barrier* synchronization, and *locking* sequences into account, enabling a more complete correction of realistic parallel programs that use MPI and OpenMP in combination, thereby avoiding the collateral violation of shared-memory event semantics.

Chapter 6

Parallel Synchronization

The extended CLC algorithm, as described in Chapter 5, accounts for clock condition violations not only in point-to-point and collective communication but also in shared-memory operations. However as a sequential algorithm, the current version of the CLC algorithm is not able to synchronize timestamps in event traces of realistic parallel applications running on large processor counts. This chapter presents a parallel implementation of the extended CLC algorithm and its integration into the Scalasca trace-analysis framework. After reviewing Scalasca's parallel trace-analysis mechanism, it describes the parallel implementation for synchronizing timestamps in event traces of MPI applications. In addition, it introduces the necessary extensions of the parallel implementation for synchronizing timestamps in event traces of applications that use MPI and OpenMP in combination. Finally, to employ the extended and parallelized CLC algorithm in computational grids, this chapter also defines the infrastructure to accurately measure clock offsets in distributed environments with hierarchical networks.

6.1 Parallel Trace Analysis

Given that this thesis focuses on a parallel timestamp synchronization method for the use within Scalasca, this section reviews Scalasca's parallel trace-analysis methodology. This methodology is based on the idea of replaying the target application's communication during the trace analysis. Therefore, this section presents a detailed description of Scalasca's replay-based trace-analysis scheme and describes how it is used to identify performance bottlenecks. A brief overview of Scalasca can be found in Chapter 1.

6.1.1 Replay-Based Trace Analysis

To accomplish Scalasca's pattern search in a scalable way, both distributed memory and parallel processing capabilities available on the target system must be exploited. Instead of sequentially processing a single global trace file, Scalasca implements a scalable trace-analysis approach [45] by processing separate local trace files in parallel and *replaying* the original communication on as many CPUs as were used to execute the target application itself. The central idea behind the replay-based analysis is to reenact the target application's communication based on the trace information so that each communication operation can be analyzed

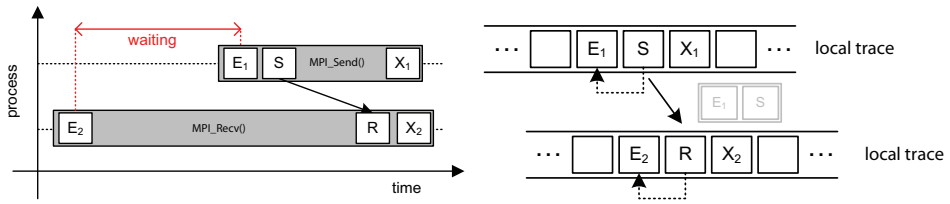
using an operation of similar type. For example, to analyze a message transfer in point-to-point mode, the required event data are exchanged using a single point-to-point operation. Since trace processing capabilities (i.e., processors and memory) grow proportionally with the number of processors, this approach guarantees good scalability at very large scales.

To maintain the efficiency of the trace-analysis process as the number of application processes increases, Scalasca's architecture follows a parallel trace-access model which is provided as a separate abstraction layer between the parallel pattern search and the raw trace data stored on disk [43]. Implemented as a C++ class library called PEARL, this layer offers random access to individual events as well as abstractions that help identify matching events, which is an important prerequisite for the pattern search. The main usage model of the trace-access library assumes a one-to-one mapping between analysis and target-application processes. That is, for every process of the target application an analysis process is created that is exclusively responsible for its trace data. For traces of programs that use the combination of MPI and OpenMP, the PEARL usage model assumes multiple processes and a fixed number of threads assigned to each process. Therefore, the pattern analysis becomes a parallel program having as many processes and threads per process as the target application that generated the trace data. Note that the current usage model is restricted in that it only supports MPI calls on the master thread (i.e., MPI funneled mode). Keeping the entire event trace in main memory during analysis thereby enables performance-transparent random access to individual events. Data exchange among analysis processes or threads is then accomplished via MPI communication or shared-memory variables.

Higher-level abstractions offered by PEARL include the context in which an event occurs, such as the call path or communication peers [91]. While special event attributes store local context information, remote event abstractions in combination with mechanisms for exchanging event data among analysis processes allow the tracking of interactions between concurrent events. The actual matching of communication events is performed by exploiting message event semantics during the parallel communication replay of the event trace. That is, whenever an analysis process recognizes events related to communication or synchronization, it engages in an operation of a similar type with its corresponding communication peer. This is discussed in more detail in the next section.

6.1.2 Parallel Pattern Search

Scalasca's trace analyzer uses the infrastructure offered by the trace-access library to traverse the local event traces in parallel from beginning to end while exchanging information at synchronization points of the target application. This information is used to locate patterns of inefficient behavior, to classify detected instances, and to quantify associated waiting times separately for every call path and process. A pattern is typically composed of multiple potentially concurrent constituent events with certain constraints regarding their relative order. The associated waiting time, which is called the *severity* of the pattern, is usually calculated as the temporal difference between selected constituents. As an example for the parallel detection of inefficient point-to-point communication, consider the so-called *Late Sender* pattern (see Figure 6.1(a)). Here, a receive operation is entered by one process before the corresponding



(a) Late Sender pattern. (b) Parallel detection.

Figure 6.1: Searching for the Late Sender pattern in parallel. The situation described by this pattern is shown on the left in a time-line diagram. How the different events are accessed and combined to verify the occurrence of the pattern is shown on the right.

send operation has been started by the other. The time lost waiting due to this situation is the time difference between the enter events of the two MPI function instances that precede the corresponding send and receive events. During the parallel replay (see Figure 6.1(b)), each analysis process traverses its local event trace data. Once a processes encounters a communication event, the pattern search is triggered by the communication events on both sides. Whenever an analysis process finds a send event, a message containing this event as well as the associated enter event is sent to the process representing the receiver using non-blocking point-to-point communication. When the receiver reaches the corresponding receive event, this message is received. Together with the local receive and enter event, a Late Sender situation can be detected by comparing the timestamps of the two enter events and calculating the time spent waiting for the sender.

After the trace analysis has been completed, a trace-analysis report is written. The trace-analysis report includes metrics, such as time, visit counts or message statistics, and also accounts for the times lost in different wait states. The report is stored as a three-dimensional array with the dimensions metric, call path, and system resource (e.g., process or thread). The user can interactively explore the report in the graphical profile browser shown in Figure 6.2. The tree in the left window pane displays patterns of inefficient performance behavior arranged in a specialization hierarchy. The numbers left of the pattern names indicate the total execution time penalty in percent. In addition, the color of the small square provides a visual clue of the percentage to quickly guide the user to the most severe performance problems. The middle window pane shows the distribution of the selected pattern's severity across the call tree. Finally, the right window pane shows the distribution of the pattern's severity at the selected call path across the application topology. Besides the application topology, the machine topology and the system hierarchy including machines, nodes, processes, and threads can be displayed.

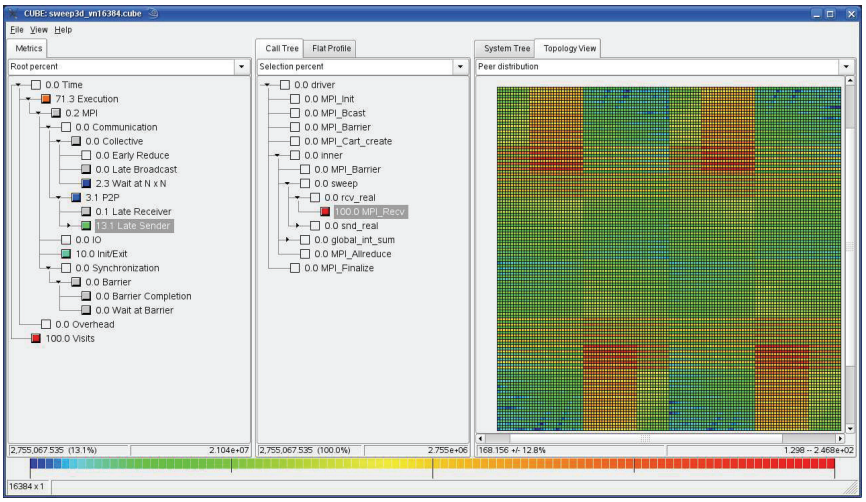


Figure 6.2: Trace-analysis report visualization and exploration in Scalasca.

6.2 Integration with Scalasca

Similar to the parallel pattern search performed by Scalasca, the CLC algorithm requires the comparison of events involved in the same communication operation, which makes it a suitable candidate for the same parallelization strategy. Instead of sequentially processing a single global trace file, the timestamp synchronization traverses separate local event traces in parallel by *replaying* the original communication on as many CPUs as were used to execute the target application itself. During the replay, sending and receiving processes exchange relevant information needed to analyze happened-before relations of the respective communication operation being replayed. This parallel version of the CLC algorithm is divided into two replay phases: a forward and backward phase. While the forward phase comprises the logical clock with the actual forward amortization, the backward phase comprises the backward amortization, which is only needed if clock condition violations appear during the forward phase [7, 10].

The CLC algorithm is implemented on top of PEARL and so the implementation is a parallel program having as many processes as the target application that generated the trace data, resulting in a one-to-one mapping between target application processes and timestamp synchronization processes. All synchronization processes read the trace data of "their" application process into main memory and traverse the traces in parallel while exchanging information at synchronization points. As already described in Section 6.1, the PEARL usage model assumes multiple processes and a fixed number of threads assigned to each process for traces of programs that use MPI and OpenMP in combination. In this case, each timestamp synchronization thread is responsible for processing the trace data of its application thread.

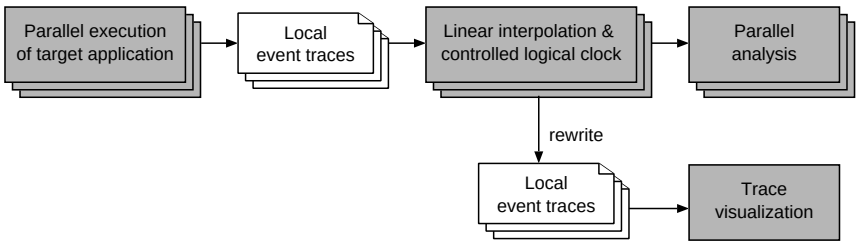


Figure 6.3: Parallel trace-analysis process. Gray rectangles denote programs and white rectangles with the upper right corner turned down denote files. Stacked symbols denote multiple instances of programs or files running or being processed in parallel.

As illustrated in Figure 6.3, the parallel CLC algorithm is applied after the traces have been loaded and before Scalasca's wait state analysis takes place. To increase the fidelity of the CLC outcome, the timestamps first undergo a pre-synchronization step. This step performs linear offset interpolation based on offset measurements taken during initialization and finalization of the target application (see Chapter 2). Once the offset values are known to each analysis process, the interpolation operation is performed locally and does not require any further communication. As an alternative to the native Scalasca wait state analysis, the traces can also be rewritten with modified timestamps, converted, and visualized using the Vampir timeline browser. In the following, Sections 6.3 and 6.4 present the parallel implementation of the extended logical clock with forward and backward amortization focusing on the synchronization of timestamps in event traces of MPI applications. Moreover, Section 6.5 presents the necessary extensions of the parallel implementation to enable the synchronization of timestamps in event traces of applications that use MPI and OpenMP in combination.

6.3 Logical Clock with Forward Amortization

The logical clock scans the process-local event trace in parallel for clock condition violations and immediately corrects these inconsistencies. The necessary remote event data are obtained by replaying the original communication using PEARL's parallel replay mechanism, which has already been described in Section 6.1. As the modification of individual timestamps might change the length of local intervals, the algorithm takes the context of the modified event into account by moving events forward but carefully compressing the local time axis starting at the affected event.

More precisely, each process i traverses its local events e_i^j from $j = 1, 2, ...$ and calculates the new timestamp $LC'_i(e_i^j)$ according to Equation 4.4, 4.4', and 4.5. Due to this iterative procedure, a process has access to the event e_i^{j-1}, which is already processed, and its new timestamp $LC'_i(e_i^{j-1})$ as required by Equation 4.4, 4.4', and 4.5. In order to access $LC'_k(e_k^l)$ on the sender side, as needed by Equation 4.4 and 4.4', the newly calculated timestamp $LC'_k(e_k^l)$ must be transmitted to process i. During the forward phase, the communication

Table 6.1: Timestamps exchanged among communication peers during forward replay for different message-passing communication types.

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

replay therefore proceeds in the same direction as the original communication while the target application was running. For every pair of logical send and receive events, the sending process sends the timestamp of the send event to the receiving process, which compares it to the timestamp of the matching receive event (minus the minimum message latency) and, if necessary, applies the timestamp correction expressed in Equations 4.4 and 4.4'. Recall that, in addition to actual send and receive events, events pertaining to entering or leaving collective message-passing operations may be classified as logical send or receive events for the purpose of the algorithm. In this case, the logical event type (logical send, logical receive, or internal event) is derived from the region name of the respective operation (e.g., `MPI_Barrier`, `MPI_Reduce`) and the role (e.g., root or master process) a particular process plays in the operation [8].

In its treatment of events the algorithm distinguishes between (logical) send or receive events and *internal* events that neither send nor receive any kind of message. A different action is performed for each of the three types. Since the correction of an internal event does not require any extra communication, the timestamp adjustment is immediately applied. A send event is adjusted locally and the new timestamp is sent via forward replay to the receiving process. On the receiver side, the order of these two steps is reversed. The adjusted send timestamp must be obtained from the sender before the correction can be performed. Finally, the receiver saves detected clock condition violations temporarily along with the associated timestamp correction (i.e., the jump Δt according to Equation 4.6) so that this information can be reused during the backward amortization phase.

While the direction of the inter-process exchange of timestamps is determined by the (logical) type of an event (i.e., send or receive), the actual communication operation invoked to accomplish the transfer depends on the operation originally used by the target application. For this purpose, collective message-passing communication is classified according to the number of peers involved on either side: *1-to-N*, *N-to-1*, *N-to-N'*, and two special cases as described in Chapter 5. The corresponding operation used during the replay depends on the respective class to which the operation belongs. Table 6.1 lists the MPI operations used during the replay along with the events which will have their timestamps exchanged. In brief, point-to-point operations are replayed using point-to-point communication, while collective operations are replayed using different flavors of collective communication. As discussed in Chapter 5, *N-to-N'* collective operations transferring a different amount of data per process

such as `MPI_Alltoallv` are currently ignored by the algorithm. To identify happened-before relations in the mentioned variable length operations, detailed information is needed on which process sent data where and from where a process received data. However, space requirements only allow us to record the aggregate amount of data sent and received for these routines. Although the extended version of the algorithm only needs information about the respective event semantics (e.g., root sends to all other processes), the accuracy of the model could be improved if the MPI-internal messaging inside collective operations was exposed using interfaces such as PERUSE [1]. In this case, the decomposition into (additional) send and receive events would be naturally given.

