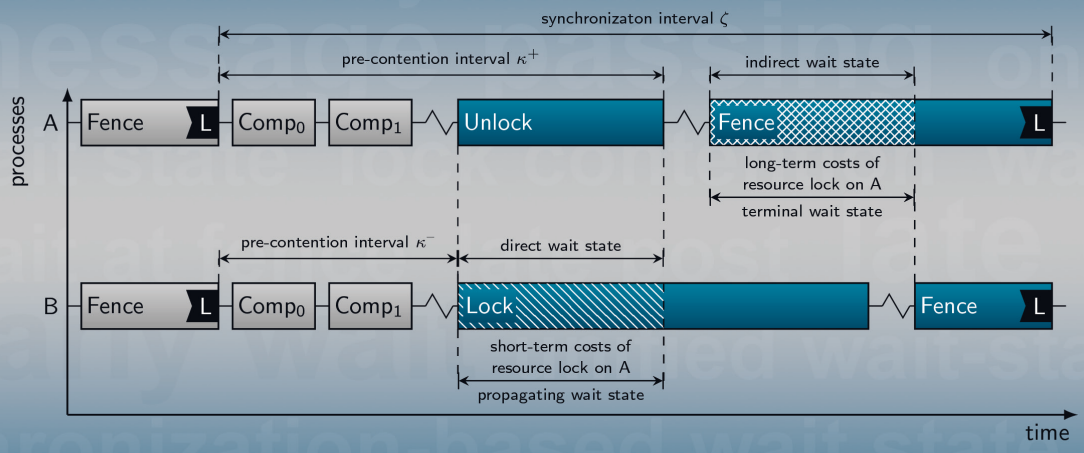


Understanding the formation of wait states in one-sided communication

Marc-André Hermanns

IAS Series
Band / Volume 35
ISBN 978-3-95806-297-9

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

Schriften des Forschungszentrums Jülich
Reihe IAS Band / Volume 35

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

Understanding the formation of wait states in one-sided communication

Marc-André Hermanns

Schriften des Forschungszentrums Jülich
Reihe IAS

Band / Volume 35

ISSN 1868-8489

ISBN 978-3-95806-297-9

Bibliografische Information der Deutschen Nationalbibliothek.
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte Bibliografische Daten
sind im Internet über http://dnb.d-nb.de abrufbar.

Herausgeber und Vertrieb:	Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag 52425 Jülich Tel.: +49 2461 61-5368 Fax: +49 2461 61-6103 zb-publikation@fz-juelich.de www.fz-juelich.de/zb
Umschlaggestaltung:	Grafische Medien, Forschungszentrum Jülich GmbH
Druck:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2018

Schriften des Forschungszentrums Jülich
Reihe IAS, Band / Volume 35

D 82 (Diss., RWTH Aachen University, 2017)

ISSN 1868-8489
ISBN 978-3-95806-297-9

Persistent Identifier: urn:nbn:de:0001-2018012504

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

Abstract

Due to the available concurrency in modern-day supercomputers, the complexity of developing efficient parallel applications for these platforms has grown rapidly in the last years. Many applications use message passing for parallelization, offering three main communication paradigms: point-to-point, collective and one-sided communication. Each paradigm fits certain domains of algorithms and communication patterns best. The one-sided paradigm decouples communication and synchronization and allows a single process to define a complete communication. These are important features for runtime systems of new programming paradigms and state-of-the-art dynamic load-balancing strategies. In any process interaction, wait states can occur, where a process is waiting for another—idling—before it proceeds with its local computation. To eliminate such wait states, runtime and application developers alike need support in detecting and quantifying them and their root causes. However, tool support for identifying complex wait states in one-sided communication is scarce. This thesis contributes novel methods for the scalable detection and quantification of wait states in one-sided communication, the automatic identification of their root causes, and the assessment of optimization potential.

The methods for wait-state detection and quantification, as introduced by Böhme et al. and extended by this thesis, build upon a parallel post-mortem traversal of process-local event traces, modeling an application's runtime behavior. Performance-relevant data is exchanged just in time on the recorded communication paths. Through the nature of one-sided communication, information on such communication paths is not available on all processes involved, impeding the use of this original approach for one-sided communication. The use of a novel high-level messaging framework enables the exchange of messages on the implicit communication paths of one-sided communication, while retaining the scalability of the original approach. This enables the identification of previously unstudied types of wait states unique to one-sided communication: lack of remote progress and resource contention. Beyond simple accounting of waiting time, other contributed methods allow pinpointing root causes of such wait states and identifying optimization potential in one-sided applications. Furthermore, they distinguish two fundamentally different classes of wait-state root causes: delays for direct process synchronization (similar to point-to-point and collective communication) and contention in case of lock-based process synchronization, whose resolution strategies are diametrically opposed to each other. Finally, the contributed methods enable the identification of the longest wait-state-free execution path (i.e., critical path) in parallel applications using one-sided communication. As only optimization of functions on the critical path will yield performance improvements, its identification is key to choosing promising optimization targets.

All of these methods are integrated into the Scalasca performance toolset. Their scalability and effectiveness are demonstrated by evaluating a variety of applications using one-sided communication interfaces running in configurations with up to 65,536 processes.

Zusammenfassung

Aufgrund der Nebenläufigkeit in modernen Supercomputern hat die Komplexität effiziente parallele Programme zu entwickeln, in den letzten Jahren rapide zugenommen. Eine Vielzahl von Programmen nutzt „Message Passing“ zur Parallelisierung, welches drei Kommunikationsparadigmen bereitstellt: Punkt-zu-Punkt, kollektive und einseitige Kommunikation. Jedes dieser Paradigmen eignet sich für eine spezifische Klasse von Algorithmen. Einseitige Kommunikation entkoppelt die Kommunikation von ihrer Synchronisation und erlaubt es, alle Kommunikationparameter auf einem Prozess zu definieren. Dies ist essenziell für Laufzeitumgebungen neuer Programmierparadigmen und bestimmte Strategien zum dynamischen Lastausgleich. In jeder Interaktion zwischen Prozessen können potentiell Wartezeiten entstehen, wo ein Prozess auf einen anderen wartet bevor er seine Berechnungen fortführen kann. Um solche Wartezeiten zu eliminieren, benötigen Entwickler von Laufzeitumgebungen und Simulationen Unterstützung bei der Erkennung und Quantifizierung der Wartezeiten und ihrer Ursachen. Die bestehende Unterstützung von einseitiger Kommunikation in Werkzeugen zur Leistungsanalyse ist nicht ausreichend. Diese Dissertation beschreibt neue Methoden zur skalierbaren Erkennung und Quantifizierung von Wartezeiten in einseitiger Kommunikation und ihrer Ursachen sowie des Optimierungspotentials.

Die beschriebenen Methoden bauen auf eine nach der Messung durchgeführte, parallele Traversierung von Prozess-lokalen Ereignisspuren auf, die das Laufzeitverhalten des Programms modellieren. Leistungsrelevante Daten werden bedarfsorientiert auf den aufgezeichneten Kommunikationspfaden ausgetauscht. Durch die inhärenten Eigenschaften einseitiger Kommunikation steht die Information über die Kommunikationspfade nach der Messung nicht auf allen beteiligten Prozessen zur Verfügung, was die direkte Verwendung des ursprünglichen Ansatzes für die Analyse von einseitiger Kommunikation erschwert. Eine neue, komplementäre Kommunikations-Infrastruktur ermöglicht den Austausch von Nachrichten auf impliziten Kommunikationswegen, während die Skalierbarkeit des ursprünglichen Ansatzes bewahrt bleibt. Dies ermöglicht die Identifikation bisher nicht untersuchter Typen von Wartezeiten wie sie in einseitiger Kommunikation auftreten: Fehlender Kommunikationsfortschritt und Konflikte beim Resourcenzugriff. Jenseits reiner Berechnung von Wartezeiten erweitern die beschriebene Methoden die Ortung von Ursachen der Wartezeiten und die Abschätzung des Optimierungspotentials für Programme mit einseitiger Kommunikation. Dabei wird zwischen zwei fundamental unterschiedlichen Klassen von Ursachen für Wartezeiten unterschieden: Verzögerungen bei direkter Prozesssynchronization, wie sie in der Punkt-zu-Punkt und kollektiven Kommunikation auftreten, und Zugriffskonflikte, wie sie in Lock-basierter Prozesssynchronization auftreten. Die Strategien zur Auflösung dieser beiden Klassen sind genau entgegengesetzt. Des weiteren ermöglichen die beschriebenen Methoden die Identifikation des kritischen Pfades in parallelen Programmen mit einseitiger Kommunikation. Die Identifizierung von Funktionen auf dem kritischen Pfad ist eine Grundvoraussetzung für die Wahl geeigneter Optimierungskandidaten.

Alle Methoden wurden im Scalasca Trace Analyzer implementiert. Ihre Skalierbarkeit und Effektivität wird Anhand von verschiedenen parallelen Programmen mit einseitiger Kommunikation auf bis zu 65,536 Prozessoren demonstriert.

Acknowledgement

During the course of my dissertation I have met many people in the high-performance computing community and especially the performance analysis community. These people have shaped my understanding of this field and also contributed to the development of my personality. As such, I am grateful for the many-faceted support I received.

I owe a dept of gratitude to Prof. Dr. Felix Wolf for his ample advice and encouragement as my primary adviser during my work on this dissertation that was a major influence on its positive outcome.