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. Note that in grid environments, the algorithm also uses an inter-machine latency to account for wide-area communications. Given that the parallel calculation of the maximum over all corresponding send events via

$$\max_{\{e_k^l \mid (e_k^l, e_i^j) \in M\}} (LC'_k(e_k^l) + \mu_{k,i})$$

in Equation 4.4' only requires the timestamps and the machine, node, and process identifiers to be exchanged to know which of the latency values must be used.

As mentioned earlier, the CLC algorithm uses so-called control variables. The control variable $\gamma_i^j \in [0,1]$ for e_i^j (the j^{th} event on process i) is a scaling factor that is applied to interval expressions when calculating the new timestamp for e_i^j. This fulfills the purpose of avoiding an avalanche-like propagation of corrections [8]. To determine the exact value for γ_i^j, however, a global view of the trace data is needed, which is too expensive to establish in Scalasca's parallel scheme, as global communication would be required for every single event. Instead, a suitable global γ is approximated by performing multiple passes of forward replay through the trace data until the maximum error across all processes is below a predefined threshold. During the first pass through the trace, the algorithm uses $\gamma = const < 1 - \epsilon$; for subsequent passes a

$$\gamma_{p+1} < \gamma_p$$

is used. More precisely, the current implementation uses a fixed $\gamma_0 = 1 - 10^{-5}$ in the first pass, while for subsequent passes γ_p is calculated with $\gamma_p = 1 - 10^{-5+p}$. In practice, however, more than one pass was never needed, as the experiments in Chapter 7 demonstrate.

6.4 Backward Amortization

The purpose of the backward amortization phase is to smooth jump discontinuities introduced during the forward amortization by slowly building up the ascension to the jump. This is achieved by applying a process-local linear correction to the interval immediately preceding the jump. However, in order to preserve the clock condition, the algorithm must not advance

Table 6.2: Timestamps exchanged among communication peers during backward replay for different message-passing communication types.

Type of operation	Timestamp exchanged	MPI function
P2P	receive	`MPI_Send()`
1-to-N	min(all exits)	`MPI_Reduce()`
N-to-1	root exit	`MPI_Bcast()`
N-to-N'	min(all exits)	`MPI_Allreduce()`
`MPI_Scan()`	min(some exits)	`MPI_Scan()`
`MPI_Exscan()`	min(some exits)	`MPI_Exscan()`

the timestamp of any send event located in this interval farther than that of the matching receive event (minus the minimum message latency), leading to the piecewise linear interpolation mentioned earlier. That is, remote event data are exchanged during the backward replay phase and subsequently used to calculate the (piece-wise) linear correction function for each violating receive event detected during forward amortization.

6.4.1 Backward Replay

The backward replay is needed to determine the matching receive event timestamp for any send event located in the amortization interval. While replaying the communication backward, the algorithm stores the timestamp of the matching receive event after forward amortization with each logical send event. With this information available, an appropriate piecewise linear interpolation function can be calculated for the amortization interval behind every receive event shifted during the forward replay due to a clock condition violation.

During the backward amortization the roles of sender and receiver are reversed: the timestamp of a logical receive event must be made available to the process of the matching send event. Table 6.2 shows the operations used during the backward replay along with the events which will have their timestamps exchanged. The backward amortization must be performed as a backward replay starting at the end of the trace with communication proceeding in backward direction to avoid the danger of deadlocks. For instance, let us consider the time lines of two processes exchanging messages as shown in Figure 6.4. In this situation, if the backward replay started at the begin of the trace with the roles of sender and receiver being reversed, as mentioned above, a deadlock would occur because each process would wait for a message from the other process. Obviously, this deadlock can be avoided if the backward replay proceeds from the end to the begin of the trace.

Given that most MPI implementations use binomial tree algorithms to perform their collective operations, the forward as well as the backward replay usually have a communication complexity for replaying collectives of $\mathcal{O}(\log N)$. Moreover, the stepwise parallel replay during the backward amortization phase could, in theory, be replaced by a single collective operation per communicator for the entire trace, but this would impose impractical memory requirements.

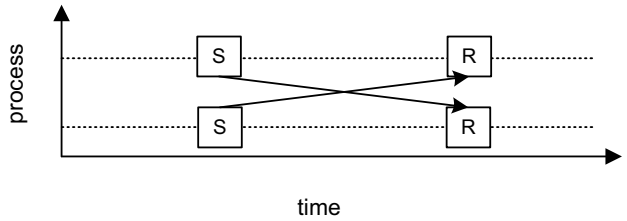


Figure 6.4: Time lines of two processes exchanging messages.

6.4.2 Piece-Wise Correction

After the backward replay, logical receive event timestamps are locally available on the logical sender side. Thus, the piece-wise correction function can be calculated. This section describes the algorithm to determine this correction function starting with an explanation of the symbols used, and followed by the actual algorithm design. The latter is additionally illustrated in Figure 6.5.

For the sake of simplicity, timestamps $LC'_i(e_i^j)$ after forward amortization are in the following denoted as t. In addition, δ always denotes timestamp differences. For a send event, δe denotes the maximum backward amortization allowed, whereas for a receive event, δe denotes the jump applied during forward correction. More precisely, the maximum backward amortization allowed for a send event is called δs_i and is given as the time difference between the timestamp of the i-th send event in the amortization interval and the corresponding forward-amortized receive event minus the minimum latency l_{min}. Further, the jump at a formerly violated receive event applied during forward correction is expressed by δr, which does not include any implicit jumps applied during the forward amortization of earlier events. As can be seen in Figure 6.5, the symbol r^* denotes the timestamp of a formerly violated receive event without forward correction and so its timestamp r after forward correction can be expressed as

$$r = r^* + \delta r.$$

The amortization interval is given as $[t_l, t_r)$, to which the correction is applied. Note that t_l denotes the left corner whereas t_r denotes the right corner of the amortization interval. In general, the (piece-wise) linear correction function is characterized by multiple linear functions $g_i(t)$ and a single linear function $g(t)$ is characterized by the slope m and the constant c as shown below

$$g(t) = m * t + c.$$

According to the CLC algorithm, the correction function represents offsets to the timestamps after forward amortization and so the new value of a timestamp t is expressed as

$$t_{new} := t + g_i(t).$$

The algorithm proceeds from the beginning to the end of the local event trace, because in this way no special treatment is required for receive events occurring inside a backward

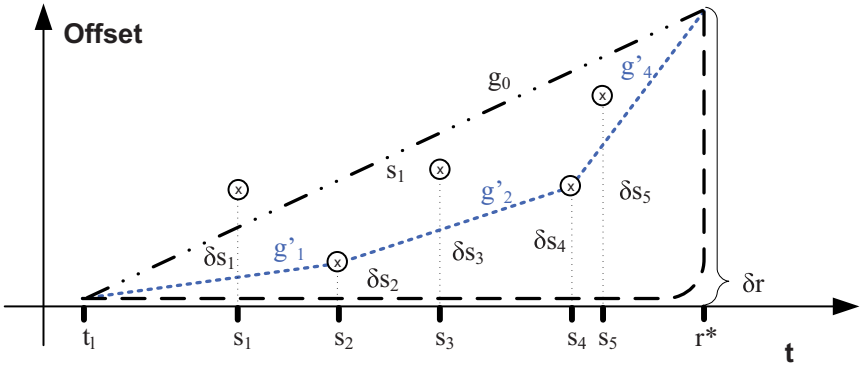


Figure 6.5: Backward amortization: Determination of the piece-wise linear correction.

amortization interval. For every receive event with $\delta r > 0$ encountered while traversing the trace in forward direction, a piece-wise linear amortization is started with the parameters

- $t_r := r* = r - \delta r$,
- $\delta e_r := \delta r$, and
- $m := const$.

Here, the left corner of the amortization interval is given as

$$t_l := t_r - \frac{\delta e_r}{m}.$$

If the value of t_l is earlier than the start of the trace t_b then the algorithm continues with $t_l := t_b$ and $m := \delta e_r/(t_r - t_b)$. Note that an event can be part of several overlapping backward amortizations and so t can be enlarged several times in the course of the algorithm.

Once a receive event with $\delta r > 0$ has been identified (at timestamp r^* in the figure), the algorithm searches the amortization interval $[t_l, t_r)$ for the send event with

$$\max_i(m_i := \frac{\delta e_r - \delta s_i}{t_r - s_i})$$

and $m_i > m$. If there is no such send event then the linear amortization g_0 is applied to $[t_l, t_r)$ according to Equation 6.1 and the algorithm continues the forward traversal of the event stream. However in Figure 6.5, the maximum slope can be found in the interval between the receive event at timestamp r^* and the timestamp s_4 of the fourth send event in the amortization interval (see linear function g'_4). As a consequence, a pice-wise linear interpolation function is needed.

$$g_0(t) := m * t - (m * t_l) \tag{6.1}$$

In general, if there is such a send event in the amortization interval, the linear amortization g'_i is applied to $[s_i, t_r)$ according to Equation 6.2.

$$g'_i(t) := m_i * t + (\delta s_i - (m_i * s_i)) \tag{6.2}$$

Additionally, if δs_i is larger than zero the algorithm starts again with the parameters

- $t_r := s_i$,
- $\delta e_r := \delta s_i$, and
- $m = \delta s_i/(s_i - t_l)$.

In our example, the algorithm starts again with $t_r = s_4$, , $\delta e_r := \delta s_4$, and $m = \delta s_4/(s_4 - t_l)$. As can be seen in this example, δe_r is initially set to δr (i.e., the jump applied during forward correction) and only set to δs_i (i.e., maximum backward amortization allowed for a send event) when needed to calculate a piece-wise linear interpolation function. As shown in Figure 6.5, the resulting piece-wise linear interpolation function is finally composed by the linear functions g'_4, g'_2, and g'_1.

6.5 MPI Combined with OpenMP

The original CLC algorithm does not account for direct violations of shared-memory event semantics in the original trace either. In fact, the algorithm neither restores nor preserves happened-before relations in shared-memory operations because the constituent events of such constructs are treated as internal events. As discussed in Chapter 4, while the algorithm corrects message-passing event semantics, it may introduce violations of the logical event order in shared-memory operations even though their event order was not violated in the original trace, as expected on SMP nodes with usually well synchronized local clocks. For this reason, this section describes the parallel implementation to synchronize happened-before relations in shared-memory event semantics, as introduced in Chapter 5.

Following the same parallelization strategy, these extensions are implemented on top of PEARL in combination with the previously described infrastructure for MPI traces. As discussed in the sections above, for traces of programs that use the combination of MPI and OpenMP, the PEARL usage model assumes multiple processes and a fixed number of threads assigned to each process. Implementing a one-to-one mapping of target application processes and threads to timestamp synchronization processes and threads, the synchronization becomes a parallel program having as many processes and threads per process as the target application that generated the trace data. Note that the current usage model is restricted in that it only supports MPI calls on the master thread (i.e., MPI funneled mode). On a more technical level, all timestamp synchronization threads read the thread-local trace data of "their" application thread into main memory and traverse the event traces in parallel while exchanging information at synchronization points. That is, each thread scans the event trace for clock condition violations and applies forward and backward amortization, as introduced in the previous sections. Hence, this section introduces the extensions of the parallel implementation aiming at

Listing 6.1: Example template function: OMP_Allreduce_max().

```
template <class T>
void OMP_Allreduce_max (T lval, T& gval)
{
  #pragma omp barrier
  #pragma omp single
  {
    if (std::numeric_limits<T>::is_integer)
      gval = std::numeric_limits<T>::min();
    else
      gval = -std::numeric_limits<T>::max();
  }
  #pragma omp critical
    gval = std::max(lval, gval);
  #pragma omp barrier
}
```

(i) identifying the logical event type (logical send, logical receive, or internal event) of an event related to a shared-memory operation and

(ii) defining operations used during the replay phases to exchange timestamps.

In order to describe happened-before relations among shared-memory events, events pertaining to creating or terminating a team of threads, entering or leaving parallel or barrier regions, and acquiring or releasing lock variables may be classified as logical send or receive events for the purpose of the algorithm. Events indicating the creation or termination of a team of threads and events indicating the acquisition and release of lock variables are directly identified through their event type as found in the original trace. For events related to entering or leaving parallel or barrier regions, the logical event type is derived from the region name (i.e., `omp_parallel`, `omp_barrier`) and the role (i.e., master thread) a particular thread plays in the operation.

Furthermore, functions are defined to exchange timestamps during the replay phases. The respective function invoked to accomplish the data transfer during the forward and backward replay depends on the operation originally used by the target application. For this purpose, shared-memory constructs are classified according to Chapter 5: team creation, team termination, barrier, and locks. The corresponding function used during the replay depends on the respective class to which the construct belongs. Note that an MPI library has software functions that already provide the necessary communication pattern (e.g., $1 - to - N$ by `MPI_Bast`), whereas OpenMP does not provide such functions. As a consequence, function templates are needed to emulate the above-mentioned communication classes in OpenMP. For instance, Listing 6.1 presents a template function which updates a shared global variable with the maximum value of thread-local variables using OpenMP constructs (`OMP_Allreduce_max()`). Given

Table 6.3: Timestamps exchanged among communication peers during the two replay phases for different shared-memory communication types.

Type of operation	Timestamp exchanged	Function
Forward replay		
Team creation	fork	`OMP_Bcast()`
Team termination	max(all OpenMP exits)	`OMP_Reduce_max()`
Barrier	max(all OpenMP enters)	`OMP_Allreduce_max()`
Locks	lock-release	`OMP_RLock()`
Backward replay		
Team creation	min (all enters)	`OMP_Reduce_min()`
Team termination	join	`OMP_Bcast()`
Barrier	min(all OpenMP exits)	`OMP_Allreduce_min()`
Locks	lock-acquisition	`OMP_RLock_rev()`

that all threads need to execute this procedure simultaneously to circumvent race conditions, two OpenMP barriers enclose the main body consisting of a single and critical section. The former sets the default value of the global variable, whereas the latter implements the update mechanism of the routine. The current implementation uses such function templates with functionality similar to their MPI counterparts. Table 6.3 lists the functions used during the replay along with the events which will have their timestamps exchanged. Besides the already discussed `OMP_Allreduce_max()` operation, `OMP_Allreduce_min()` is used to determine the minimum value of thread-local variables. In addition, `OMP_Reduce_max()` and `OMP_Reduce_min()` update a master-local variable with the maximum or minimum value of a thread-local variable, respectively. Next, `OMP_Bcast()` updates a shared variable with the value of a master-local variable.

The timestamp synchronization also encompasses the preservation of the logical event order in lock-acquisition/lock-release lock sequences. Single locks are identified by their lock identifier. In addition, the tracing library maps critical sections onto lock events. Note that the name of a critical section corresponds to a lock identifier. However, the tracing library does not provide event attributes to determine which lock occurred before another lock (see Chapter 1). As a consequence, the chronological sequence of locks in the original trace is all we "know" about the event order which should be preserved during the timestamp synchronization. For this purpose, the chronological event order of lock events along with their lock identifier is determined and stored in a global data structure before the algorithm is applied. Subsequently during the forward replay, the locking and unlocking is replayed using the two function templates `OMP_ALock()` and `OMP_RLock()`, respectively. `OMP_ALock()` waits unless all preceeding locks are processed by the algorithm. Once all preceeding locks are processed, the timestamp of the matching lock-release event required to fulfill the happened-before relation can be read. Note that this timestamp was already updated in the `OMP_RLock()` function. Two special versions `OMP_ALock_rev()` and `OMP_RLock_rev()` are used during the backward replay, because in this case the role of logical send and receive events needs to be determined in backward order.