Moreover, I am grateful to Prof. Dr. Matthias Müller for his advice and support, serving as a referee for this thesis, and granting me a second scientific home at the IT Center of RWTH Aachen University. Furthermore, I thank Prof. Dr. Erika Ábrahám and Prof. Dr. Jan Borchers for serving on my exam committee.

Thanks are due to Dr. Rüdiger Esser and Dr. Norbert Attig for their support during my time at the Jülich Supercomputing Centre, enabling me to start and end this project in an extraordinary work environment.

I am grateful to Dr. Bernd Mohr for him being a steady source of support for more than thirteen years since I first started to work in the field of high-performance computing and performance analysis. I also thank my colleagues at Forschungszentrum Jülich, German Research School for Simulation Sciences, and RWTH Aachen University for their inspiring feedback and ideas over the years. Special thanks are due to Dr. Markus Geimer for acting as a sounding board for many of my ideas, commenting on early versions of my thesis, and for supporting me through the ups and downs of the process. Moreover, many thanks to Dr. David Böhme for our discussions on his original wait-state formation model that laid the foundation of my own work on the unified wait-state formation model. I thank Dr. Sriram Krishnamoorthy of Pacific Northwestern National Laboratory for the insightful discussions on ARMCI internals and great partnership during our joint work.

Furthermore, I thank Nadine Daivandy for designing the title page and Dr. Beatrice Hermanns for her excellent work on copy-editing my manuscript.

Finally, I thank my parents for all their love and support throughout my life. Moreover, this work would not have been possible without the unwavering love and support of my wife Annette and my daughters Ira and Kara.

Contents

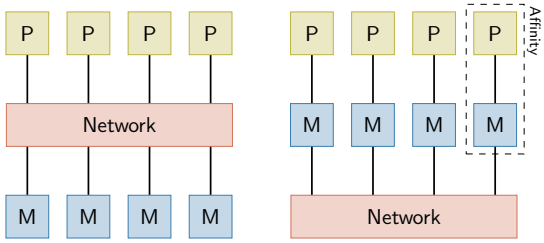
List of Figures

List of Tables

1. Introduction

In the past decades, scientific simulation has established itself, next to theory and experiment, as the third pillar of research. Scientists can use simulations to verify theories where experiments would prove to be too difficult, dangerous, or expensive. However, the persistent urge to improve scope and detail of such simulations lead to a great demand in computational power. High-performance computing (HPC) has been the foundation for scientific simulation for many years, striving to meet the simulation's demand in such power. From its very inception, the field of high-performance computing has embraced concurrency to increase computing performance. The key idea is that multiple processing elements working together will solve a given task better than a single element. This means the parallel work helps to either solve a given problem faster resulting in an improved *time to solution* or solve larger problems in the same time a single element would solve a smaller problem, increasing *problem resolution*.

The following sections introduce the landscape of high-performance computing. They discuss parallel system architectures as well as one-sided communication in the context of programming models in high-performance computing. Furthermore, they present methods for the performance analysis of parallel applications in HPC environments and examine the support of existing tools for the automatic identification of complex performance metrics such as wait states and their root causes in complex behavioral patterns of applications using one-sided communications.



(a) Uniform shared memory access (UMA).

(b) Non-uniform shared memory access (NUMA).

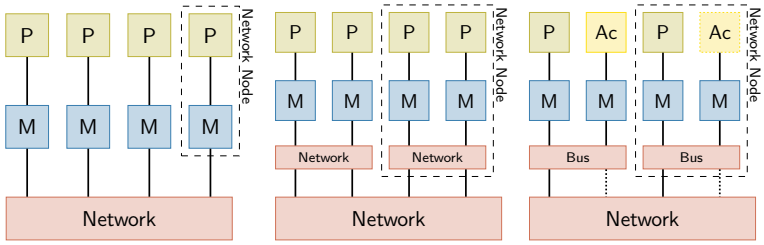
Figure 1.1.: Different variants of shared-memory architectures. All processors have access to a global shared memory. (Based on [128])

1.1. Parallel system architectures

Hardware architectures can be classified according to how they express parallelism and concurrency to the software, especially how parallel software can access the available memory. Systems where processing elements share access to a single global memory are classified as *shared-memory architectures*. In contrast, systems where processing elements only have access to parts of the memory globally available are classified as *distributed-memory architectures*.

Shared-memory architectures. Shared-memory architectures allow multiple processing elements to access the full system memory. The most prominent form of shared-memory architectures today is the symmetric multi-processor (SMP). It describes two or more identical processors accessing a global shared-memory space. In the past decades, the processing elements in SMPs have undergone a transition from uni-core to multi-core CPUs. This transition is based on the continuously growing integration of transistors on a single chip. In 1965, Gordon Moore published an article on the evolution of transistor density in processors, which became known as Moore's Law [101]. It observes the doubling of the number of transistors per processor roughly every two years. This rate has been sustained over many decades since its initial publication, although in 2010 the International Technology Roadmap for Semiconductors [10] updated this observation to a doubling of transistors every three years and predicted this rate to continue until 2020. Initially, the chips used the extra number of transistors to integrate more complex hardware to improve the instruction rate of a single instruction stream. Additionally, the clock rate increased, enabling more instructions per second. Such improvements directly benefit any application without modifying the application itself. Around 2002, it became evident that further optimizations for a single instruction stream were no longer feasible, as instruction-level parallelism could not be exploited any further and higher clock rates raised problems with heat dissipation and leakage [154]. Ever since, every new chip generation uses the still unbroken trend towards higher transistor integration densities to integrate more cores into a single chip, increasing the number of parallel instruction streams and fueling concurrency in the shared-memory domain.

Next to uni-core and multi-core processing elements, SMPs can be further distinguished by their costs of accessing the main memory. In *uniform memory access* (UMA) systems the time needed per memory access is independent of both the processing element and the memory location being processed [62]. Such uniformity was achieved in classic multiprocessing environments by connecting all processing elements to the same memory network—the memory bus (Figure 1.1a). With increasing number of processing elements, this bus can become a serious performance bottleneck. In *non-uniform memory access* (NUMA) systems access time depends on the processing element and the memory location being processed [62]. The non-uniformity is usually the result of a different interconnection network among the processors. Memory blocks show affinity to a processing element to which it is local (Figure 1.1b). Accesses to local memory is much faster than accesses to memory associated with another processing element. While this allows for a better scalability with concurrent accesses to different parts of the memory through multiple access paths, it also creates "locality effects" when different paths have different time costs associated with them.



(a) Distributed-memory architecture.

(b) Distributed shared-memory architecture.

(c) Heterogeneous architecture.

Figure 1.2.: Distributed-memory architectures.

Most modern CPU architectures use caches to exploit *locality of reference* [46] in computations. Coherence protocols are needed to ensure data consistency among different copies of data in different caches. For NUMA architectures, implementations of such protocols in software bear a significant overhead. Therefore, NUMA architectures soon provided hardware implementations for these protocols, forming the now prevalent CPU architecture of cache-coherent NUMA (cc-NUMA) systems. In such ccNUMA systems, high-speed hardware interconnects such as Intel's Quickpath™ [71] or HyperTransport™ [69] in AMD's Direct Connect Architecture facilitate the communication among the individual cores. These high speed interconnects can also build the foundation for fast inter-node communication hardware such as Extoll [115] and NumaConnect [136] that enable hardware support for larger global-address space systems [179]. Despite such high-speed interconnects, in practice full hardware-enabled shared-memory access among all processing elements of a cluster still poses serious scalability challenges that currently impede pure shared-memory systems at large scale.

Distributed-memory architectures. In distributed-memory architectures, processes only have access to their local memory. Remote memory can only be accessed by explicit communication over an external network (Figure 1.2a). The time for such accesses is about an order of magnitude higher than memory accesses in shared-memory systems discussed previously. Data needed by multiple processing elements needs to be replicated and/or exchanged explicitly by the software. Data consistency, such as coherency of replicated data, is then taken care of by software. Nevertheless, the scalability of distributed-memory architectures has proven to be better than that of shared-memory systems by several orders of magnitude. While current multi- and many-core systems comprise tens or hundreds of cores, distributed-memory architectures may comprise several tens of thousands of single-core nodes. The nodes of a cluster are connected through a network. As the network is the primary means of communication in the parallel system, its properties (topology, latency, and bandwidth) can be a significant performance factor for a system [8, 45, 102]. Clusters with an extreme number of network nodes—so-called *massively parallel processors* (MPPs)—are often custom-built, employing special low-latency, high-throughput networking devices connected in scalable network topologies, such as a *fat tree* or multi-dimensional *meshes* and *tori* [62].

Hybrid architectures. With the trend to multi-core processors, pure distributed-memory architectures mostly vanished from the high-performance computing landscape. Instead, a system architecture arose that combines both previous architectures, the hybrid *distributed shared-memory architecture* (DSM) as shown in Figure 1.2b. Such systems may comprise several hundreds of thousands or even millions of processing elements, combining parallelism between nodes as well as on the nodes. Examples of such large-scale hybrid architectures are the IBM Blue Gene solution, such as the JUQUEEN system at Jülich Supercomputing Centre, the Mira system at Argonne Leadership Computing Facility, or the Sequoia system at Lawrence Livermore National Laboratory, each comprising between almost half a million and three million cores, or the Cray XE systems, such as the Hector at Edinburgh Parallel Computing Centre or Monte Rosa at the Swiss National Supercomputing Centre with several tens of thousand processing elements.