6.6 Wide-Area Communication

The solution of critical numerical problems may require more processing power and memory capacity than is available on a single parallel machine. Often, coupling multiple independent parallel machines (i.e., metahosts) to form a more powerful metacomputer [84] 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 recent study by Wong and Goscinski [95], achieving satisfactory application performance is difficult. 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. Automatic pattern search in event traces is a suitable method to identify wait states that appear as a result of using a metacomputer consisting of multiple geographically dispersed metahosts [12]. For this reason, the next section describes the necessary extensions of the Scalasca trace-analysis framework to support the automatic performance analysis of metacomputing applications.

6.6.1 Metacomputing Scenario

The trace analysis performed by Scalasca identifies wait states that occur when processes reach synchronization points at different moments. When developing efficient applications for metacomputers, a major difficulty arises from load balancing on the heterogeneous hardware found in these environments. Since load imbalance often manifests itself as processes arriving in an untimely manner at synchronization points, the general concept behind the pattern analysis is well suited to guide application developers in recognizing problems of this kind. For this reason, Scalasca's trace analysis was extended to metacomputing environments consisting of multiple independent parallel computers or clusters. Major challenges addressed include

(i) establishing a global view of trace data in the absence of a global file system and

(ii) synchronizing timestamps across a hierarchy of network links with different latencies.

The extensions also encompass the definition of new wait-state patterns so that waiting times resulting from inter-metahost communication can be distinguished from purely local ones. To distinguish pattern instances that result from processes on different metahosts waiting for each other, special "grid" versions of its existing patterns were added to Scalasca. In the case of point-to-point communication, the analysis recognizes whether sender and receiver reside on different metahosts. In the case of collective communication, the entire communicator is searched for processes differing in their machine (i.e., metahost) location component. The grid versions of these patterns simply check whether communication across different metahosts has taken place. Figure 6.6 shows an example of a trace-analysis report with three different metahosts. The graphical browser organizes the grid patterns in a hierarchy, which allows a convenient in-depth study of the performance behavior at varying levels of granularity.

Using the grid-enabled version of Scalasca in combination with statistical analyses and timeline visualization provided by Vampir, the performance of the multi-physics code MetaTrace [39] was optimized on the heterogeneous and geographically dispersed metacomputing

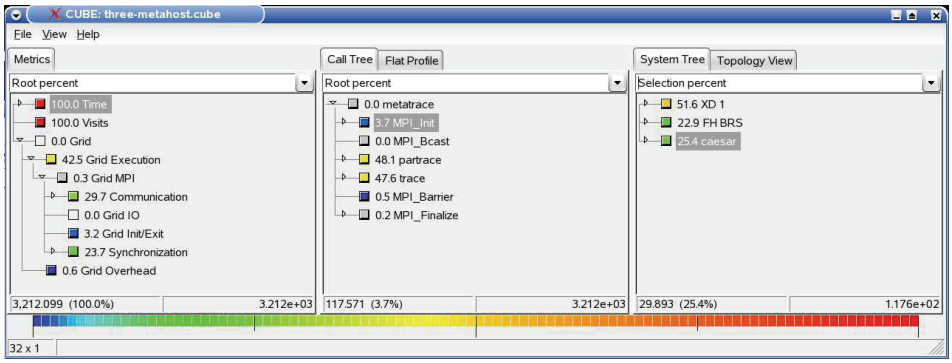


Figure 6.6: Analysis result presentation of an application running on three metahosts: Grid-specific patterns are distributed across three metahosts.

testbed Viola [16] by more than a factor of two, eliminating a large fraction of the inter-metahost waiting times previously observed [6].

The extended and parallelized CLC algorithm is well suited for computational grids because it accounts for the hierarchy of latencies as found in these systems through an additional inter-machine latency. Moreover, its distributed memory and processing scheme can establish the global view of trace data in the absence of a global file system. However, the algorithm requires timestamps with limited errors, which can be achieved through linear offset interpolation between program start and end. The linear offset interpolation mechanism, as discussed in Chapter 2, needs to be revised because its design ignores the above-mentioned hierarchy of latencies. Thus, to employ the replay mechanism in metacomputing environments, the next section defines the necessary infrastructure to accurately measure clock offsets in distributed environments based on hierarchical networks.

6.6.2 Hierarchical Offset Measurement

The current linear offset interpolation approach 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 offset relative to each other, which is calculated by subtracting their offsets relative to the master, might be unacceptable high in relation to the latency of the internal network between them [12].

The current linear offset measurement follows a centralized approach that is based on the transitivity of offset relations (see Chapter 2). As illustrated in Figure 6.7, all worker processes

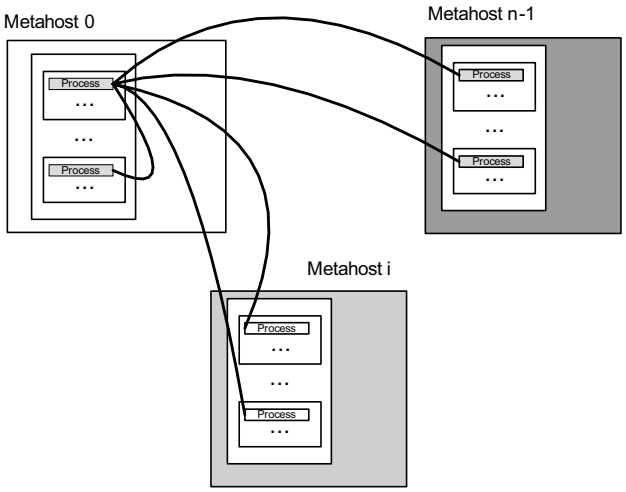


Figure 6.7: Flat offset measurements between each worker process and a single master process.

measure their offset relative to the master and it is assumed that the offsets relative to each other can be derived from their offset to the master. The error of the offset measurement between two processes at a given moment (derived 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.

The current offset measurement approach is *flat* in that all workers measure their offset by contacting the master directly without taking the hierarchy of network latencies between them into account. In contrast, Figure 6.8 presents the new scheme following a *hierarchical* approach: Using a unique metahost identifier, each metahost determines a local master process. After that, one metamaster is chosen from 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 global clock, this second step is omitted. Finally, the offset to the metamaster is calculated by adding the two measured offset values. Since all workers within the same metahost now use the same inter-metahost offset measurement, their relative offset remains unaffected. An experimental validation of the new approach is presented in Chapter 7.

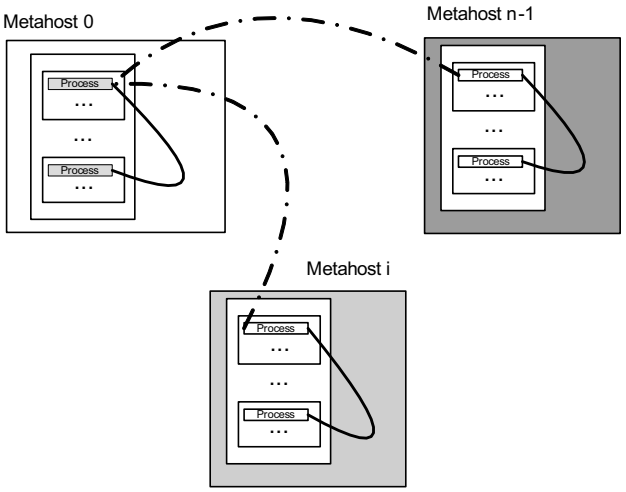


Figure 6.8: Hierarchical offset measurements between each local worker process and their local master.

6.7 Summary

As a serial algorithm, the original CLC method offers only limited scalability. However, it is crucial that the timestamp synchronization scales to large numbers of application processes. Hence, this chapter presented a parallel CLC algorithm design for the retroactive correction of timestamps based on logical clocks.

To accomplish the synchronization in a scalable way, both distributed memory and parallel processing capabilities are exploited by processing separate local trace files in parallel and replaying the original communication on as many CPUs as were used to execute the target application itself. During the replay, sending and receiving threads exchange relevant information needed to synchronize event timestamps according to the CLC algorithm. This parallel version of the CLC algorithm is divided into two replay phases: a forward phase for the logical clock with forward amortization and a backward phase for the backward amortization. Only if clock condition violations appear during the forward phase, the backward phase is needed to smooth the introduced jump discontinuities at formerly violated receive events.

The algorithm has been incorporated into the Scalasca framework to facilitate trace analyses of longer runs on larger cluster systems and in metacomputing environments. The replay-based CLC algorithm allows the correction of large-scale applications and should guarantee good scaling behavior, which is investigated in Chapter 7. As introduced in Chapter 5, the algorithm uses happened-before relations in message-passing and shared-memory operations to enable the correction of parallel programs with collective and shared-memory operations.

Due to its distributed memory and processing scheme, the parallel CLC algorithm is well suited for metacomputing environments and also accounts for the hierarchy of latencies as

found in these systems. As the algorithm, however, requires timestamps with limited errors, which can be achieved through linear offset interpolation between program start and end, the linear offset interpolation mechanism is revised because its design ignores the above-mentioned hierarchy of latencies. The proposed hierarchical mechanism for measuring clock offsets is able to deal with substantially different latencies in external and internal networks during offset measurements across geographically dispersed metahosts.

In the next chapter, we evaluate the parallel CLC algorithm in terms of its suitability for realistic parallel programs running on cluster systems or metacomputing environments. Before investigation the scaling behavior, we review the accuracy of the extended and parallelized CLC algorithm. In particular, we investigate whether the collaterally introduced deviations of local intervals remain within acceptable limits. Finally, an experimental validation of the hierarchical offset measurement scheme is presented.

Chapter 7

Experimental Evaluation

To circumvent the obstacles for trace analysis arising from the absence of synchronized hardware clocks and their non-constant clock drifts, this thesis describes an approach to retroactively synchronizing event traces of realistic parallel programs. Extensions of the controlled logical clock to handle collective and shared-memory operations have been already proposed in Chapter 5, followed by the parallel algorithm design and its implementation within the Scalasca trace-analysis framework in Chapter 6. Nonetheless, an experimental evaluation is crucial to validate accuracy and scalability of the proposed mechanisms for retroactively synchronizing timestamps of concurrent events. First, this chapter experimentally investigates the hierarchical offset measurement scheme in metacomputing environments. Second, it gives evidence of the frequency and the extent of clock condition violations in event traces of parallel programs after applying linear offset interpolation. Third, it presents an evaluation of the accuracy and scalability of the parallel controlled logical clock algorithm when applied to traces of parallel programs running on either cluster systems or metacomputers.

7.1 Experimental Setup

This section summarizes the computing systems used to evaluate the mechanisms proposed in Chapters 5 and 6. It starts with the cluster systems followed by the metacomputers and their setup used in this evaluation study.

7.1.1 Cluster Systems

In order to evaluate the scalability and accuracy of the parallel controlled logical clock algorithm and also to give evidence of the frequency and the extent of clock condition violations in event traces, experiments were taken on the following cluster systems:

Cacau consists of 200 compute nodes, each with 1 dual-core Intel Xeon EM64T CPU running at 3.2 GHz. Located at the High Performance Computing Center Stuttgart, each node of Cacau is linked with a Voltaire Infiniband network and a Gigabit Ethernet. The measured MPI inter-node latency was $4.7\ \mu s$, the measured MPI intra-node latency was $1.0\ \mu s$.

Jaguar is a Cray XT4/5 system located at the National Center for Computational Sciences at Oak Ridge National Laboratory. The XT4 partition used for our experiments has a total number of 7,832 quad-core 2.1 GHz AMD Opteron nodes. Each node is connected to a distinct Cray SeaStar router through HyperTransport with all the SeaStars arranged in a 3-D-torus network topology. The measured MPI inter-node latency was $8.6\ \mu s$, the measured MPI intra-node latency was $0.6\ \mu s$.

MareNostrum consists of 2,560 JS21 blade compute nodes, each with 2 dual-core IBM 64-bit PowerPC 970MP processors running at 2.3 GHz. Located at the Barcelona Supercomputing Center, the compute nodes of MareNostrum communicate primarily through a Myrinet network with Myrinet adapters integrated on each server blade. The measured MPI inter-node latency was $7.7\ \mu s$, the measured MPI intra-node latency was $1.3\ \mu s$.

Nicole consists of 32 compute nodes (stage 3), each with two quad-core AMD Opteron processor running at 2.4 GHz. Located at the Jülich Supercomputing Centre at Forschungszentrum Jülich, each node is linked with an Infiniband network. The measured MPI inter-node latency was $4.5\ \mu s$, the measured MPI intra-node latency was $1.5\ \mu s$.

7.1.2 The Viola Metacomputer

The evaluation of the hierarchical offset measurement scheme was carried out on the Viola grid. Viola [16] is a project funded by the German Ministry for Education and Research, which provides a testbed for advanced optical network technology. A major focus is the enhancement and test of new advanced grid applications. Applications on the Viola grid cover various research disciplines including environmental research, the design of complex technological systems like biosensors and crystal growth for microchip wafer production, and structural mechanics in engineering. This section describes the network topology, hardware architecture, and middleware of the Viola grid used in the course of this evaluation study.

Network Topology and Hardware Architecture: The network behind the Viola grid consists of a 10 Gbps backbone network with connections to workstations and compute clusters located at various sites in Germany including Sankt Augustin, Jülich, Bonn, Nürnberg, and Erlangen. The nodes of the connected compute clusters are linked to the backbone with 1 Gbps adapters. The network topology of the testbed section used in this study is illustrated in Figure 7.1.

As can be seen in Figure 7.1, the section comprises three sites, the Center of Advanced European Studies and Research (CAESAR) in Bonn, FH Bonn-Rhein-Sieg Sankt Augustin (FH-BRS), and Forschungszentrum Jülich (FZJ), which are connected via high-speed optical links offering a bandwidth of 10 Gbps between each pair of sites. The three sites lie between 20 and 100 km apart and provide the following cluster systems:

- A PC Linux cluster with 32 2-way Intel Xeon SMP nodes at 2.6 GHz with a Gigabit Ethernet interconnect located at CAESAR.

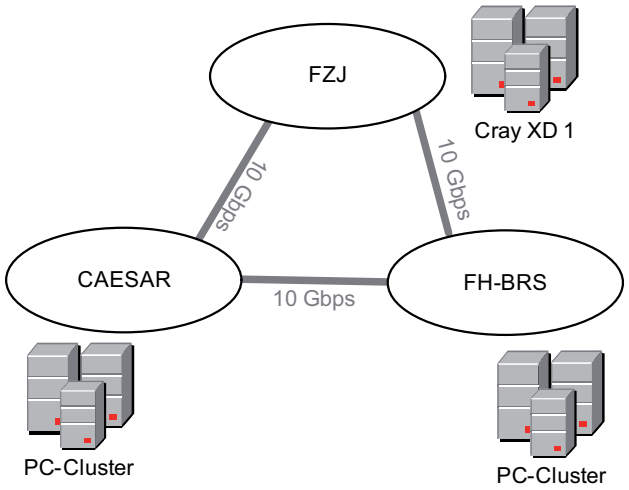


Figure 7.1: Network topology of the Viola grid section used for the experiments.

- A PC Linux cluster with 6 4-way AMD Opteron PC nodes at 2.0 GHz with a usock over Myrinet interconnect located at FH-BRS.
- A Cray XD1 Linux cluster with 60 2-way AMD Opteron SMP nodes at 2.2 GHz with a usock over RapidArray interconnect located at FZJ.

These components form a very heterogeneous metacomputer layout with a hierarchy of different network latencies and varying characteristics of the compute clusters, which differ also with respect to their operating systems (different versions of Linux) and compilers.

Middleware: Running a parallel application on such a metacomputer needs also middleware components for application startup and a wide-area communication library for the transfer of data between application processes residing on geographically dispersed metahosts. The middleware interacts with local resource managers to co-schedule jobs on different clusters. The communication library should support transparent high-bandwidth and low-latency message transfers between all nodes of the attached clusters.

The co-scheduling of jobs on different clusters in the Viola grid is managed by the grid middleware UNICORE [47] which has been enhanced by adding a meta-scheduler for the simultaneous allocation of compute and network resources. Bierbaum et al. [14] described this UNICORE-based infrastructure supporting the co-allocation of metacomputing resources in more detail, with special emphasis on the intricate task of coordinating network allocation with application startup.

Viola uses MetaMPICH [15], the grid-enabled MPICH-based MPI implementation developed at RWTH Aachen University, to establish direct connections to the external network from each node. MetaMPICH supports these direct connections through a multi-device architecture that

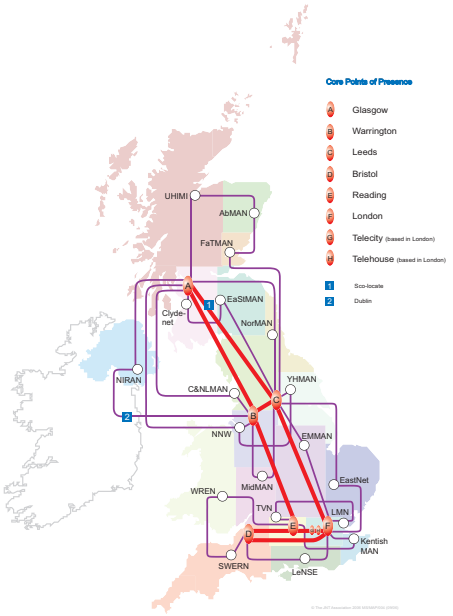


Figure 7.2: Network topology of the Janet backbone [57].

allows external communication within the Viola grid with the maximum bandwidth of 1 Gbps per node across the wide-area network without the involvement of dedicated router processes. The high bandwidth of the backbone can only be used if the data transmission between the clusters is done in parallel.

7.1.3 National Grid Service

In order to evaluate the controlled logical clock algorithm on a metacomputer, measurements were taken on the UK National Grid Service (NGS) grid [72], because at that time the Viola grid was already decommissioned. 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 Figure 7.2. NGS resources are linked to this

backbone and comprise the four founding members 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 metacomputer constituents at University of Leeds and 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 $4.4\ \mu s$, the measured MPI intra-node latency was $1.5\ \mu s$.
- 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 $4.4\ \mu s$, the measured MPI intra-node latency was $1.5\ \mu s$.

The nodes of the connected compute clusters are linked to the backbone with 1 Gbps adapters. Again, 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) [40, 63]. 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 on each site, whereas HARC creates reservations for cross-site jobs and implements the co-allocation across the wide-area network.

Moreover, NGS uses MPIg, an MPICH-based grid-enabled MPI implementation, to establish direct connections to the external network from each node [59]. 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 devices supporting different flavors of communication substrates. Similar to MetaMPICH, 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.

Table 7.1: Latencies of the internal and external networks in Viola.

	mean [μs]	std. deviation [μs]
FZJ - FH-BRS (external network)	9.88E+02	3.86E+00
FZJ (internal network)	2.15E+01	8.14E-01
FH-BRS (internal network)	4.44E+01	3.60E-01

7.2 Hierarchical Offset Measurement

To accurately measure clock offsets in distributed environments with hierarchical networks, hierarchical offset measurements have been proposed in Chapter 6. This section presents experimental results demonstrating the benefits of these hierarchical offset measurements [12]. First, latency measurements confirm the assumption that offset measurements among distributed machines are less accurate. Second, further measurements of clock condition violations indicate that the hierarchical scheme may improve the accuracy of offset measurements. Third, using a multi-physics application, a comparison of analysis reports calculated from timestamps synchronized with either flat offset measurements or hierarchical offset measurements finally demonstrates that the hierarchical scheme may increase the accuracy of the overall analysis.

Investigating the latency between the sites at FZJ and FH-BRS of the Viola grid reveals that the latency of the external network exceeds the latency of the internal network by two orders of magnitude. Also, the latencies of the internal networks differ significantly. Table 7.1 shows the latencies measured with a small MPI benchmark based on MetaMPICH. The standard deviation is an indicator for the precision of offset measurements across these links, which confirms the assumption that offset measurements across the external network are much less accurate in absolute terms than those across the internal networks.

Moreover, the accuracy of the hierarchical synchronization scheme was verified using another benchmark that has been specifically designed to exchange a large number of short messages between varying pairs of processes. In this way, the benchmark produces pairs of send and receive events that are chronologically close to each other. Table 7.2 shows the number of clock condition violations found in traces from this benchmark for synchronization based on

(i) a single flat offset measurement without compensation for drift (i.e., offset alignment at program initialization),

(ii) two flat offset measurements (i.e., at program initialization and finalization) and subsequent linear interpolation, and

(iii) two hierarchical offset measurements (i.e., at program initialization and finalization) and subsequent linear interpolation.

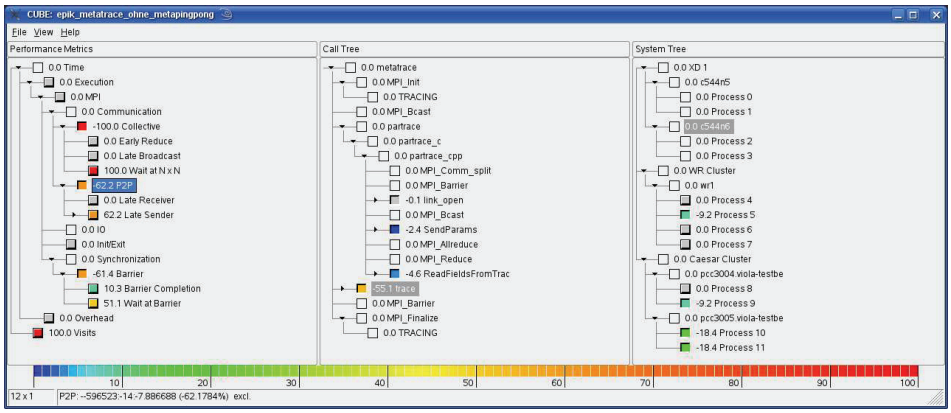
Although clock condition violations are still possible, this experiment shows that the hierarchical scheme was able to significantly reduce the number of clock condition violations.

Table 7.2: Number of clock condition violations.

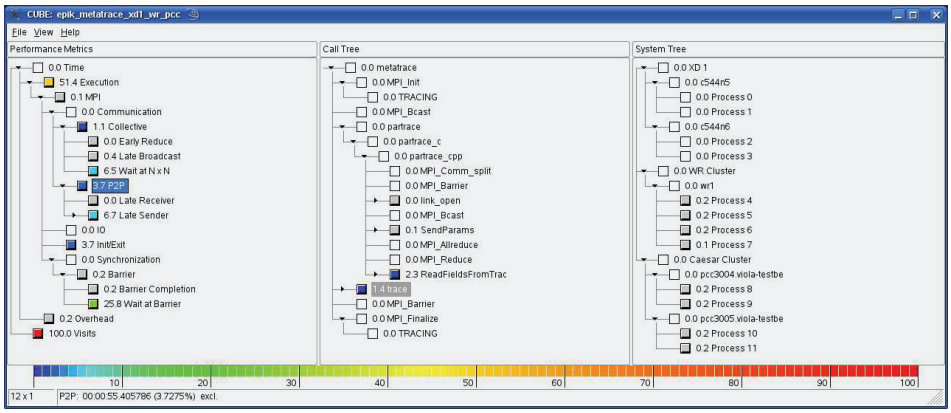
Measurement	Number of clock condition violations
single flat offset	7560
two flat offsets	2179
two hierarchical offsets	0

The environmental-science application Metatrace [39] was used as a test application on the Viola grid. To simulate the transport of pollutants in groundwater, Metatrace is split into two parallel simulation submodels, Trace and Partrace. Whereas Trace simulates water flow in porous media, Partrace computes the transport of solutes in this water flow. Trace applies a three-dimensional domain decomposition with nearest-neighbor communication, whereas Partrace tracks individual particles. For simulating pollutant transport in non-steady flows, the simultaneous execution of both submodels is crucial. Metatrace couples the two submodels through a parallel connection between the two submodels. This connection is mainly used in one direction for the transfer of the distributed three-dimensional velocity field from Trace to Partrace whenever Trace completes a simulation step. The mostly unidirectional communication scheme makes Metatrace suitable for running efficiently in a metacomputing environment. Running each submodel on a single metahost allows the internal communication to benefit from the cluster-internal low-latency networks with synchronization as well as data exchange between the two submodels left to the high-latency network between the different clusters. The unidirectional and low-frequency communication between the two submodels is done synchronously over the Viola backbone network through the node-local network adapters. After receiving the data, Partrace replicates the received velocity field on each node by synchronously distributing it across all Partrace nodes using a systolic loop.

The instrumented application was executed on the Viola grid using three metahosts and subsequently analyzed. Linear offset interpolation was applied between program initialization and finalization using either flat offset measurements or hierarchical offset measurements. Figure 7.3 shows the respective analysis reports, where analysis data were calculated based on timestamps synchronized with either two flat offset measurements (Figure 7.3(a)) or two hierarchical offset measurements (Figure 7.3(b)), respectively. While the former report shows inconsistant dependencies (i.e., sunken reliefs represent negative times), the latter report describes consistent results and so may form the basis for a subsequent performance optimization. For instance, the former report indicates that -62.2% of the execution time was spent in MPI point-to-point communication calls which is impossible. In contrast, the latter report shows that 3.7% of the execution time was spent in MPI point-to-point communication calls. Note that this time does not include the time lost waiting in MPI point-to-point communication, because this waiting time is assigned to the respective *Late Sender* and *Late Receiver* patterns. Indeed, this comparison shows that the hierarchical scheme significantly increases the accuracy of the overall analysis. Although hierarchical offset measurements increase the accuracy of timestamps to some degree, traces may, however, still exhibit clock condition violations. Given that the CLC algorithm works best on timestamps with limited clock errors, hierarchical offset measurements can be used to satisfy this requirement. Section 7.3.3



(a) Two flat offset measurements.



(b) Two hierarchical offset measurements.

Figure 7.3: Analysis reports calculated based on timestamps synchronized with flat offset measurements or hierarchical offset measurements.

presents results showing that the CLC algorithm removes residual inconsistencies in event traces taken on metacomputers.

7.3 Logical Synchronization

This section investigates the accuracy and scalability of the extended and parallelized CLC algorithm using traces of parallel programs taken on either cluster systems or metacomputers. It also gives evidence of the frequency and the extent of clock condition violations in event traces of a realistic parallel programs.

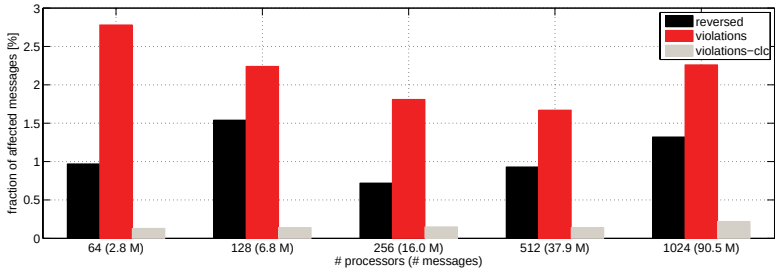
7.3.1 Message Passing

This section focuses on message-passing applications and evaluates the CLC algorithm in terms of its accuracy and scalability on a range of cluster systems using the SMG2000 and LAMMPS benchmarks [10, 11]. On a more technical level, after investigating whether clock condition violations occurred in the original traces, the efficiency and effectiveness of the logical synchronization approach are demonstrated by calculating metrics showing the accuracy and scalability of the correction algorithm. While the former is addressed by the extent to which the length of local intervals is preserved, comparing the scaling behavior of the correction algorithm with the original target application indicates the scaling quality, because the replay-based processing scheme mimics the original communication.

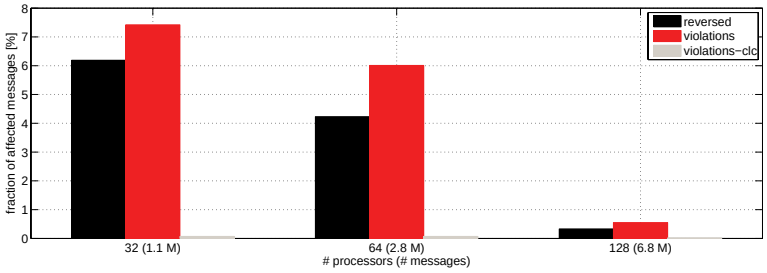
The MPI version of the ASC SMG2000 benchmark [17] was used as a first test application on MareNostrum and Cacau. The SMG2000 benchmark is a parallel semi-coarsening multigrid solver that uses a complex communication pattern and performs a large number of non-nearest-neighbor point-to-point communication operations. Applying a weak scaling strategy, a fixed $16 \times 16 \times 8$ problem size per process with five solver iterations was configured. Semi-automatic instrumentation was used to reduce the overall number of events and the call tree depth.

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 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.