As power consumption becomes more and more important, large-scale computing platforms currently move to *heterogeneous* systems using accelerators such as general-purpose GPUs (GPGPUs) to take compute load off the host CPU. Such accelerators are many-core chips, with up to several hundred compute cores, using *single instruction multiple threads* (SIMT) [122] computation, which is comparable to *single instruction multiple data* (SIMD) [55]. Each GPU has individual memory separate from the host's memory where data needs to reside during computation. Data either has to be copied explicitly by the user or implicitly by the runtime system between host and GPU memory. As will be discussed in more detail in Section 1.2.1, several software solutions exist to ease such memory handling. However, these architectures are still *multi-stage distributed-memory* systems, with accelerators forming the second stage. Figure 1.2c shows the schematic structure of such systems. While with CUDA GPUDirect RDMA, NVIDIA GPUs can effectively communicate directly with other NVIDIA GPUs across the network (indicated by the dotted line in the figure) [115, 132, 144], such features are not yet generally and widely available for accelerator cards. Within a node, multiple host CPUs can have access to multiple accelerators on the same node. Due to the distinctly different architecture of the processing elements of host and accelerators, such systems are often classified as *heterogeneous* architectures. Examples of such systems are the Cray XK systems, such as the Titan system at Oak Ridge National Laboratory and the Blue Waters system at National Center for Supercomputing Applications, or the one-of-a-kind system Tianhe-1A at the National Supercomputing Center, Tianjin, China. Each of these uses NVIDIA GPGPUs as accelerators.

However, accelerators do not necessarily have to be GPGPUs. Other heterogeneous systems use Intel's latest Xeon Phi [74] processor. It offers the advantages of accelerators in terms of energy efficiency and offloading of computation for the host, but at the same time it can also be used as a first class citizen in message-passing applications as it comprises up to 61 low-energy x86-compatible compute cores per chip, being able to work as a general-purpose compute element. Example for large systems using Xeon Phi processors is the Stampede system at the Texas Advanced Computing Center and the Tianhe-2 system at the National Super Computer Center in Guangzhou, China, which at the time of this writing held the title of the fastest supercomputer in the world [96]. As heterogeneous systems show very good power efficiency and the costs for GPGPUs are low due to their mass-marketability in the Gaming industry, they are growing to be one of the most popular architectures among the fastest computing systems in the world. However, the different levels of concurrency and their heterogeneity make them challenging

to program. A challenge, that is addressed in several new programming paradigms for such architectures.

1.2. One-sided communication in the context of programming models in HPC

Addressing the multi-faceted parallel architectures in high-performance computing, some programming models—both for shared and distributed memory—support very fine grained control over parallelism to enable highly efficient application code for specific platforms. Other programming models strive to minimize the programmer's burden with the details of low-level parallelism, such as load balancing or data locality strategies, providing a more abstract and logical programming view to the user. The following sections will give a broad overview of programming models in use in the HPC community today, provide details on message passing and one-sided communication in particular, and give a range of potential applications for them.

1.2.1. Introduction to programming models in HPC

To exploit hardware properties optimally, low-level interfaces to parallelism are usually designed very close to their target hardware architecture. Special purpose programming models exist for: (1) multi-core systems, (2) many-core systems, (3) distributed memory system, as well as (4) hybrid systems as outlined in the previous sections.

Programming multi-core systems. Although multi-processing and multi-threading were available to the programming community for some time, their initial use was on time-sharing systems, where programmers using multiple instruction streams better utilize the different units of the processor. In high-performance computing, however, a lot of application threads focus on the use of the floating point unit. Thus, oversubscribing a processor with more instruction streams than hardware threads available often does not yield the desired performance improvements. In fact, the additional overhead of scheduling threads in and out may even decrease overall performance. Therefore, supercomputing systems today schedule and even bind a single thread per core. Multi-threading interfaces started to gain traction with the rise of the multi-core processors, where threading overheads amortize much better with more concurrent threads. Also, with more threads sharing memory, common data needs to be replicated less often.

POSIX Threads (Pthreads) [114] give a very fine-grained control over threads, but may also be complex to program, especially in terms of load balance and thread utilization. Nevertheless, as one of the main threading interfaces on UNIX-style operating systems, HPC programmers also adopted it in their simulation codes. The most prevalent multi-threading interface in HPC, however, is OpenMP [125]. Its interface is based on compiler directives that are inserted right before code regions and describe how the following block should be parallelized, including necessary thread synchronization. The compiler then inserts all the low-level threading calls into the code during compilation. As virtually all compilers available on high-performance computing systems

today support OpenMP, it effectively lowers the bar for non-expert programmers to incrementally parallelize the relevant blocks of a larger simulation code. With the release of the fourth version of the OpenMP standard, it also supports tasks, making it interesting for a broader range of algorithms. Task-based programming has been a shared-memory programming model for many years with prominent libraries such as Cilk [133] and more recently Cilk++ [91], Intel Threading Building Blocks (TBB) [79] and the Habanero family of parallel languages [12, 29, 33]. One of the key concepts for balancing the work load among tasks is work stealing [20]. In work stealing, a thread exposes its own work queue for other threads without work to steal parts of it. This enables a dynamic balancing of work load even in situations of highly irregular initial work load.

Programming many-core systems. Programming GPGPUs and accelerators in general usually involves the use of a special compiler along with the programming interface. NVIDIA provides CUDA [123] as the primary means to program their GPGPUs, next to the more general OpenCL [78]. Both involve explicit compilation of the source code targeting the accelerator. While CUDA is only available for NVIDIA products, OpenCL is a more portable programming model, available for GPGPUs of multiple vendors and other types of accelerators. CUDA provides the GPUDirect programming interface enabling its GPUs to transparently interact with other GPUs on the same node and across the network [132, 144] without involving the host CPU. To allow for many different types of accelerators to be programmable by OpenCL, its interface is much broader and sometime regarded as more complex than that of CUDA. To address the complexity of programming accelerators in orchestration with their host CPUs portably, directive-based programming models, such as HMPP [31], and OpenACC [124], arose. They enable the definition of code blocks to be executed on the accelerator, while leaving the scheduling and data transfers involved to the respective runtime system. Also other directive based programming models, originally developed for distributed computing, such as XcalableMP [90], are also starting to focus on supporting accelerators with XcalableACC [113]. Furthermore, also OpenMP as the de-facto standard for shared-memory programming in HPC, seeks to extend its tasking interface to support accelerators [17].

Programming distributed memory and hybrid systems. In high-performance computing, system architectures with distributed memory have a long tradition. To program such architectures, data has to be explicitly replicated between processes. Such replication is done using messages passed between processes. It is so fundamental to the programming of these architectures that they are also named *message-passing architectures* [128].

The most prevalent interface used for message passing in high-performance computing is the Message Passing Interface (MPI) [104]. Although it enables very fine-grained message exchange through different message-passing paradigms (as discussed later), it can be regarded as a high-level interface at the same time, as it already provides application-level guarantees on message reliability and order. MPI defines three communication paradigms, which will be discussed in more detail in the following sections. All paradigms in common is the specification of explicit communication buffers that define the source and destination buffer.

Similar to Pthreads for shared-memory, message passing is the low-level building block for higher-level programming concepts. In the past years, approaches to enable shared-memory-like programming on distributed memory architectures has gained attention of the HPC community. The partitioned global address space (PGAS) programming model provides a shared-memory view to the programmer, independent of the underlying hardware architecture. As such, developers can use it on pure shared-memory, pure distributed-memory, and distributed shared-memory architectures alike. The programming view provides a single logical global view on memory, accessible from all processes. At the same time, this global view is partitioned, exposing the notion of affinity to the programmer. This can ease programming even on pure shared-memory systems, where non-uniform memory-access affects performance, but programming models such as Pthreads and OpenMP do not expose such information, although for the latter, Schmidl and colleagues recently provided library support to exploit memory affinity [140]. In scenarios in which data exchange through shared memory is not available, a runtime system has to facilitate remote data access. The partitioned global address space programming model can either be implemented as a library, such as Global Arrays [119] or UPC++ [181], as an extension of existing programming languages such as Unified Parallel C (UPC) [32] or Co-array Fortran [121, 134], or as new languages such as Cray Chapel [21], IBM X10 [39, 137], and Habanero Java [33].

1.2.2. The message passing model

In high-performance computing, the prevalent interface for message passing is the Message Passing Interface (MPI) [104]. Communication libraries implementing the MPI standard are available on virtually all HPC platforms, providing maximum application portability across platforms. As each platform may have a different MPI implementation, the performance of an application may, however, vary considerably. Additional to portability, MPI provides high-level abstractions for the communication infrastructure, mostly hiding implementation specifics from the application developer.

All communication in MPI is bound to a communication context called *communicator*. Within a communicator, all processes have a unique id, their *rank*, used for addressing of messages. Communication using one communicator does not interfere with communication using another communicator. In regard to explicit involvement of communicating processes, the interfaces can be distinguished into three different paradigms: (1) point-to-point, (2) collective, and (3) one-sided communication. None of these paradigms by itself is best for all algorithms in programming distributed systems. Instead the individual paradigms complement each other in different communication scenarios, providing the building blocks for implementing distributed algorithms of any kind.