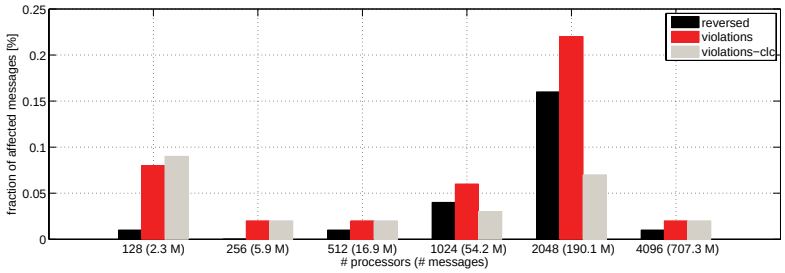
The LAMMPS benchmark was used as a second test application on Jaguar. This benchmark is a classical molecular dynamics code designed to run efficiently on parallel computers [61, 77]. It was developed at Sandia National Laboratories, a US Department of Energy (DOE) facility, and is shipped with the Sequoia benchmark suite [3]. Automatic compiler instrumentation was used to instrument the LAMMPS application, which ran with a scaled-size Lennard-Jones potential causing it to execute 10000 iterations. Applying a weak scaling strategy the execution time was approximately constant and roughly 20 min. Given that tracing the full run would consume a prohibitively large amount of storage space, the main computational loop was traced only between iteration 4500 and 5500. Again this "partial" tracing corresponds to the recommended practice of tracing only points that warrant a more detailed analysis. Whereas the sleep statements in SMG2000 caused the traces to have long periods without events, the LAMMPS traces did not contain such periods and therefore represent more "continuous" event



(a) SMG2000 on MareNostrum.



(b) SMG2000 on Cacau.



(c) LAMMPS on Jaguar.

Figure 7.4: Percentage of messages with the order of send and receive events being reversed, of messages with clock condition violations, and of clock condition violations explicitly corrected by the CLC algorithm during forward amortization.

streams. Compared to the SMG2000 traces, the LAMMPS traces included a larger event density (i.e., number of events per time) because filtering of uninteresting functions was not yet supported on Jaguar.

Figure 7.4 shows the frequency of clock condition violations on MareNostrum, Cacau, and Jaguar for differently scaled SMG2000 and LAMMPS runs. Since the number of violations varies between runs, the numbers represent averages across three measurements for each

Table 7.3: Average and maximum errors of message events in reversed order for SMG2000 and LAMMPS.

Platform	Avg. error [μs]	Max. error [μs]
	SMG2000	
MareNostrum	2.6	323
Cacau	4.3	186
	LAMMPS	
Jaguar	2.3	96

configuration. The numbers show the percentage of messages with the order of send and receive events being reversed in the original trace, of messages with clock condition violations ($t_{recv} < t_{send} + l_{min}$) in the original trace, and of clock condition violations explicitly corrected by the CLC algorithm during forward amortization. The numbers also include logical messages that can be derived by mapping collective communication onto point-to-point semantics. When visualized, messages with the order of send and receive events being reversed seem to flow backward in time. The number of violations explicitly corrected by the CLC algorithm is usually smaller than the initial number of violations because some of them are already implicitly removed during forward amortization before a correction can be applied. On MareNostrum, around 1.0% of the messages flow backward in time, while on Cacau the percentage ranges between 0.3 and 6%. Higher latencies on MareNostrum offer a potential explanation for the smaller number of violations detected on this system because higher latencies naturally insert a larger temporal distance between send and receive events of the same message. Even though the number of inconsistent messages on Cacau seem to decrease with growing numbers of processes, the results on MareNostrum do not confirm a clear correlation between the two indicators. On Jaguar too, all traces contained clock condition violations which shows that further synchronization is important to enable accurate trace analysis. In contrast to the SMG2000 experiments, a longer execution time along with fewer messages imposes a larger chronological distance between sending and receiving events and therefore the percentage of violated event semantics is smaller for the LAMMPS experiments. In addition, the 128 process experiment on Jaguar shows that the percentage of clock condition violations corrected by the CLC algorithm can also be larger than the percentage of clock condition violations in the original trace. Such a situation occurs, when the corrections during forward amortization cause new violations in subsequent operations. Table 7.3 lists the average and maximum displacement errors (i.e., the time the receive event appears earlier than the send event) of message events in reversed order, as seen in the original trace.

Of course, the transformation performed by the CLC algorithm raises the question of how accurate the modified traces actually are. To answer this question, it must first be acknowledged that traces with clock condition violations are inaccurate because they are inconsistent. The behavior they reflect violates causation and is therefore impossible. The CLC algorithm eliminates these inconsistencies, improving the accuracy of inter-process timings. A very simple metric quantifying this improvement is the fraction of clock condition violations found

Table 7.4: SMG2000: Relative deviation of the event distance. Unless indicated numbers are given in percent and rounded to two digits after the decimal point.

	MareNostrum					Cacau		
# CPUs	64	128	256	512	1024	32	64	128
Average	0.00	0.01	0.01	0.00	0.01	0.00	0.01	0.00
Maximum	27.17	461.74	411.52	311.64	974.44	82.78	69.31	16.77
Percentage of intervals with deviation above threshold								
$>$ 0.0%	80.80	96.21	97.34	98.23	99.07	92.43	93.67	24.27
$>$ 0.01%	0.15	0.63	0.39	0.48	0.81	0.47	0.27	0.04
$>$ 0.1%	0.15	0.61	0.38	0.46	0.79	0.45	0.26	0.04
$>$ 1%	0.01	0.10	0.06	0.07	0.18	0.09	0.04	0.00
$>$ 10%	0.00	0.01	0.00	0.00	0.01	0.01	0.00	0.00
$>$ 100%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Percentage of execution time consumed by intervals with deviation above threshold								
$>$ 0.0%	80.51	96.22	96.86	92.20	95.17	50.37	92.50	23.00
$>$ 0.01%	0.41	0.95	0.58	0.71	0.72	0.24	0.68	0.05
$>$ 0.1%	0.20	0.23	0.28	0.21	0.24	0.19	0.55	0.04
$>$ 1%	0.01	0.02	0.03	0.02	0.02	0.04	0.11	0.00
$>$ 10%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
$>$ 100%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

in the original trace (second column in Figure 7.4), which are all removed in the modified trace.

However, the necessary corrections applied to concurrent events in the course of the algorithm also modify – as a collateral effect – relative process-local event timings, which may lead to differences in the lengths of local intervals when comparing the original with the modified trace. Again, since the original timestamps have been taken with clocks that exhibit unstable drifts, there is, of course, no guarantee that the measured interval lengths are correct, although the deviation can be expected to be small in comparison to the total interval length. In addition, the original timings are all we "know" about the target application's local execution behavior. For this reason, this section evaluates how this knowledge has been preserved in the modified trace. To assess the fidelity of local timings after applying the CLC algorithm, the relative deviation of local interval lengths is determined, considering two different types of intervals:

(i) intervals between an event and the first event of the same process, which is referred to as the *event position*, and

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

Comparing local intervals in the original trace with their counterparts in the synchronized trace, the maximum relative deviation of the event position across all SMG2000 measurements

Table 7.5: LAMMPS: Relative deviation of the event distance. Unless indicated numbers are given in percent and rounded to two digits after the decimal point.

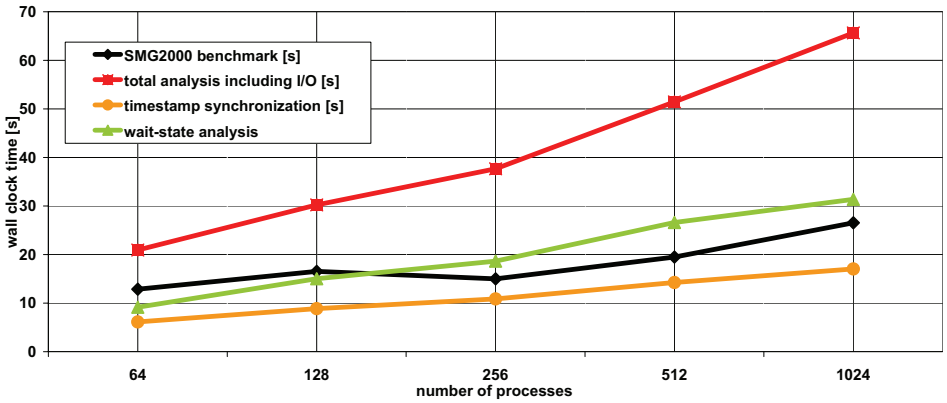
	Jaguar					
# CPUs	128	256	512	1024	2048	4096
Average	0.00	0.00	0.00	0.00	0.00	0.00
Maximum	392.28	80.55	278.71	884.76	1819.27	1459.41
Percentage of intervals with deviation above threshold						
$>$ 0.0%	32.44	9.20	7.46	38.06	65.74	43.34
$>$ 0.01%	0.07	0.01	0.01	0.08	0.19	0.12
$>$ 0.1%	0.07	0.01	0.01	0.08	0.18	0.12
$>$ 1%	0.03	0.01	0.00	0.02	0.04	0.04
$>$ 10%	0.00	0.00	0.00	0.00	0.00	0.00
$>$ 100%	0.00	0.00	0.00	0.00	0.00	0.00
Percentage of execution time consumed by intervals with deviation above threshold						
$>$ 0.0%	0.95	0.24	0.53	2.78	6.86	2.54
$>$ 0.01%	0.00	0.00	0.00	0.00	0.01	0.00
$>$ 0.1%	0.00	0.00	0.00	0.00	0.01	0.00
$>$ 1%	0.00	0.00	0.00	0.00	0.00	0.00
$>$ 10%	0.00	0.00	0.00	0.00	0.00	0.00
$>$ 100%	0.00	0.00	0.00	0.00	0.00	0.00

was below $0.0001\ \%$ and the maximum absolute deviation was $425.18\ \mu$s, roughly corresponding to the maximum displacement error observed (see Table 7.3). In addition, the maximum relative deviation across all LAMMPS measurements was again negligible and the maximum absolute deviation of $115.62\ \mu$s is also fairly consistent with the maximum displacement error observed (see Table 7.3).

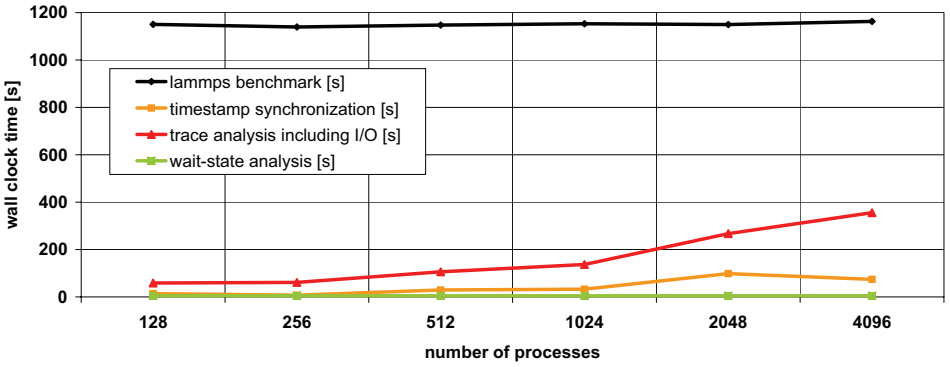
Tables 7.4 and 7.5 show the relative deviation of the event distance across different combinations of platform and number of processors for the SMG2000 and LAMMPS benchmarks. The numbers in individual columns are percentages and are rounded to two digits after the decimal point. They represent the maximum across three measurements. To account for the relatively long "correct" stretches artificially introduced by the sleep statements before and after the main computation in a SMG2000 trace, only the middle section of the trace between the sleep statements was considered. Furthermore, since deviations in larger intervals are more relevant to performance analysis than those in smaller intervals, the average was calculated using

$$\frac{\sum |\Delta t|}{\sum |t|}$$

to assign appropriate weight to larger intervals, with Δt being the absolute deviation and t being the original interval length.



(a) SMG2000 on MareNostrum.



(b) LAMMPS on Jaguar.

Figure 7.5: Scalability of the parallel timestamp synchronization on MareNostrum and Jaguar.

It can be seen that in spite of very small averages, deviations of occasionally more than 100% are still possible. Although the backward amortization is designed to smooth sudden jumps introduced by the logical clock with forward amortization, it can happen that a send event cannot be advanced far enough without causing a new clock condition violation when passing the corresponding receive event. To evaluate frequency and extent of such situations, two different metrics were calculated:

(i) the percentage of intervals whose deviation exceeds a certain threshold and

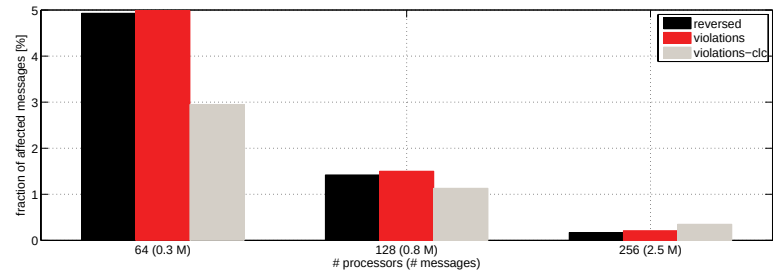
(ii) the percentage of execution time (accumulated across all processes) consumed by intervals whose deviation exceeds the threshold.

Table 7.6: Distribution of reversed messages and violated messages detected during the timestamp synchronization on Nicole.

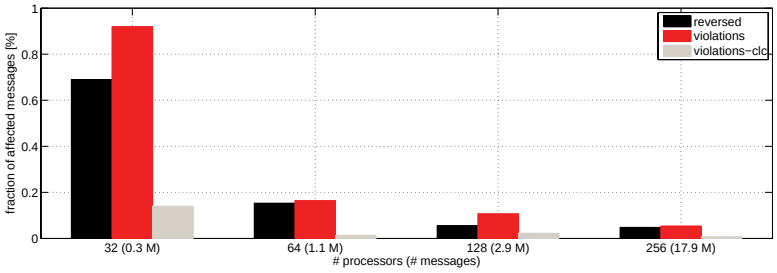
	PEPC			Jacobi			
# CPUs	64	128	256	32	64	128	256
# processes	16	32	64	16	32	64	128
# threads	4	4	4	2	2	2	2
Distribution of reversed messages							
message-passing	100.00	100.00	100.00	100.00	100.00	100.00	100.00
shared memory	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Violated messages detected during synchronization							
message-passing	44.50	46.16	42.59	81.15	84.80	81.78	40.36
shared memory	55.60	53.84	57.41	18.85	15.20	18.22	59.64

The results for SMG2000 (on MareNostrum and Cacau) and LAMMPS (on Jaguar), given in Tables 7.4 and 7.5 respectively, indicate that larger deviations are rare and that their influence on performance analysis results will usually be negligible. For the LAMMPS experiments on Jaguar, a higher percentage of intervals is corrected than the time fraction consumed by these intervals because the traces represent a more "continuous" event streams and so not all intervals are located in the inner 1000 iterations.