Point-to-point communication. Point-to-point or two-sided communication defines two explicit roles for the communicating processes. The *sender* is the source of the data, while the *receiver* is the destination. Both, sender and receiver, are involved in the communication explicitly. Neither process has any knowledge or influence on the memory location of the data passed in the message on the other side. Different communication protocols (modes) define how the data transfer can be facilitated. In *synchronous* communication mode, the sender will wait for

the receiver to acknowledge readiness for data reception. In *buffered* mode, the message payload is copied to an internal buffer and sent from there. In this way, the communication library can safely return to the application where the original communication buffer can be modified again without affecting the ongoing communication. In *ready* mode, the sender assumes readiness of the receiver and starts sending the data without waiting for prior acknowledgement.

Collective communication. In parallel applications in scientific computing, the processes are often tightly coupled. Data then may need to flow from one process to all other processes (broadcast), from all processes to one process (gather), or all processes need to wait for each other, before any single process can continue (barrier). Additionally to pure communication, some collective communication may allow the combination of both communication and computation in a single call (reduce). Variants of these reductions, where the result is available not just on one but on all participating processes, are frequently used in scientific simulations to evaluate a global stop criterion in iterative methods. All of these collective communication patterns explicitly involve a group of processes. Such communication patterns can also be implemented by the user using point-to-point communication. However, a user-level implementation may not yield best performance across different platforms.

A collective communication interface abstracts from the effective communication pattern to a certain communication task that should be achieved. It therefore decouples its implementation from the user code and enables the use of specific optimizations in the message passing library either in software [161] or hardware [60]. The runtime system can then choose the appropriate communication pattern based on different communication parameters, such as the network layout, distance between communication partners, or message size. This potential for optimization drives more and more communication interfaces and programming frameworks to define a collective communication interface [9, 105, 117].

One-sided communication. The third communication paradigm in message passing is one-sided communication. Whereas in point-to-point communication the sender and receiver are also the source and destination of the data, one-sided communication defines two new roles: (1) the *origin* and (2) the *target*. The roles of origin and target are not defined by the direction of data flow, as they can both be sending and receiving data during a one-sided data transfer. Their role in the communication paradigms is rather defined by which process actively defines the communication parameters and which process provides the memory location the one-sided data transfer manipulates. Furthermore, one-sided communication often decouples the data transfer from the memory and process synchronization, enabling communication primitives with less overhead than comparable point-to-point transfers. This is especially useful for small message transfers, where any overhead has a larger impact on the overall performance.

The origin process is the initiator of the call. It defines all communication parameters, including memory locations on both processes. A one-sided communication involves two buffers, a local one, provided by the origin, and a remote one, provided by the target. One-sided communication can only be employed efficiently when all communication parameters are known to the origin process. The target does not actively participate in the communication other than providing the memory for the origin's operation. The origin can both send (i.e., put) data to or receive

(i.e., get) data from the target. Using hardware support, such as remote direct-memory access (RDMA), the communication may complete without involving the target process CPU at all. Synchronization and with it completion of one-sided operations is decoupled from the actual operation and—depending on the interface semantics—may or may not involve the target. *Active target synchronization* explicitly involves the target in the completion of the accesses. The application on the target side therefore implicitly knows when the origin's accesses have completed. *Passive target synchronization* does not explicitly involve the target and is therefore on the application level agnostic to the origin's access.

The fundamental operations in one-sided communication are *put* and *get*, similar to the *load* and *store* processor instructions for the local memory. Additionally, most one-sided interfaces also define *atomic* operations, allowing reduction operations and more complex operations, such as *fetch-and-increment* and *test-and-set*, all of which perform more complex tasks than simple data transfers. Such atomics are also modeled after local processor instructions, and ease the effort of implementing distributed data structures [66, 89]. As part of the one-sided interface, also these more complex calls don't require an explicit matching call on the target.

To clarify terms used in this thesis, *one-sided* describes the nature of the communication or programming interface in that only a single process defines the full communication. A *remote memory access* (RMA) refers to the specific access of a target's memory buffer through an operation, such as put or get. *Remote direct memory access* (RDMA) refers to hardware support for RMA operations allowing the operation to complete without intervention of the target processing element. In summary, RMA operations have a one-sided interface and may be implemented using RDMA features of the network card.

The MPI interface is not as commonly used for one-sided communication as it is for point-to-point and collective communication. The initial one-sided interface in MPI was defined with MPI 2.0 [103]. Several shortcomings in the initial design made it unsuitable for the efficient implementation of higher-level programming language runtime systems and libraries [28]. Therefore other one-sided interfaces, such as ARMCI [118] and different flavors of SHMEM [37, 43, 131, 178], were able to establish themselves for such uses. In 2012, the MPI 3.0 standard updated its one-sided interface [106] to remedy the initial shortcomings and to provide a portable and efficient interface for such usage scenarios.

Building on the one-sided communication concept, *active messages* [126, 171] can associate functions—so-called *handlers*—with a message, to be executed after its arrival on the remote process. In this way, origin processes may obtain remote-completion information by registering a corresponding handler that sends an acknowledgement message from the target back to the origin once the first message is received completely on the target side. As the origin does not explicitly have to wait for the acknowledgement but instead is notified asynchronously about it, it allows for effective overlap of communication with computation, which may reduce waiting times in the communication. In the past years, active-message libraries became very popular as the low-level programming interface for large-scale communication infrastructures. IBM's Blue Gene/P and Blue Gene/Q use the Deep Computing Messaging Framework (DCMF) [87] and PAMI [40], respectively. Cray uses DMAPP [160] as the user-level interface to their Gemini interconnect [170], and GASNet [27] is the active-message framework used to implement Berkeley's and other's partitioned global address space (PGAS) runtime systems. Such libraries

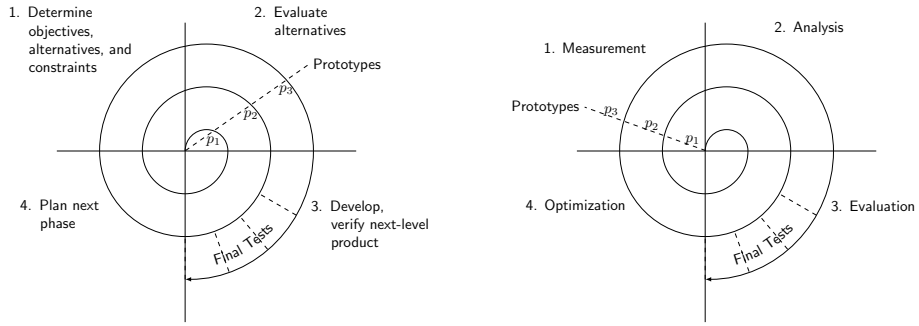
also form the foundation for higher-level programming models such as Charm++ [76] or Active Pebbles [173].

1.2.3. Scope of application of one-sided communication

Depending on the distributed algorithm, developers can use one-sided communication to improve their distributed algorithms, as shown by Siebert and Träff [146] and Sawyer and Mirin [138]. However, the detailed and explicit data exchange of message passing can become complex to program. Chamberlain et al. [34] call this "programming in the *fragmented view*", as the view on the data is fragmented into multiple local parts on the individual processes. They propose *global-view* programming instead. While for shared-memory architectures, this can be achieved easily, distributed-memory architectures still require the explicit data copy. One-sided communication can provide the needed services, enabling higher-level, global view programming models to be employed on large-scale distributed architectures. The most prominent of these are (1) the partitioned global-address space programming model and (2) work stealing in task queues as provided by distributed tasking libraries. Both approaches have gained interest among application developers in the past years, as they enable programming on a higher level of abstraction, where communication is implicitly handled by an underlying runtime system.

Partitioned global-address space. As already outlined in the previous sections, partitioned global address space languages thrive to provide a shared-memory view independent of the underlying hardware architecture. One key characteristic in shared memory programming is that each thread can access shared data without the explicit involvement of another thread. One-sided communication lends itself in such scenarios, as processes can engage in communication independently. If data needs to be exchanged through message passing, only one-sided communication directly provides the required interface to the user. As such, all programming models in global-view programming use one-sided communication in their runtime system to facilitate the remote data exchange. The analysis of the performance of this underlying one-sided communication is therefore extremely important to runtime and compiler writers of those global-view languages.

Task parallelization. With larger system sizes, load imbalances in applications can be the source of significant waiting time, diminishing parallel efficiency. Therefore, the work-stealing scheduling scheme has attracted more interest for large-scale systems since it was successfully implemented for distributed-memory architectures [48, 50, 54, 98]. The foundation of such distributed-memory tasking schemes are one-sided communication or active messages to allow for thieves to steal tasks without explicit target intervention. With such scheduling libraries in place on the host CPU, runtime systems may schedule CUDA [116] and OpenCL [78] tasks also on larger heterogeneous systems with distributed memory.



(a) Simplified form of the spiral model of software engineering [22].

(b) The performance optimization cycle.

Figure 1.3.: Spiral models of software engineering and optimization. Both models are based on an iterative approach.

1.3. Performance analysis of one-sided communication

The brief overview of existing hardware architectures has shown that with growing size, high-performance computing systems can become challenging to use efficiently, as the amount of concurrent process interactions becomes hard to keep track of. With some performance analysis tools, developers can only assess the as-is situation of an application, without classification into good or bad application behavior. A developer then has to rely on expert knowledge to interpret the measurement data whether the recorded behavior needs improvement or not. More advanced performance tools can identify inefficiencies and wait states in recorded application behavior. In a wait state, an application mostly idles without performing any productive work. However, in the complexity of modern supercomputing applications, just knowing the symptom of a problem does not suffice to effectively resolve it, especially as the manual identification of the root cause of the wait states often requires expert knowledge to succeed. It is therefore of utmost importance that tools enable the automatic detection of such causes, codifying the expert knowledge needed for identifying those parts of the code with the highest optimization potential. Furthermore, as waiting times often materialize at points of process interaction, performance tools should enable their identification and the correct quantification of the interaction's effects on other parts of the application and processes. As Van De Vanter and colleagues have pointed out, software development tools, such as performance analysis tools, are of strategic value to the whole high-performance computing infrastructure [167].