Finally, Figure 7.5 compares the scaling behavior of the parallel timestamp synchronization with the Scalasca wait-state analysis, the total analysis, and both test applications, respectively. Note that, the total analysis includes the parallel timestamp synchronization, the Scalasca wait-state analysis, the time for loading the traces, and the time for writing the analysis report. The scalability was evaluated on MareNostrum and Jaguar because of their larger size compared to Cacau. The results demonstrate that the wait-state analysis and the parallel timestamp synchronization scale easily to thousands of processes. The fact that the wait-state analysis, the parallel timestamp synchronization, and the execution of the respective test application exhibit roughly equivalent scaling behavior can be explained by the replay-based nature of the two analysis mechanisms. The execution time of LAMMPS is much larger than the execution time of the wait-state analysis and the parallel timestamp synchronization, because it includes the execution of all phases. In contrast to the SMG2000 experiments, the LAMMPS experiments show that at larger scales the timestamps synchronization clearly consumes more time than the wait-state analysis which can be explained by the aforementioned larger event density in the LAMMPS traces. More precisely, the wait state-analysis calculates only a simple profile metric (i.e., execution time of the respective code region) for non-communication related events, whereas the logical clock with forward amortization determines a new timestamp for each event. Moreover, the execution time of the parallel synchronization for LAMMPS on Jaguar shows a higher variation across different processor configurations than the execution time of the wait-state analysis does, which may have been caused by data dependences of the backward amortization. By design, the backward amortization initially performs a parallel backward replay whose runtime, of course, depends on the communication behavior of the target application and the number of passes. Note that across all experiments, more than one



(a) PEPC on Nicole.



(b) Jacobi on Nicole.

Figure 7.6: Percentage of messages with the order of send and receive events being reversed, of messages with clock condition violations, and of clock condition violations explicitly corrected by the CLC algorithm during forward amortization. The numbers only include point-to-point messages and logical messages derived by mapping collective message-passing communication onto point-to-point communication.

pass was never needed. Subsequently, discontinuities preceeding formerly violated receive events are smoothed by recursively calculating piece-wise linear interpolation functions and applying those to all events in the respective amortization interval. Hence, the execution time depends on the overall number of backward amortization steps and the number of events in the amortization intervals. Nonetheless, the fact that the total time needed by the integrated Scalasca analysis (synchronization and wait-state analysis) including loading the traces and writing the analysis report grows more steeply finally demonstrates that I/O (e.g, loading the traces) will increasingly dominate the overall behavior beyond 1024 processes, rendering the additional cost of the synchronization negligible.

7.3.2 Message Passing Combined with Shared Memory

The parallel CLC algorithm accounts for violations of message-passing and shared-memory event semantics. Thus, the algorithm is also suitable for hybrid parallel programs that use

Table 7.7: Average and maximum errors of message events in reversed order on Nicole.

Platform	Avg. error [μs]	Max. error [μs]
	PEPC	
Nicole	21.7	531
	Jacobi	
Nicole	3.5	98

MPI and OpenMP in combination. This section evaluates the CLC algorithm in terms of its accuracy and scalability for two hybrid test applications. It follows the same evaluation scheme as presented in Section 7.3.1 and lists only application-specific results.

As a first test application served the Pretty Efficient Parallel Coulomb (PEPC) application on the Nicole cluster. PEPC, which was developed at the Jülich Supercomputing Centre, is a parallel tree-code for rapid computation of long-range Coulomb forces in N-body particle systems [76]. In the course of this evaluation study, the original parallel processing scheme – an MPI implementation of the Barnes-Hut tree code according to the Warren-Salmon 'Hashed Oct Tree' structure – was enriched with shared-memory parallelism within the solver and integrator part of the application [5, 89].

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 [2]. In our test configurations, the runtime was approximately 30, 15 and 7.5 min. Given that automatic compiler instrumentation was used to instrument the PEPC solver and tracing the full run would consume a prohibitively large amount of storage space, selective tracing was applied so that the solver and integrator part were only traced during iteration 50. Similar to the LAMMPS experiments, the traces included a large event density and did not contain long time periods without events, as the SMG2000 experiments did.

For the purpose of accuracy evaluation, a hybrid version of the Jacobi solver [31], 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 application on Nicole. 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, was combined with MPI-based parallelism. Automatic compiler instrumentation was used and, following a strong scaling strategy, a fixed matrix size of 2000×2000 was configured. Similar to the SMG2000 experiments, a longer run was emulated by inserting sleep statements immediately before and after the main computational phase so that it was carried out ten minutes after initialization and ten minutes before finalization.

With respect to the programming model, Table 7.6 presents the distribution of reversed messages as seen in the original trace and of violated messages detected during the synchronization process. Although all traces contained clock condition violations, only violations of message-passing event semantics occurred in the original trace. To preserve the logical event order in the corrected trace, shared-memory event semantics, however, were temporarily violated and subsequently restored by the CLC algorithm. While a pure message-passing

Table 7.8: PEPC and Jacobi: Relative deviation of the event distance on Nicole. Unless indicated numbers are given in percent and rounded to two digits after the decimal point.

	PEPC			Jacobi			
# CPUs	64	128	256	32	64	128	256
# processes	16	32	64	16	32	64	128
# threads per process	4	4	4	2	2	2	2
Average	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Maximum	214.97	314.23	194.40	8.25	1.94	18.32	70.51
Percentage of intervals with deviation above threshold							
$>$ 0.0%	75.18	85.97	86.26	49.09	10.03	51.07	34.09
$>$ 0.01%	0.19	0.23	0.25	0.17	0.04	0.10	0.09
$>$ 0.1%	0.18	0.21	0.24	0.16	0.03	0.10	0.09
$>$ 1%	0.03	0.04	0.06	0.02	0.00	0.01	0.01
$>$ 10%	0.00	0.01	0.01	0.00	0.00	0.00	0.00
$>$ 100%	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Percentage of execution time consumed by intervals with deviation above threshold							
$>$ 0.0%	73.41	69.22	34.30	40.43	8.60	25.98	30.11
$>$ 0.01%	0.44	0.59	0.22	0.17	0.03	0.06	0.59
$>$ 0.1%	0.13	0.03	0.01	0.11	0.02	0.05	0.06
$>$ 1%	0.00	0.00	0.00	0.00	0.00	0.00	0.00
$>$ 10%	0.00	0.00	0.00	0.00	0.00	0.00	0.00
$>$ 100%	0.00	0.00	0.00	0.00	0.00	0.00	0.00

synchronization that does not account for shared-memory event semantics would leave these semantics violated, the CLC algorithm recognizes such situations and correctly preserves the correct order of shared-memory events in the synchronized event stream, enabling a consistent trace analysis. Moreover, the frequency of clock condition violations is shown in Figure 7.6 which includes the percentage of messages with the order of send and receive events being reversed in the original trace, of messages with clock condition violations in the original trace, and of clock condition violations explicitly corrected by the CLC algorithm during forward amortization. Given that none of the shared-memory event semantics was violated in the original trace, the numbers for reversed messages and violations only include point-to-point messages and logical messages that can be derived by mapping collective message-passing communication onto point-to-point communication. Again, all numbers represent averages across three measurements for each configuration. The extent of clock condition violations is shown in Table 7.7 which lists the average and maximum displacement errors (i.e., the time the receive event appears earlier than the send event) of logical message events in reversed order, as seen in the original trace.

To assess the fidelity of local timings after applying the CLC algorithm, the relative deviation of the event position and the event distance was again determined. For the Jacobi experiments

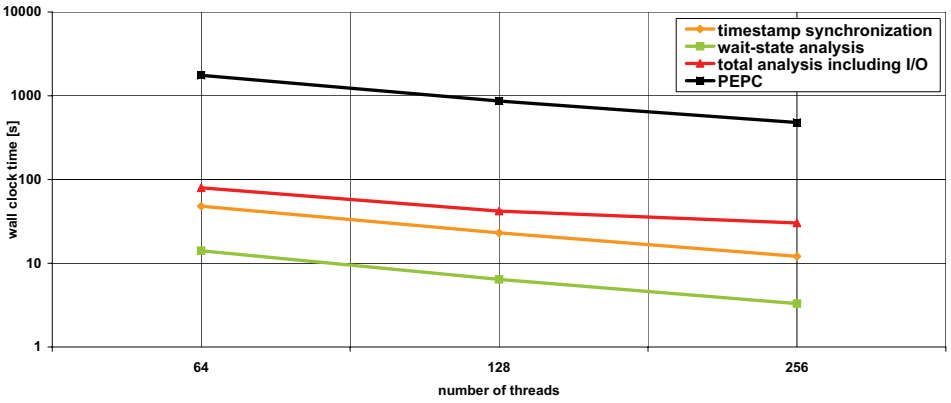


Figure 7.7: Scalability of the parallel timestamp synchronization and PEPC application on Nicole.

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 (see Table 7.7).

Moreover, Table 7.8 shows the relative deviation of the event distance across different numbers of processors for both test applications on Nicole. Again, the numbers in individual columns are percentages and represent the maximum across three measurements. Confirming our previous observation, it can be seen that in spite of very small averages, deviations of occasionally more than 100% are still possible, but larger deviations are rare and their influence on performance analysis results will usually be negligible.

In order to evaluate the runtime behavior of the hybrid synchronization method, Figure 7.7 presents scaling results of the parallel timestamp synchronization, the Scalasca wait-state analysis, the total analysis, and the PEPC solver itself. Note that, the total analysis includes the parallel timestamp synchronization, the Scalasca wait-state analysis, the time for loading the traces, and the time for writing the analysis report. The results demonstrate that the wait-state analysis, the parallel timestamp synchronization, and the execution of the PEPC benchmark itself exhibit roughly equivalent scaling behavior. To guarantee a fair comparison between the respective runtimes, it must be acknowledged that the wait-state analysis does not yet support the full OpenMP pattern analysis and only calculates some basic OpenMP metrics, whereas the parallel timestamp synchronization covers a more complete set of shared-memory event semantics. To consider the speedup behavior in isolation, Figure 7.8 shows speedup characteristics of the parallel timestamp synchronization, the wait-state analysis, and the PEPC benchmark itself. Again, the numbers for each configuration represent averages across three measurements and are normalized to the respective execution time of the 64

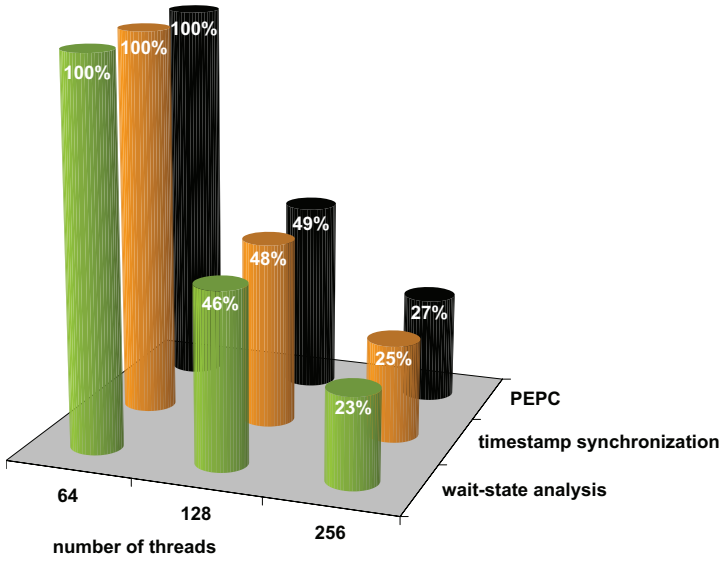


Figure 7.8: Normalized execution time of the parallel timestamp synchronization on Nicole.

thread configuration. As can be seen in this figure, the replay-based analysis mechanisms and the PEPC application show a similar and almost ideal speedup behavior.

7.3.3 Wide-Area Communication

Given the fact that performance optimization for a single cluster is already a non-trivial task that requires substantial tool support, we already argued that this is even more important for metacomputing environments. Of course, when carried out in a metacomputing environment, the accuracy of the pattern analysis also depends on the comparability of timestamps taken on processors potentially located on geographically dispersed metahosts. Although the proposed hierarchical offset measurement scheme already significantly increases the accuracy of the overall analysis, traces may still exhibit clock condition violations. Desirable as a prerequisite of the CLC algorithm, hierarchical offset measurements can be, however, used to limit the error of timestamps. This section presents results showing that the CLC algorithm removes remaining inconsistencies in event traces taken in metacomputing environments. It shows evaluation results of the CLC algorithm taken on the NGS grid using the SMG2000 benchmark and follows the same evaluation scheme as presented in Section 7.3.1, listing only application specific results.

As a test application, the MPI version of the SMG2000 benchmark was used on the NGS grid [17]. Similar to the former SMG2000 experiments, a fixed $16 \times 16 \times 8$ problem size

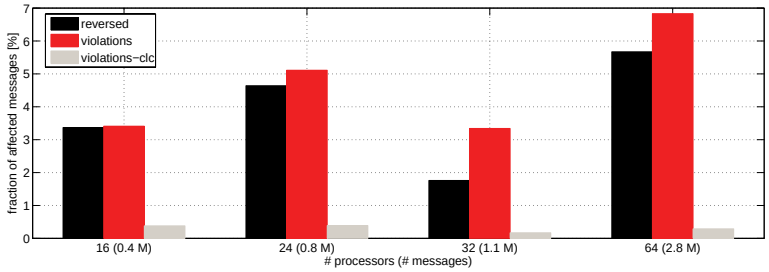


Figure 7.9: Percentage of messages with the order of send and receive events being reversed, of messages with clock condition violations, and of clock condition violations explicitly corrected by the CLC algorithm during forward amortization on the NGS grid.

per process with five solver iterations was configured. A longer run was also emulated by inserting sleep statements immediately before and after the main computational phase so that it was carried out ten minutes after initialization and ten minutes before finalization, resulting in an execution time of roughly 20 min.

To provide evidence of the frequency and the extent of clock condition violations, Figure 7.9 shows the percentage of messages with the order of send and receive events being reversed in the original trace, of messages with clock condition violations in the original trace, and of clock condition violations explicitly corrected by the CLC algorithm during forward amortization. Again, the numbers represent averages across three measurements for each configuration. 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 across all configurations was 6.72 μs.

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 and thus the relative deviation of event position and event distance is again determined. 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, Table 7.9 shows the relative deviation of the event distance across different numbers of processors. The numbers in individual columns are percentages and are rounded to two digits after the decimal point. They represent the maximum across three measurements. It can be seen that larger deviations are still possible in spite of very small averages. The results given in Table 7.9 indicate that larger deviations are rare and that their influence on performance analysis results will usually be negligible, as in the cases previously discussed.

Table 7.9: SMG2000: Relative deviation of the event distance on the NGS grid. Unless indicated numbers are given in percent and rounded to two digits after the decimal point.

	NGS grid			
# CPUs	16	24	32	64
# at Leeds	8	8	16	32
# at Manchester	8	16	16	32
Average	0.00	0.00	0.00	0.00
Maximum	110.88	20.23	61.83	160.13
Percentage of intervals with deviation above threshold				
$>$ 0.0%	68.51	58.01	60.23	80.48
$>$ 0.01%	0.43	0.98	0.28	0.58
$>$ 0.1%	0.42	0.98	0.27	0.57
$>$ 1%	0.03	0.05	0.03	0.08
$>$ 10%	0.00	0.00	0.00	0.00
$>$ 100%	0.00	0.00	0.00	0.00
Percentage of execution time consumed by intervals with deviation above threshold				
$>$ 0.0%	66.65	54.75	39.52	74.97
$>$ 0.01%	0.76	0.50	0.36	0.98
$>$ 0.1%	0.16	0.10	0.07	0.11
$>$ 1%	0.01	0.00	0.00	0.00
$>$ 10%	0.00	0.00	0.00	0.00
$>$ 100%	0.00	0.00	0.00	0.00

The runtime behavior of the parallel CLC algorithm was also evaluated on the NGS grid. Figure 7.10 presents scaling results of the parallel timestamp synchronization, the Scalasca wait-state analysis, the total analysis, and the SMG2000 benchmark itself. The results confirm our previous observation that the wait-state analysis, the parallel timestamp synchronization, and the execution of the SMG2000 benchmark itself exhibit roughly equivalent scaling behavior – again a result of the replay-based nature of the two analysis mechanisms and the communication-bound performance characteristics of SMG2000. Finally, we can conclude that the extended and parallelized CLC algorithm is suitable to run efficiently in a metacomputing environment.

7.4 Summary

This chapter presented an evaluation of the accuracy and scalability of the parallel controlled logical clock algorithm when applied to traces of parallel programs running on a range of cluster systems including metacomputing environments. The accuracy was investigated by

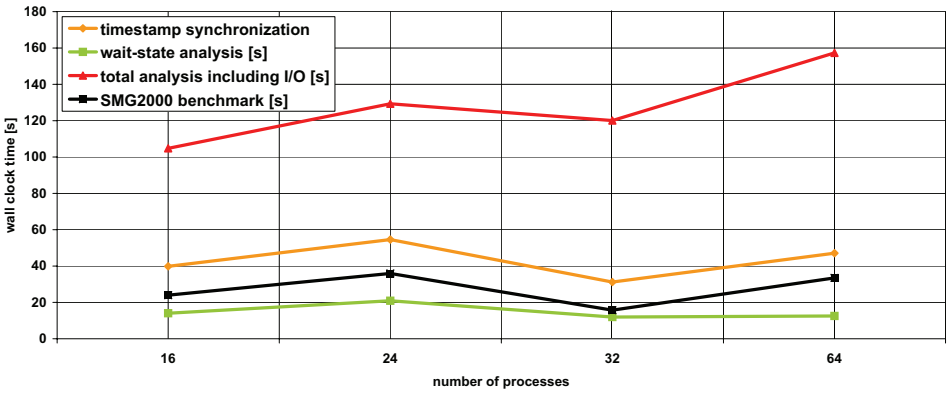


Figure 7.10: Scalability of the parallel timestamp synchronization and SMG2000 benchmark on the NGS grid.

determining the extent to which the length of local intervals was preserved. Two different interval types were considered: (i) intervals between an event and the first event of the same process (i.e., event position) and (ii) intervals between adjacent process-local events (i.e., event distance). The scaling behavior of the correction algorithm was examined by comparing it with the original target application. This indicates the quality of the scaling behavior because the replay-based processing scheme is intended to mimic the original communication behavior.

As the results demonstrate, the algorithm eliminates inconsistent timings of concurrent events while only marginally changing the length of intervals between local events – even if wide-area communication is involved. Traces of hybrid parallel programs that use MPI and OpenMP in combination only contained clock condition violations in message-passing event semantics. However, to preserve the logical event order, shared-memory event semantics were temporarily violated and subsequently restored by the CLC algorithm. While a pure message-passing synchronization that does not account for shared-memory event semantics would leave these semantics violated, the CLC algorithm recognizes such situations and preserves the correct order of shared-memory events in the synchronized event stream. Moreover, the parallel version of the algorithm easily scales to 4,096 application processors and shows potential for even larger configurations. Thus, the extended and parallelized CLC algorithm is an accurate and scalable option for removing inconsistencies in event traces.

Chapter 8

Summary and Outlook

World-wide efforts to build machines with performance levels in the petaflops range acknowledge that supercomputers are indispensable to satisfy the requirements of many applications in science and engineering. Driven by the availability of inexpensive commodity components produced in large quantities, clusters now represent the majority of supercomputing systems, exhibiting a vast diversity in terms of architecture, interconnect technology, and software environment. As the current trend in microprocessor development continues, computer architects are realizing further performance gains by using larger numbers of moderately fast processor cores rather than by further increasing the speed of uni-processors. Therefore, supercomputer applications are being required to harness much higher degrees of parallelism in order to satisfy their growing demand for computing power.

In order to effectively utilize these complex large-scale computer systems, scientists and engineers need powerful and robust performance-analysis tools. Such tools not only help to improve the scalability characteristics of scientific codes and thus expand their potential, but also allow domain experts to concentrate on the science underneath rather than to spend a major fraction of their time debugging their code and tuning it for a particular machine. However, to cope with cross-cluster diversity, tool developers can make only little assumptions with respect to the availability of non-standard features such as a global clock.

Event tracing is one frequently used technique by software tools with a broad spectrum of applications ranging from performance analysis [90], performance modeling [60] and prediction [82] to debugging [53]. In particular, event traces are helpful in understanding the performance behavior of parallel programs since they allow the in-depth analysis of communication and synchronization patterns. For instance, the Scalasca toolset scans event traces of parallel applications for wait states that occur when processes fail to reach synchronization points in a timely manner. Usually, events are recorded along with the time of their occurrence to measure the temporal distance between them and/or to analyze their relative order. Typical events being recorded include entering or leaving functions, sending or receiving messages, and events related to collective communication or synchronization. Event traces are frequently used to analyze MPI or hybrid MPI/OpenMP codes.

As shown in a study conducted as part of this thesis, measuring the time between concurrent events necessitates either a global clock or well-synchronized processor-local clocks because the accuracy of trace analyses depends on the comparability of timestamps taken on different processors and may be adversely affected by non-synchronized clocks leading to inaccurate relative event timings. Such inaccuracies may cause a given interval to appear shorter or

longer than it actually was, or introduce violations of the logical event order, which requires a message to be received only after it has been sent. Inconsistent trace data may not only lead to false conclusions, for instance, when the impact of communication patterns is quantified, but may also confuse the user of trace-visualization tools by causing message arrows to point backward in time-line views. Even more strikingly, trace-analysis tools, like KOJAK, may also cease to work in a satisfactory manner if they rely on the correct order to function properly.

While some custom-built clusters such as IBM Blue Gene [54] offer relatively accurate global clocks, most clusters provide only processor-local 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 [67]. Assuming that potentially different drifts of local clocks remain constant over time, linear offset interpolation can be applied to map local onto global timestamps. Since software clocks such as `MPI_Wtime()` or `gettimeofday()` often leverage network synchronization via NTP, which can lead to sudden drift adjustments, hardware clocks such as IBM's time base register have been identified in this thesis as alternatives with at least approximately constant clock drifts. However, as this thesis revealed, even those suffer from drift deviations that may compromise the accuracy of linear offset interpolation – especially when the application runs longer than a few minutes. As a consequence, although linear offset interpolation can restore the consistency of the trace data to some degree, many traces of MPI applications spanning multiple nodes of a cluster system may exhibit inaccurate relative event timestamps harming further analyses.

As the errors of single timestamps are hard to assess, violations of the logical event order can be easily detected and offer a toehold to increase the fidelity of inter-process timings. The controlled logical clock (CLC) algorithm [79, 80] developed by Rolf Rabenseifner accounts for such violations in point-to-point communication by shifting message events in time as much as needed while trying to preserve the length of local intervals. The algorithm restores the logical event order using happened-before relations among point-to-point communication events. This algorithm is, however, not suitable for many parallel applications because (i) it ignores collective and shared-memory communication and (ii) as a serial algorithm it offers only limited scalability. Additionally, while correcting point-to-point event semantics, the algorithm may also introduce new violations, because it does not consider the logical order of MPI collective events and OpenMP events. This thesis has addressed these shortcomings by extending the algorithm to restore and preserve event semantics related to collective and shared-memory operations and by parallelizing the extended version to make it suitable for large-scale systems including computational grids.

Given that a happened-before relation between two events can be modeled as an exchange of a logical message between both events, the basic idea behind the semantic extension is to consider collective message-passing and shared-memory operations as being composed of multiple logical point-to-point messages. Taking the semantics of the different flavors of collective message-passing (e.g., *1-to-N*, *N-to-1*) and shared-memory operations (e.g., *team creation*, *team termination*) into account, the extensions allow the logical event order among constituent events of these operations to be preserved and restored.

In order to accomplish the timestamp synchronization in a scalable way, both distributed memory and parallel processing capabilities available on the target system are exploited by

processing separate local event traces in parallel and replaying the original communication on as many CPUs as were used to execute the target application itself. The central idea behind the replay is to reenact the target application's communication based on the trace information so that each communication operation is again executed. Hence, the timestamp synchronization is a parallel program having as many processes and threads as the target application that generated the trace data, resulting in a one-to-one mapping of target application and timestamp synchronization processes and threads. As the algorithm needs event data at the sender and receiver side of the original communication, both a parallel forward and a parallel backward replay are used to exchange data among communicating peers involved on either side.

The extended and parallelized algorithm is also well suited for computational grids because (i) it accounts for the hierarchy of latencies found in these systems and (ii) its distributed memory and processing scheme establishes a global view of the trace data in the absence of a global file system. In addition, this thesis defines the necessary infrastructure to accurately measure clock offsets across a metacomputer with a hierarchy of latencies between its various, geographically dispersed nodes.

The algorithm was integrated into the Scalasca trace-analysis framework. To demonstrate the methodology in real supercomputing scenarios, the extended and parallelized algorithm was applied to traces of realistic parallel programs taken on a range of cluster systems including computational grids. As the results demonstrate, the algorithm eliminates inconsistent timings of concurrent events while only marginally changing the length of intervals between local events – even if wide-area communication is involved. Moreover, the parallel version of the algorithm easily scales to 4,096 application processors and shows potential for even larger configurations.

The main accomplishments and findings of this thesis can be summarized as follows:

- This thesis has shown that linear offset interpolation is insufficient to guarantee the consistency of the logical event order in event traces.
- The CLC algorithm has been extended to synchronize the timestamps of MPI collective events, which allows the correction of traces from a much broader range of MPI applications.
- The CLC algorithm has been extended to synchronize the timestamps of OpenMP shared-memory events, which – in combination with the previous extension – allows the correction of traces from hybrid applications.
- The CLC algorithm has been parallelized to make it suitable for traces from large-scale applications.
- The thesis has defined the necessary infrastructure to record, synchronize, and automatically analyze traces in computational grids.
- It has been validated that the new version of the algorithm only marginally changes the length of intervals between local events – even if wide-area communication is involved.
- The scalability of the extended and parallelized CLC algorithm has been demonstrated with up to 4,096 application processes.

Future enhancements should aim at further improving both the functionality and scalability of the parallel CLC algorithm. Currently, offset values among participating clocks are measured at initialization and finalization and are subsequently used as parameters of the linear correction function. Not to perturb the program, offset measurements in between are avoided. However, Doleschal et al. [29] propose periodic offset measurements during global synchronization operations while limiting the effort required in each step by resorting to indirect measurements across several hops. More precisely, clock offset are periodically measured at global synchronization points while the target application is running. In order to reduce the measurement overhead, clock offsets are measured pair-wise using a communication scheme given as a bipartite, regular graph containing a Hamilton cycle. In this way, all clock offsets can be determined – either directly or indirectly. These offsets are subsequently used as parameters of a piece-wise linear interpolation function. Apparently, larger temporal distances between offset measurements decrease the measurement overhead but also the accuracy of the piece-wise linear interpolation. The controlled logical clock may be used to remove residual inconsistencies between such periodic offset measurements. Hence, a combined method of periodic offset measurements and CLC algorithm is desirable.

Finally, although the extended version of the algorithm only needs information about the respective event semantics (e.g., root sends to all other processes), the accuracy of the model could be improved if the MPI-internal messaging inside collective operations was exposed using interfaces such as PERUSE [1]. In this case, the decomposition into (additional) send and receive events would be given naturally. This may be especially useful for synchronizing collective MPI operations executed on large processor counts.

References

[1] MPI PERUSE - An MPI extension for revealing unexposed implementation information. `www.mpi-peruse.org`.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Joint Computer Conferences*, pages 483–485, Atlantic City, NJ, USA, 1967. ACM.

[3] ASC - Advanced Simulation and Computing. ASC Sequoia Benchmark Codes, May 2009. `https://asc.llnl.gov/sequoia/benchmarks`.

[4] Özalp Babaoğlu and Rogério Drummond. (Almost) no cost clock synchronization. Technical Report TR86-791, Cornell University, 1986.

[5] Joshua E. Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[6] Daniel Becker, Wolfgang Frings, and Felix Wolf. Performance evaluation and optimization of parallel grid computing applications. In *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 193–199, Toulouse, France, February 2008. IEEE Computer Society Press.

[7] Daniel Becker, John C. Linford, Rolf Rabenseifner, and Felix Wolf. Replay-based synchronization of timestamps in event traces of massively parallel applications. In *Proceedings of the International Conference on Parallel Processing - Workshops, 1st International Workshop on Simulation and Modelling in Emergent Computational Systems*, pages 212–219, Portland, OR, September 2008. IEEE Computer Society Press.

[8] Daniel Becker, Rolf Rabenseifner, and Felix Wolf. Timestamp synchronization for event traces of large-scale message-passing applications. In *Proceedings of the 14th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science 4757, pages 315–325, Paris, France, September-October 2007. Springer.

[9] Daniel Becker, Rolf Rabenseifner, and Felix Wolf. Implications of non-constant clock drifts for the timestamps of concurrent events. In *Proceedings of the IEEE Cluster Conference*, pages 59–68, Tsukuba, Japan, September 2008. IEEE Computer Society Press.

[10] Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John C. Linford. Replay-based synchronization of timestamps in event traces of massively parallel applications. *Scalable Computing: Practice and Experience, Special Issue: Simulation in Emergent Computational Systems*, 10(1):46 – 60, March 2009.

[11] Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John C. Linford. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Computing, Special Issue EuroPVM/MPI 2007*, 35(12):595–607, December 2009.

[12] Daniel Becker, Felix Wolf, Wolfgang Frings, Markus Geimer, Brian J. N. Wylie, and Bernd Mohr. Automatic trace-based performance analysis of metacomputing applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Long Beach, CA, March 2007. IEEE Computer Society Press.

[13] Marina Biberstein, Yuval Harel, and Andre Heilper. Clock synchronization in Cell BE traces. In *Proceedings of the 14th Euro-Par Conference*, Lecture Notes in Computer Science 5168, pages 3–12, Las Palmas de Gran Canaria, Spain, August - September 2008. Springer.

[14] Boris Bierbaum, Carsten Clauss, Thomas Eickermann, Lidia Kirtchakova, Arnold Krechel, Stephan Springstubbe, Oliver Wäldrich, and Wolfgang Ziegler. Orchestration of distributed MPI-applications in a UNICORE-based grid with MetaMPICH and metascheduling. In *Proceesings of the 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science 4192, pages 174–183, Bonn, Germany, September 2006. Springer.