1.3.1. Performance optimization cycle

Software performance optimization is not a one-time engagement in the software engineering process. As shown in Figure 1.3, it is quite similar to the spiral model of software engineering [22] and in fact should be integrated into any iterative software development process. Boehm's spiral model distinguishes these phases: (1) determining the objectives, (2) evaluating alternatives,

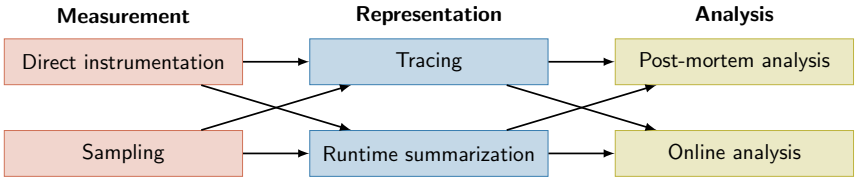


Figure 1.4.: Combinations of different methods in measurement, data representation, and analysis.

(3) developing the code, and (4) planning the next phase. These phases improve the software iteratively, creating several prototypes in the process and eventually leading to the production-ready software package. As shown in Figure 1.3b, the stages of the performance optimization cycle are: (1) measurement, (2) analysis, (3) evaluation of the results, and (4) code optimization. An initial performance measurement may reveal several inefficiencies. After further investigation, manually or automatically supported by a tool, a list of potential optimization targets will emerge. The developer should address the most promising target first. After optimizing a specific part of the code, the developer needs to re-evaluate the application's performance, as the behavior may have changed significantly depending on the optimization, now revealing a different list of further optimization candidates. Given the scale of current supercomputers with up to a million concurrent processes and the resulting complexity of process interaction, automating most of the work in these steps is of prime importance.

Just like there is no single best programming model, there is also hardly a silver bullet in performance evaluation of parallel applications. Therefore, a list of different approaches exist to each of the stages in performance optimization, each with its own advantages and disadvantages. Figure 1.4 shows how performance tools can combine the different methods in the three stages of measurement, data representation, and analysis of performance data.

Measurement

To discuss an application's performance, the developer needs to obtain measurement data. As a broad overview of the application's performance, developers commonly insert timing routines manually into parts of the larger simulation codes to get an overall impression of the application performance [11, 14, 148, 153, 172]. They measure specific metrics, such as the execution time of specific code regions of the application (e.g., the initialization, an iteration, or the I/O). However, without appropriate tool support, the insertion of timers into the application is only practical for a limited number of code regions. Usually, they can only convey a coarse-grain view of the application's behavior during runtime. To investigate an application's behavior more closely, a more detailed view is desirable. To obtain such a view, performance analysis distinguishes two fundamental methods: *sampling* and *direct instrumentation*.

Sampling interrupts the running application mostly in regular intervals to inspect the current state of the application, before resuming the execution of the application. This has the advantage that the number of interruptions and accompanying perturbations is easily manageable through the sampling interval. However, this method cannot make any precise statement about the

application for the time between two samples. The data obtained with sampling is usually processed and evaluated statistically. Additionally, sampling does not have explicit access to function parameters, which therefore cannot be analyzed directly.

Direct instrumentation explicitly inserts measurement code into the application at certain points of interest, called *events*, such as function entry and exit, or synchronization and communication points. A special software—the instrumenter—can do so either at the source code [100] or the binary level [16]. Inserting the measurement code directly into those parts of the code that are of particular interest guarantees recording their execution at runtime, generating measurements with exact function-call count as well as matching send and receive events. In the case of pre-instrumented libraries, as provided by MPI implementations [107], direct instrumentation can also give access to function parameters, which can then be measured and analyzed later on. The same holds for callback interfaces such as GASP [92, 152] or MPI Peruse [108]. However, the insertion of this extra code may imply additional overhead. First, the insertion of additional code creates a different executable and may therefore obstruct compiler inlining and also result in a different memory footprint during measurement. Second, depending on the length of the instrumented code sections, perturbation can be especially severe in situations where functions are frequently called, for example getters and setters in object-oriented programming. Performance tools based on direct instrumentation cannot eliminate either disadvantage completely, but usually control them well enough in practice, e.g., by selective or dynamic instrumentation [111, 155, 162].

Data representation

Whichever method of information gathering is used, a measurement system must then decide how to handle the gathered information. It can either summarize the information at runtime, creating a so-called *profile*, or record individual data points in a so-called *event trace*.

A profile is a very compact representation of the applications behavior. As individual data points of performance metrics are aggregated over runtime, the memory requirements are usually independent of the length of measurement. While statistical properties of the set of data points, such as the minimum and maximum duration, can be computed and retained in the profile information, the user does not have any access to individual data points beyond these properties. Especially dynamic behavior in applications, such as specific forms of load imbalances, cannot be inferred from profile data alone. Furthermore, any aggregation at measurement time perturbs the initial measurement. Depending on the complexity of the aggregation method, the level of perturbation can range from low (for simple additions) to high (for complex statistical metrics).

In contrast, an event trace is a detailed account of events in a process or thread during execution. Commonly, event tracing requires more storage space than profiles, and the size of the trace often depends on the length of measurement. However, some trace formats allow for almost constant size trace files for specific applications [109]. Event traces can be processed in different ways. As they still contain all details of the application's runtime behavior, one can use them to create a profile, visualize them, or use the event data for further automatic analysis. Depending on the memory resources available to the application under measurement it may not always be practical

to generate a full event trace for the full application. Even after a successful measurement, further analysis still needs to process potentially large trace files, raising scalability requirements for trace-processing tools.

Analysis and evaluation

Tightly connected with how measurement data is obtained and stored is the analysis and evaluation of the data. The analysis can take place directly at measurement time (online) or after the program finished execution (post-mortem). As with the approaches to measurement and storage of performance data, each method of analysis has its merits and specific field of application. The analysis of the measurement data and the evaluation of its results in a given performance tool often directly depend on each other—i.e., the evaluation method directly relates to the analysis methods applied to the measurement data.

The simplest evaluation method of analysis to implement in a performance tool is the manual inspection of the measurement data without any further processing. Manual inspection of measurement data needs a high degree of expert knowledge on part of the user. Especially novice users may have difficulties in weighing the severity of an observed behavior, for example the time spent in a specific part of the application or the time used for communication. Nevertheless, depending on the presentation of the measurement data manual inspection can still deliver critical insights in the detailed behavior of a parallel application. As the process of understanding may be time consuming, manual analysis of the measurement data is done post-mortem.

Automatic performance analysis seeks to lower the costs in terms of time spent in the manual evaluation of measurement data by identifying performance relevant information in the data and guiding the user directly to those. The analysis works as a filter to the measurement data. Depending on the analysis approach, a non-expert may still need further assistance, if the results are either not filtered enough or filtered too strictly, hiding certain behavioral aspects of the application. For the understanding of wait states in parallel applications, two factors are critical: (1) the identification of a wait state and (2) the identification of its cause. It may depend on the measurement approach whether the analysis step can extract such information automatically.

Online analysis provides direct insights into the application behavior while still keeping the memory requirements low as measurement data is consumed or aggregated right away [169]. However, any additional computation at measurement time may induce further perturbation of the measurement, leading to a distorted model of the application performance. Most online analysis approaches use only process-local data, which sometimes limits capabilities of the analysis approach. To use measurement data available only on a remote process, this data needs to be communicated explicitly. As this communication should be done transparently for the user, some performance tools therefore use so-called *piggybacking* to exchange inter-process information—either by using additional messages on the original application's communication paths or by adding additional data to the original application's data transfers. However, the methods commonly used for piggybacking and may influence the application behavior significantly, as shown by a study of Schulz et al. [142].

The post-mortem analysis avoids the drawbacks of online analysis, but often comes at a price as well. As the analysis does not process the measurement data, but stores it until a later point in

time, memory requirements are usually much higher. The key idea of post-mortem analysis is to avoid perturbation of the original measurement run by avoiding any unnecessary computation at measurement time, thus to be effective, the measurement system has to carefully balance runtime overhead with memory requirements.

If set up properly, post-mortem analysis of event trace data enables deeper analysis strategies than simple time accounting. For example, independent of the origin of the trace data (sampling or direct instrumentation), timeline visualization is a common technique to understand the evolution of the application behavior. It enables a detailed view of process interactions and thus can enable the better understanding of execution patterns, yet, as screen space is limited, developers may need further assistance to isolate points of interest in the timeline. Automatic trace analysis methods, such as those presented in this thesis, seek to provide this assistance, by identifying such points of interest in the application's execution trace and enable effective resolution of any inefficiencies.