[15] Boris Bierbaum, Carsten Clauss, Martin Pöppe, Stefan Lankes, and Thomas Bemmerl. The new multidevice architecture of MetaMPICH in the context of other approaches to grid-enabled MPI. In *Proceesings of the 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science 4192, pages 184–193, Bonn, Germany, September 2006. Springer.

[16] BMBF (Ministry for Education and Research). *Vertically Integrated Optical Testbed for Large Applications in DFN (VIOLA)*. `www.viola-testbed.de`.

[17] Peter N. Brown, Robert D. Falgout, and Jim E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000.

[18] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Phil Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[19] Holger Brunst and Wolfgang Nagel. Scalable performance analysis of parallel systems: Concepts and experiences. In *Proceedings of the International Conference on Parallel Computing*, pages 737–744, Dresden, Germany, 2003. Elsevier Science Publishers B. V., Amsterdam.

[20] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, November 2000.

[21] Cell Broadband Engine resource center, May 2009. `www.ibm.com/developerworks/power/cell`.

[22] Anthony Chan, William Gropp, and Ewing Lusk. Scalable log files for parallel program trace data (draft). Technical Report ANL/MCS-TM-249, Argonne National Laboratory, 2000.

[23] Barbara Chapman, Gabriele Jost, Ruud van der Pas, and David J. Kuck. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, illus. edition, 2007.

[24] Intel Corporation. Cluster OpenMP user's guide rev 1.3., May 2005. `http://cache-www.intel.com/cd/00/00/32/91/329148_329148.pdf`.

[25] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[26] Janice E. Cuny, Alfred A. Hough, and Joydip Kundu. Logical time in visualizations produced by parallel programs. In *Proceedings of the 3rd Conference on Visualization*, pages 186–193, Boston, MA, USA, October 1992. IEEE Computer Society Press.

[27] Luiz De Rose, Ted Hoover Jr., and Jeffrey K. Hollingsworth. The dynamic probe class library – an infrastructure for developing instrumentation for performance tools. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, Washington, DC, USA, April 2001. IEEE Computer Society.

[28] Gerardus J.W. van Dijk and Johannes van der Wal. Partial ordering of synchronization events for distributed debugging in tightly-coupled multiprocessor systems. In *Proceedings of the 2nd European Conference on Distributed Memory Computing*, Lecture Notes in Computer Science 487, pages 100–109, Munich, Germany, April 1991. Springer.

[29] Jens Doleschal, Andreas Knüpfer, Matthias S. Müller, and Wolfgang Nagel. Internal timer synchronization for parallel event tracing. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science 5205, pages 202–209, Dublin, Ireland, September 2008. Springer.

[30] Jack Dongarra, Thomas Sterling, Horst Simon, and Erich Strohmaier. High Performance Computing: Clusters, Constellations, MPPs, and Future Directions. *Computing in Science and Engineering*, 7(2):51–59, March-April 2005.

[31] Antonio J. Dorta, Casiano Rodriguez, Francisco de Sande, and Arturo Gonzalez-Escribano. The OpenMP source code repository. In *Proceedings of the 13th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 244–250, Lugano, Switzerland, February 2005. IEEE Computer Society Press.

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

[33] Andrzej Duda, Gilbert Harrus, Yoram Haddad, and Guy Bernard. Estimating global time in distributed systems. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 299–306, Berlin, Germany, September 1987. IEEE Computer Society Press.

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

[35] Dennis Edwards and Phil Kearns. DTVS: A distributed trace visualization system. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, pages 281–288, Dallas, TX, USA, October 1994. IEEE Computer Society Press.

[36] Y. Etsion and D. Feitelson. Time Stamp Counters Library - Measurements with Nano Seconds Resolution. Technical Report 2000-36, The Hebrew University of Jerusalem, August 2000.

[37] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. *Australian Computer Science Communications*, 10(1):56–66, February 1988.

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

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

[40] Ian T. Foster. Globus toolkit version 4: Software for service-oriented systems. In *Proceedings of the International Conference on Network and Parallel Computing*, Lecture Notes in Computer Science 3779, pages 2–13, Tokyo, Japan, 2006. Springer.

[41] Free Software Foundation. GCC 4.3.2 manual – options for code generation conventions. `http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Code-Gen-Options.html`, 2008.

[42] Markus Geimer, Sameer S. Shende, Allen D. Malony, and Felix Wolf. A generic and configurable source-code instrumentation component. In *Proceedings of the International Conference on Computational Science*, Lecture Notes in Computer Science 5545, pages 696–705, Baton Rouge, LA, USA, 2009. Springer.

[43] Markus Geimer, Felix Wolf, Andreas Knüpfer, Bernd Mohr, and Brian J. N. Wylie. A parallel trace-data interface for scalable performance analysis. In *Proceedings of the Workshop on State-of-the-Art in Scientific and Parallel Computing*, Lecture Notes in Computer Science 4699, pages 398–408, Umeå, Sweden, June 2006. Springer.

[44] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science 4192, pages 303–312, Bonn, Germany, September 2006. Springer.

[45] Markus Geimer, Felix Wolf, Brian J.N. Wylie, and Bernd Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing*, 35(7):375–388, July 2009.

[46] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the 21st Hawaii International Conference on System Sciences*, pages 166–175, Kailua-Kona, HI, USA, 1988. IEEE Computer Society Press.

[47] Sven Haubold, Hartmut Mix, Wolfgang Nagel, and Mathilde Romberg. The UNICORE grid and its options for performance analysis. In *Performance analysis and grid computing*, pages 275–288, Norwell, MA, USA, 2004. Kluwer Academic Publishers.

[48] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 4 edition, 2006.

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

[50] Richard Hofmann. Gemeinsame Zeitskala für lokale Ereignisspuren. In *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen*, pages 333–345, Aachen, September 1993. Springer.

[51] Richard Hofmann. *Gesicherte Zeitbezüge für die Leistungsanalyse in parallelen und verteilten Systemen*. PhD thesis, University of Erlangen, 1993.

[52] Richard Hofmann and Ursula Hilgers. Theory and tool for estimating global time in parallel and distributed systems. In *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, pages 173–179, Madrid, Spain, January 1998. IEEE Computer Society Press.

[53] Simon Huband and Chris McDonald. A Preliminary Topological Debugger for MPI Programs. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 422–429, Brisbane, Australia, 2001. IEEE Computer Society Press.

[54] IBM Blue Gene Team. Overview of the IBM Blue Gene/P project. *IBM Journal of Reserach and Development*, 52(1/2), 2008.

[55] InfiniBand Trade Association, May 2009. `www.infinibandta.org`.

[56] Intel Corporation, May 2009. `www.intel.com`.

[57] Janet: UK's Education and Research Network, May 2009. `www.ja.net`.

[58] Jean-Marc Jézéquel. Building a global time on parallel machines. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 392, pages 136–147, Nice, France, 1989. Springer.

[59] N. Karonis, B. Toonen, , and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.

[60] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. DiP: A parallel program development environment. In *Proceedings of the European Conference on Parallel Computing*, Lecture Notes in Computer Science 1124, pages 665–674, Lyon, France, August 1996. Springer.

[61] LAMMPS. Molecular dynamics simulator, May 2009. `http://lammps.sandia.gov`.

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

[63] Jon Maclaren. HARC: The Highly-Available Resource Co-allocator. In *Proceedings of On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, Lecture Notes in Computer Science 4804, pages 1385–1402, Vilamoura, Portugal, November 2007. Springer.

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

[65] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, October 1989. Elsevier Science Publishers B. V., Amsterdam.

[66] Message Passing Interface Forum. MPI: A message passing interface standard, version 2.1, June 2008. `www.mpi-forum.org/mpi2_1`.

[67] David L. Mills. On the accuracy and stablility of clocks synchronized by the network time protocol in the Internet system. *SIGCOMM Comput. Commun. Rev.*, 20(1):65–75, 1990.

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

[69] Bernd Mohr, Aolan Malony, Sameer Shende, and Felix Wolf. Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128, August 2002.

[70] Myricom, Inc, May 2009. `www.myri.com`.

[71] Wolfgang Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. Vampir: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.

[72] NGS: National Grid Service, May 2009. `www.grid-support.ac.uk`.

[73] Leonid Oliker, Andrew Canning, Jonathan Carter, Costin Iancu, Michael Lijewski, Shoaib Kamil, John Shalf, Hongzhang Shan, Erich Strohmaier, Stéphane Ethier, and Tom Goodale. Scientific application performance on candidate petascale platforms. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1–12, Long Beach, CA, March 2007. IEEE Computer Society Press.

[74] Open MPI. `www.open-mpi.org`.

[75] OpenMP (Open specifications for Multi Processing). `www.openmp.org`.

[76] Susanne Pfalzner and Paul Gibbon. *Many-Body Tree Methods in Physics*. Cambridge University Press, October 1996.

[77] Steve J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, March 1995.

[78] Robert L. Probert, Hualong Yu, and Kassem Saleh. Relative-clock-based specification and test result analysis of distributed systems. In *Proceedings of the 11th Annual International Phoenix Conference on Computers and Communications*, pages 687–694, Scottsdale, AZ, USA, April 1992. IEEE Computer Society Press.

[79] Rolf Rabenseifner. The controlled logical clock - a global time for trace based software monitoring of parallel applications in workstation clusters. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed*, pages 477–484, London, UK, January 1997. IEEE Computer Society Press.

[80] Rolf Rabenseifner. *Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen (The controlled logical clock, a global clock for trace-based monitoring of parallel applications)*. PhD thesis, University of Stuttgart, Department of Computer Science, Stuttgart, March 2000.

[81] Michel Raynal. A distributed algorithm to prevent mutual drift between n logical clocks. *Information Processing Letters*, 24:199–202, 1987.

[82] Germán Rodriguez, Rosa M. Badia, and Jesús Labarta. Generation of simple analytical models for message passing applications. In *Proceedings of the European Conference on Parallel Computing*, Lecture Notes in Computer Science 3149, pages 183–188, Pisa, Italy, August - September 2004. Springer.

[83] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[84] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44 – 52, June 1992.

[85] Standard Performance Evaluation Corporation. SPEC MPI2007 benchmark suite, 2007. `www.spec.org/mpi2007`.

[86] Andrew S. Tannenbaum and Maarten van Steen. *Distributed Systems - Principles and Paradigms*. Prentice-Hall, Inc., international edition, 2002.

[87] The Beowulf Cluster Site, May 2009. `www.beowulf.org`.

[88] Michael L. Van De Vante, Douglass E. Post, and Mary E. Zosel. HPC needs a tool strategy. In *Proceedings of th 2nd International Workshop on Software Engineering for High Performance Computing System Applications*, pages 55–59, St. Louis, MO, USA, May 2005. ACM.

[89] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the Conference on High Performance Networking and Computing*, pages 12–21, Portland, Oregon, United States, 1993. ACM.

[90] Felix Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. PhD thesis, RWTH Aachen, Forschungszentrum Jülich, February 2003. ISBN 3-00-010003-2.

[91] Felix Wolf, Daniel Becker, Markus Geimer, and Brian J. N. Wylie. Scalable performance analysis methods for the next generation of supercomputers. In *Proceedings of the John von Neumann Institute for Computing Symposium*, volume 39 of *NIC-Series*, pages 315–322, Jülich, Germany, February 2008.

[92] Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, Wolfgang Frings, Karl Fürlinger, Markus Geimer, Marc-André Hermanns, Bernd Mohr, Shirley Moore, Matthias Pfeifer, and Zoltan Szebenyi. Usage of the Scalasca toolset for scalable performance analysis of large-scale parallel applications. In *Proceedings of the 2nd HLRS Parallel Tools Workshop*, pages 157–167, Stuttgart, Germany, July 2008. Springer. ISBN 978-3-540-68561-6.

[93] Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, November 2003.

[94] Felix Wolf and Bernd Mohr. EPILOG binary trace-data format, version 1.3. May 2004.

[95] Adam K. L. Wong and Andrzej M. Goscinski. Using an enterprise grid for execution of MPI parallel applications - a case study. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science 4192, Bonn, Germany, September 2006. Springer.

[96] Zhonghua Yang and T. Anthony Marsland. Annotated bibliography on global states and times in distributed systems. *Operating Systems Review*, 27(3):55–74, July 1993.

References

Lucundi acti labores.

CICERO, DE FINIBUS 2

Schriften des Forschungszentrums Jülich
IAS Series

1. **Three-dimensional modelling of soil-plant interactions: Consistent coupling of soil and plant root systems**
   by T. Schröder (2009), VIII, 72 pages
   ISBN: 978-3-89336-576-0
   URN: urn:nbn:de:0001-00505

2. **Large-Scale Simulations of Error-Prone Quantum Computation Devices**
   by D. B. Trieu (2009), VI, 173 pages
   ISBN: 978-3-89336-601-9
   URN: urn:nbn:de:0001-00552

3. **NIC Symposium 2010**
   Proceedings, 24 – 25 February 2010 | Jülich, Germany
   edited by G. Münster, D. Wolf, M. Kremer (2010), V, 395 pages
   ISBN: 978-3-89336-606-4
   URN: urn:nbn:de:0001-2010020108

4. **Timestamp Synchronization of Concurrent Events**
   by D. Becker (2010), XVIII, 116 pages
   ISBN: 978-3-89336-625-5
   URN: urn:nbn:de:0001-2010051916

Supercomputing is a key technological pillar of modern science and engineering, indispensable for solving critical problems of high complexity. However, to effectively utilize today's supercomputing systems, scientists and engineers need powerful and robust software development tools. One technique widely used by such tools is event tracing. Recording time-stamped runtime events in event traces is especially helpful to understand the performance behavior of parallel programs, since it enables the post-mortem analysis of communication and synchronization patterns.

The accuracy of such analyses depends on the comparability of timestamps taken on different processors and may be adversely affected by nonsynchronized clocks. Inconsistent trace data may not only lead to false conclusions and confuse the user of trace-visualization tools but also may break tools if they rely on the correct event order to function properly. Although linear offset interpolation can restore the consistency of the trace data to some degree, time-dependent drifts and other inaccuracies may still disarrange the original succession of events.

The already familiar controlled logical clock algorithm accounts for such violations in point-to-point communication. It is, however, not suitable for realistic applications because it ignores collective and shared-memory communication and – as a serial algorithm – offers only limited scalability. To address these shortcomings, the algorithm was (i) extended such that it also restores the event semantics of collective and shared-memory operations and (ii) parallelized to make it suitable for large-scale systems including computational grids. The extended and parallelized version was evaluated in practice by integrating it into the Scalasca trace-analysis framework and applying it to traces of realistic applications taken on single cluster systems and computational grids.

This publication was written at the Jülich Supercomputing Centre (JSC) which is an integral part of the Institute for Advanced Simulation (IAS). The IAS combines the Jülich simulation sciences and the supercomputer facility in one organizational unit. It includes those parts of the scientific institutes at Forschungszentrum Jülich which use simulation on supercomputers as their main research methodology.

IAS Series
Volume 4
ISBN 978-3-89336-625-5

JÜLICH
FORSCHUNGSZENTRUM