1.3.2. Performance tools

Application developers can draw from a variety of performance analysis tools on HPC platforms today, however, most of them either provide only limited support for one-sided communication or focus only on single use-cases, platforms, interfaces, or programming models. This section summarizes the capabilities of performance tools for high-performance computing applications, focusing on their support of one-sided communication. As MPI is the prevalent programming interface on high-performance computing systems, most performance tools unsurprisingly place a particular focus on it, yet, this section also lists tools for other commonly used one-sided communication libraries, such as ARMCI and SHMEM. As individual tools often support several techniques within the taxonomy presented in the previous section, they can often not be classified sharply into a single category. Therefore, the classification of tools is based on their main field of application. Table 1.1 summarizes the discussion of this section; specific limitations marked in the table are discussed in the remainer of this section.

Call-path profiling

Call-path profiling usually encompasses measurement results to be presented as a runtime summary broken down per call-path of the measured application. Such profiles can be obtained from direct measurement, sampling, or a hybrid mix of the two techniques. Furthermore, the profile can be created through runtime summarization or after measurement, based on a detailed event trace. The lightweight MPI profiler mpiP [168][191] developed by the Oak Ridge National Laboratory generates text-based profiles that summarize time spent in MPI. It can be used by simply linking a pre-instrumented measurement library with the MPI application. It breaks down timings by individual call sites, which are detected by call-stack analysis within the measurement library. To enable proper call-site identification, the user needs to enable debugging symbols when compiling the application. PerfSuite [85][196], developed by NCSA, is a similar profiling tool for MPI. Using sampling, it focuses on the creation of call-path profiles in combination with hardware performance counters. It also allows derivation of new metrics using the existing ones. IBM provides the High Performance Computing Toolkit (HPCT) [139][186] for their

Tool name	Sampling	Direct Measurement	Runtime summary	Tracing	Online Analysis	Post-mortem Analysis	Wait-state detection	Contention detection	Root-cause identification	MPI one-sided comm.	ARMCI	SHMEM	Other
Carnival [94]		●		●		●	○		○				
Charm++ Projections [51]		●	●										●
CrayPat, Apprentice^2 [47]	●	●	●	○		●	○			●		●	●
HPCToolkit [158, 159]	●		●	○				○	○	●	●	●	●
IBM HPCST [41, 42, 70]		●	●	●		●	○			●			●
IBM HPCT [139]		●	●	●						●			
Intel MPI Trace Analyzer [72]		●	●	●						●			
Intel VTune Amplifier [73]	●		●				○	○		●	●	●	●
mpiP [168]		●	●							●			
Open|SpeedShop [143]	●	●	●	●	●					●	●	●	●
Paradyn [97]		●	●										
Parallel Perf. Wizard [150, 151]		●	●			●						○	●
Paraver [130]	●	●		●						●			
Perfsuite [85]	●		●							●	●	●	●
Periscope [59]		●	●		●		○			○			
Score-P [81]	○	●	●	●			○			★		☆	
Scalasca [26, 57]	○	●		●		●	★[a]	★	★[a]	★	★		
ShmemTracer [95]		●		●								●	
TAU [145]	●	●	●	●						●	●	●	●
Vampir [110, 112]	○	●		●						●	○	●	○

● Supported ○ With limitation ★ Thesis contribution ☆ Partial thesis contribution

Table 1.1.: List of related performance analysis tools in high-performance computing and the scope of their support in terms of the detection of wait states and their root causes, specifically in one-sided communication.

[a] Extended existing functionality for one-sided communication.

pSeries, eSeries, and Blue Gene solution platforms. Like PerfSuite, it allows the collection of performance-counter data. It uses binary instrumentation to collect performance relevant data at runtime and can create both call-path time profiles and event traces for the instrumented functions. Furthermore, it provides a timeline visualization for its event traces. The SHMEM one-sided communication interface supports similar interposition of a pre-instrumented library to the PMPI interface. Using this interface, ShmemTracer [95] developed by the San Diego Supercomputing Center, provides a lightweight tracing library. It creates binary trace files and allows time profiling based on those traces. The trace files can also be processed by the related

tool ShmemSimulator, based on the PSINS simulation framework [163], to simulate different aspects of SHMEM's network performance on a given platform. The Score-P [81][199] measurement infrastructure supports the generation of Cube [147][182] performance profiles, as well as event traces in the OTF2 trace format. Building on the work of Szebenyi et al. [155], Score-P supports the commbination of sampling for user functions and direct instrumentation communication functions in a single measurement, to reduce measurement overhead. Using interposition of pre-instrumented libraries, it supports the measurement of MPI and OpenSHMEM one-sided communication. Further support for ARMCI, based on the generic OTF2 RMA event model, is planned. For point-to-point and collective communication a research prototype supports the lightweight estimation of wait states [93] during runtime summarization. Each of these five performance tools and measurement infrastructures provides basic time profiles for one-sided communication.

Rice University's HPCToolkit [7][185] focuses on sampling-based call-path profiling to identify scalability bottlenecks even at large scale [157]. As the sampling is also performed for the communication library call, it is agnostic to the communication paradigm, thus support arbitrary one-sided communication interfaces. While it can provide insights also into implementation-specific behavior, it relies heavily on expert knowledge to derive further information as to where wait states originate from or how to resolve them. The TAU [145][200] open-source profiling framework of the University of Oregon supports the measurement of MPI, ARMCI, and OpenSHMEM one-sided communication. Furthermore, it also supports measurements of the one-sided communication runtime of Charm^{++} [18]. It provides extensive profile exploration using the graphical user interfaces ParaProf [15] and PerfExplorer [67, 68]. While it generally supports the generation of trace data using the OTF [80] and OTF2 [52] trace formats, it refers the user to other trace-based performance tools for further analysis.

Paradyn [97][193] of the University of Wisconsin, Madison, is an online performance tool. It uses the Dyninst [16] binary instrumentation infrastructure to dynamically insert measurement probes into the running applications. It creates call-path time profiles based on the instrumented functions. Based on those profiles, time spent in one-sided communication can be evaluated by the user. Open|SpeedShop [143][192] is an open-source profiling tool developed by the Krell Institute and Lawrence Livermore National Laboratory. It supports both static and dynamic instrumentation using Dyninst [183]. Using dynamic instrumentation, it supports the online aggregation of performance data to a graphical user interface using MRNet [135] It also allows performance-counter and call-stack sampling as further measurement techniques. While the focus lies in light-weight runtime summarization, it can also generate OTF [80] trace files. The further analysis of such trace files is, similar to TAU, delegated to other trace-based performance analysis tools.

Online analysis

The Periscope [59][197] performance tool of the Technical University Munich provides online analysis of MPI applications. It uses piggyback messages to exchange inter-process performance information. Using such information, it can identify wait states in the communication and can pinpoint those to the source location where they occur. The wait-state detection, however, is only available for point-to-point and collective communication. For one-sided communication it

supports simple time profiling. The Charm++ framework provides its own performance analyzer Projections [51]. Projections visualizes internal performance data that is used by the runtime system itself to steer internal load distribution algorithms. To investigate and eliminate load imbalances, it focusses on processor utilization, which it can analyze post-mortem based on event traces obtained at runtime. It emphasizes processor utilization to investigate and eliminate load imbalances, but relies on the user to identify problematic behavior.

Trace visualization

Trace visualizations commonly show timeline diagrams, potentially accompanied by other visualizations of corresponding data. Timelines are two-dimensional plots, where one dimension is the (usually) normalized time during application execution and the other dimension is the execution context, such as a process or thread. By plotting the timelines of several processes or threads side by side, differences in behavior, expressed by calling functions at different times or calling different functions altogether, can easily be spotted by the user.

Although the common application of HPCToolkit can be seen in the generation of runtime summaries, it also supports the generation of traces from sampling data. An appropriate timeline visualization helps users to identify load imbalances among processes [156]. Jumpshot [180] is a classical timeline visualization tool developed by the Argonne National Laboratory. It is distributed with the MPI Parallel Environment (MPE) [190]. It supports the visualization of timelines for the full range of MPI functions, including MPI one-sided communication. The Intel MPI Trace Analyzer [72][188] is a commercial tool to measure performance of MPI applications. It supports both runtime summarization and tracing, and can visualize timeline diagrams including one-sided communication. The commercial Vampir [110, 112][201] trace visualizer developed at the Technical University of Dresden allows the investigation of trace measurements stored in the OTF [80], EPILOG [176] and OTF2 [52] formats. Additionally to the trace visualization, it also displays statistical data for the time range currently selected in the timeline view. It supports the display of one-sided communication, however, relies on the expert knowledge of the user to identify inefficiencies and their root causes. As its support for one-sided communication is based on the OTF2 generic RMA event model [6], it will readily support any one-sided interface, such as ARMCI, supported by Score-P in the future. Paraver [130][195], developed by the University of Barcelona, also focuses on the visualization of trace data using timelines. The traces are based on a hybrid approach using sampling for application level functions and direct instrumentation of the MPI library, obtained by its own measurement system Extrae [88]. Paraver's unique feature is the visualization of metric streams—supporting function call information as well as performance counter metrics. It further supports deriving new metric streams from existing ones.

As the primary purpose of timeline visualization is to give a detailed account of the application behavior, all of these visualization tools rely very much on expert knowledge to identify wait states and further performance metrics.

Automatic performance analysis

To reduce the amount of expert knowledge to effectively analyze the performance of parallel applications, some tools employ additional automatic analysis methods to provide performance metrics beyond basic time accounting. Using such tools lowers the entry bar for developers investigating and optimizing the performance of their parallel applications effectively. For their supercomputing platforms, Cray provides the measurement and analysis system CrayPat [44] and the corresponding graphical user-interface Apprentice^2 [47]. They automatically detect performance bottlenecks and static load imbalance among processes, however, they do not detect their root causes. As the SHMEM one-sided communication interface is commonly used on these platforms, the native Cray tools support these naturally. The IBM High Productivity Computing Systems Toolkit (HPCST) [70][187] is another tool that focuses on the productivity aspect of performance analysis. For their platforms, it provides a framework to identify problematic programming patterns and suggests possible solutions for an identified pattern to the user [41, 139]. It uses static code analysis to identify problematic programming patterns and also accepts input from other performance analysis tools.

Performance analysis tools targeted at partitioned global address space languages often support one-sided communication interfaces naturally, as those languages use it extensively. In this context, HPCST also supports tracking remote memory accesses in UPC and correlating them to the corresponding program structures [42]. The Parallel Performance Wizard (PPW) [150, 151][194] of the University of Florida relies on the callback-based instrumentation interface GASP [152] to provide information on the one-sided communication runtime. This may comprise information on the call sites of communication calls to enable relating implicit communication back to its originating source location. However, although GASP is an open interface, the number of runtime systems supporting it is limited. Currently, only the GASNet [27][184] runtime system seems to support GASP, limiting its applicability in a larger context. Still, with GASNet being the major runtime of choice for available open source compilers for Unified Parallel C (UPC) [165], PPW provides a profiling framework for identifying bottlenecks in the original source code context of UPC programs.

As mentioned in the previous sections, one-sided communication enables runtime systems to support global-view programming, similar to multi-threading and tasking, on distributed memory architectures. It is therefore important to also evaluate the state of the art in multi-threading performance tools as a reference point for tool support of such global-view programming models. In particular, locks are available in both thread-based programming and one-sided communication. For multi-threaded applications, HPCToolkit supports the identification of lock contention [158], including blame shifting to identify its root cause. Furthermore, Tallent et al. presented work on identifying bottlenecks in multi-threaded work-stealing approaches [159]. The Intel VTune Amplifier XE [73][189] is a sampling-based performance tool, providing some automatic analysis techniques for thread synchronization, including a *locks-and-wait* analysis, targeted to expose waiting time due to lock contention in thread synchronization. Furthermore, it supports the detection of idle time within the MPI implementation if used with Intel MPI.

The Carnival [94] performance tool, as presented by Meira Jr. and colleagues, directly focussed on the understanding of performance phenomena rather than merely detecting them. It formally introduced *cause-and-effect* analysis as a means to identify and attribute root causes to wait states

and implements it in a pipeline of serial analysis steps. It performs trace-based identification of points of process synchronization, however, its serial implementation limits its applicability to current scales of concurrency. The Scalasca toolset [57], jointly developed by the Forschungszentrum Jülich and the Technical University of Darmstadt, mostly use direct instrumentation to create time profiles and event traces. While versions 1.x provided its own instrumentation and measurement system to generate trace files in EPILOG format, it relies since version 2.0 on Score-P to generate OTF2 event traces. Scalasca's unique feature is the highly parallel search for wait states in communication and synchronization, using a post-mortem communication replay technique. As the contributions of this thesis are based on this approach, it is subject of more detailed presentation in Section 2.

1.4. Contribution of this thesis

Ensuring application performance is a critical aspect of software engineering, particularly in high-performance computing, where underperforming simulation codes may reduce the amount of scientific output achievable in a given time. For developers of parallel simulation codes and communication runtime systems alike to regard one-sided communication as a first-class communication paradigm for their parallel software, proper software tools need to be available to them. Especially on today's large-scale supercomputing systems and even larger scale of future systems, the concurrency presented to a software developer can be challenging to employ efficiently.

Summarizing the requirements on state-of-the-art performance analysis methods for one-sided communication to equal those available for the common paradigms, point-to-point and collective communication, a generic performance-analysis solution should

- detect wait states in one-sided communication and synchronization;
- identify their root causes;
- guide users to appropriate optimization targets in the application; and
- work at large scale.

However, as the overview of performance analysis tools for one-sided communication in Section 1.3.2 has shown, only a few of the tools address multiple of these requirements, and none of them covers all. The performance analysis methods presented in this thesis directly address these requirements. Specifically, it comprises the following contributions.

A generic formal event model for one-sided communication. Event-based performance analysis is based on a formal event model, specifying place and extent of event information recorded during measurement. The work presented in this thesis significantly contributed to the definition of a generic event model for OTF2 [6], used in the Score-P measurement framework, which is the measurement system for multiple performance tools, such as Scalasca [57], Vampir [112], Periscope [59], and TAU [145].

Definition of wait-state patterns for passive target synchronization. In addition to the wait state patterns for active-target synchronization defined in previous work, this thesis presents two previously unstudied types of wait states: *lack of communication progress* and *lock contention*.

Both wait-state types occur in passive-target synchronization, a synchronization scheme commonly used by one-sided communication libraries.

Scalable detection of wait-states in one-sided communication. To identify performance anomalies at scale, the methods detailed in this thesis build upon the parallel, post-mortem traversal of process-local event traces used in Scalasca [56]. This enables process-local performance data to be evaluated in parallel, while performance data relevant for the investigation of process interaction is exchanged ad hoc on the recorded communication paths.

Replay-based communication on implicit communication paths. Scalasca's replay infrastructure relies on the communication information to be available to the process-local replay. For passive-target synchronization, such information cannot directly be inferred on the target process. The work presented in this thesis introduces a high-level messaging framework based on the active-message paradigm enabling communication on implicit communication paths of an application, while retaining the overall parallel ad-hoc communication approach.

Classification of wait states as synchronization-based or contention-based. Synchronization-based wait states, as investigated by Böhme et al. [26], are fundamentally different from contention-based wait states, such as those occurring at lock-based mutual exclusion. While the former is caused by two activities not being concurrent, the latter is caused by two activities being concurrent. This thesis formally introduces the concept of contention points as an additional point of process synchronization in parallel application. Furthermore, it combines it with the original wait-state formation model to form a unified wait-state formation model.

In summary, these contributions directly address the previously stated requirements for state-of-the-art performance analysis methods and brings performance analysis methods for one-sided communication on par with methods for point-to-point and collective communication.

Summary

Since about 2002, improvements of the serial performance of computing elements in processors have slowed down significantly. Concurrency in multi- and many-core processors became the new driver for performance improvements on the processor level. With parallelism on the node and the processor level, the number of processing elements has grown rapidly in the last decade, increasing complexity in understanding application behavior.

Software developers need portable development tools to help reducing this complexity and enable their productivity [167]. This includes performance analysis tools that increase the understanding of inefficient application behavior and its root causes to identify potential optimization targets productively. Message passing is the most prevalent programming paradigm in high-performance computing, providing three distinct communication paradigms: point-to-point, collective, and one-sided communication. While the former two paradigms are well supported by performance tools in HPC, support for the latter is not as extensive. Any lack of support of a specific communication paradigm will directly affect the level of productivity this paradigm can be employed with. Before non-expert developers can regard one-sided communication as a first-class option when choosing the best paradigm for their parallel application, its support by performance analysis methods needs to equal the available support for the other two paradigms. Moreover,

one-sided communication is often used in runtime systems of other parallel programming models. An extensive support of one-sided communication by performance analysis methods will therefore directly affect optimization capabilities for such runtime systems.

This thesis contributes methods for the scalable and portable performance analysis of one-sided communication constructs with respect to the identification and classification of inefficient behavior and the identification of their root causes and corresponding optimization potential. Its automatic detection of wait states and their root causes simplifies the understanding of an application's behavior and exposes those parts of the application responsible for inefficiencies. In combination with the correct assessment of the critical path, it enables developers to analyze their parallel application with any combination of communication paradigms.

Organization of the document

The remainder of this thesis is organized as follows. Chapter 2 discusses Scalasca's parallel communication-replay approach in more detail. Chapter 3 defines the performance metrics for one-sided communication used by the Scalasca toolset and their scalable detection and quantification in a parallel application. Based on this, Chapter 4 will show how these metrics relate to and interact with the performance metrics of other communication paradigms used in multi-paradigm applications. Chapter 5 demonstrates the applicability of the presented contributions with computational kernels, benchmarks, and real world applications. Finally, Chapter 6 summarizes the contributions of this thesis and proposes further work.

2. Event-trace analysis using communication replay

As outlined in the previous chapter, the analysis methods for one-sided communication presented in this theses are based on the Scalasca framework. Therefore, this chapter reviews its workflow and architecture, to enable a deeper understanding of the presented methods and extensions. It places a focus on event tracing in general and Scalasca's implementation of the replay approach in particular.

Scalasca [198] is a trace-based performance analysis toolset predominantly for parallel applications [57]. Its analyzer processes event traces of parallel applications, automatically identifying wait states in inter-process communication and thread synchronization. Beyond this identification of wait states, the analysis provides performance indicators and root-cause information on the identified wait states to further assist the performance optimization of those applications. Figure 2.1 shows how the analysis process is embedded in the overall workflow. Scalasca uses a combination of source-level instrumentation and pre-instrumented libraries to insert the necessary measurement code into the parallel application. Versions 1.x of the Scalasca toolset use its internal measurement system EPIK, whereas versions 2.0 and further use the Score-P measurement system [81], a joint development of several performance tool groups. The prototype implementation of the methods presented in this thesis are based on the EPIK measurement system. The measurement system, as part of the application executable, can either create a runtime summary, a detailed event trace, or both during application execution. As direct instrumentation can lead to significant changes in the application behavior, users should perform one or more measurements creating runtime summaries, and use these to refine and optimize the measurement configuration. Once the measurement configuration is suitably refined, the user can configure the measurement system to create a measurement archive containing detailed event traces. For each process in the application, an individual event trace is created in the measurement archive. Each of these process-local events contains general information, such as the time the event was recorded, and may also contain event-specific information, such as an id of the function that is entered or left, or information on the interaction of processes and threads, such as communication or synchronization partners. The analysis outputs a summarized wait-state report similar to the summary report after runtime summarization. The wait-state report, however, contains additional information on the location of wait states and further performance metrics computed during trace processing.

Scalasca uses a parallel replay to process the event traces in a scalable fashion. A replay, in Scalasca's terminology, is the ordered processing of event-local traces. For each event, the replay infrastructure can trigger callback functions registered by the analysis tool to perform specific tasks. This approach is extremely flexible and enables a multitude of different tasks performed on the trace data.

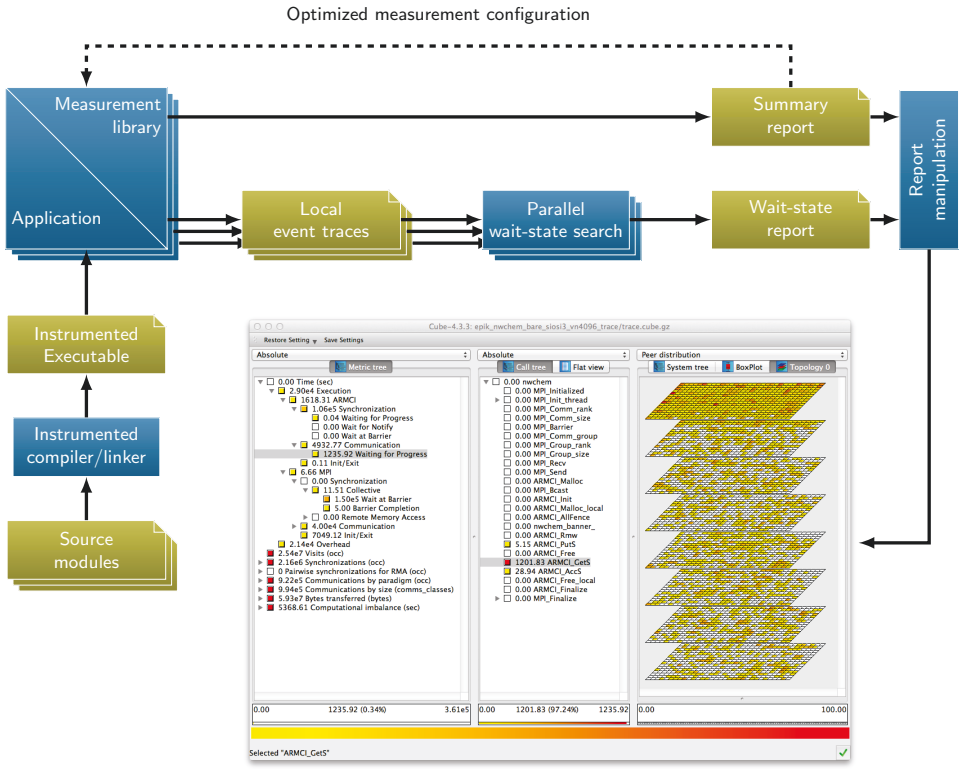


Figure 2.1.: The Scalasca workflow.

Wait-state detection The parallel detection of wait states is the initial use case for the replay approach. The wait-state analyzer [58] registers callback functions for flow events, effectively tracking the source context—the call path—of any event in the local trace. Furthermore, the callback functions registered for communication and synchronization events exchange timestamp information with the respective remote communication and synchronization partner. The analyzer can then use this information to compute potential waiting time, as defined in Chapter 3.

Timestamp synchronization Performance analysis methods comparing timestamp information of different sources heavily rely on their comparability. To ensure the best possible quality of timestamps, Scalasca uses the replay technique to validate and correct timestamps and effectively ensure the semantic validity of the measurement using the controlled logical clock [13]. The algorithm uses the recorded communication events to trigger the exchange of timestamps, just like the wait-state analysis does. When a violation of communication semantics is encountered, such as a *receive* ending before its corresponding

send started, the timestamps of the corresponding events are modified to reflect the correct semantics again. In principle, the replay technique also supports further post-mortem timestamp synchronization schemes, such as the one presented by Doleschal et al., which also corrects clock drifts beyond clock condition violations.

Performance simulation To help developers estimate the gain or costs of potential optimizations to their applications, Scalasca uses the replay approach also to simulate application behavior [64]. In multiple replays, the callback functions first modify an existing event trace according to a user-provided simulation configuration and then simulate the application behavior through re-enacting communication and computation in real time. Modifications can be the balancing of time spent in a certain *call path* across processes or the reduction of time spent in a specific region, which simulates a performance optimization. The simulator writes a modified measurement archive that can then be processed using the wait-state analyzer.

Root-cause analysis Beyond the detection and quantification of waiting time in parallel applications, Scalasca uses the replay approach also to detect imbalanced code regions as the source of these wait states [26]. Similar to the wait-state analysis, information about root causes are exchanged at communication and synchronization points.

Critical-path analysis Additionally to the wait-state and root-cause analysis, Scalasca follows dependencies between processes from the end to the beginning of the trace, to identify activities of different processes and threads on the critical path [24]. It further derives so-called *performance indicators* to reveal overall time lost due to load imbalances and help developers identify optimization targets. The relevant information is exchanged by callbacks triggered at communication and synchronization events.

This broad variety of applications demonstrates the flexibility of the overall approach. To enable the replay, the approach relies on specific information being available in the trace. Specifically, it requires a lossless tracking of communication and synchronization functions of a program. This is, as is explained later in this chapter, due to its reuse of the applications' measured communication paths during the analysis, based on purely local information. If one process does not record its part of a communication during measurement, it will not participate in analyzing the same communication in the replay. In most cases this will lead to a deadlock during the analysis. To ensure the consistency of the communication information distributed across the participating processes, the measurement therefore uses direct instrumentation. This enables the measurement system to track every communication in the application, ensuring that the replay based on the measurement of any correct application will not deadlock.

Wait states, as introduced in Chapter 1, describe states where processes sit idle, waiting for a remote process to perform a certain action, such as starting the communication. For the identification and quantification of wait states, Scalasca compares the information on *enter* and *exit* of corresponding communication functions. Such information is generally not available to the local process without further communication of information on these dependencies to and from other processes. While the analysis could estimate wait states by comparing the time actually spent in communication to an ideal communication time [93], one cannot identify its cause. One method to transfer the necessary information during measurement would be the use of so-called piggyback messages. Such messages describe the additional payload being sent alongside of an

application-level message. Piggyback messages enable the identification of wait states during measurement as well as the aggregation of such information in a runtime summary. However, they are limited to wait states that occur on the receiver side, as information only flows with the original direction of the message, and not in opposite direction. By using event traces and postponing a more detailed analysis until the end of the measurement, Scalasca can overcome both shortcomings and accurately identify wait states on both the sending and the receiving side of a communication. While this also increases memory requirements during measurement, it allows to shift some of the computation done for a runtime summary to a time when it does not influence the measurement anymore.

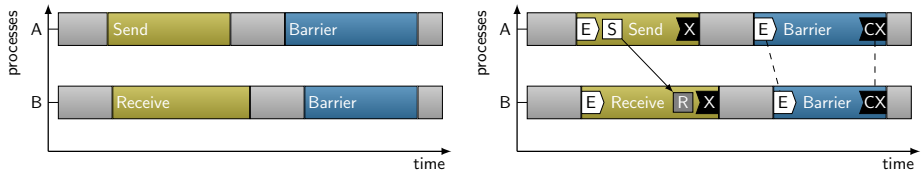
While the mere identification of wait states is more forgiving to individual wait states being missed by any analysis process in general, the identification of root causes and the assessment of the critical path of a parallel application need a rigorous identification of all synchronization points to remain valuable, as will be explained in Chapter 4. As sampling cannot guarantee to record all points of synchronization, direct instrumentation is required at least for such points. Control flow information beyond the synchronizing calls may be collected either through direct instrumentation or sampling methods, in a balance between accuracy and overhead [155].

In summary, the replay method employed by the Scalasca toolset commands the use of direct instrumentation for all event sources that may cause the wait states. The flexibility of the method then allows for an accurate identification of wait states and their root causes, as well as for performance indicators helping in assessing optimization potential.

2.1. Event tracing

Event-based performance analysis models application behavior by recording state changes of the application measured. Such execution state changes are called *events*. The information needed to analyze the performance are encoded in *event records*. Because of the tight relation of event records and events, the term *event* is also often used to refer to specific event records. As some of this information is specific to single records, a set of multiple event records—the *event set*—is used during measurement. The semantics of individual records in the event set and their relation to each other is described by the *event model*, which forms the basis for modeling application behavior and performance.

Examples for different events available to model the application behavior are those referring to the flow of control during execution, such as entering and leaving a code region, and those referring to points of interaction and synchronization of concurrent processes or threads in a parallel application. Figure 2.2c lists the small subset of the EPILOG event set [176] used by the EPIK measurement system of Scalasca 1.x relevant for the understanding of this chapter. The events shown can roughly be classified in those describing the control flow (left) and those describing communication (right). Events such as the collective exit CX (middle) belong to both. It describes both the end of the respective function call and the communication involved in the collective operation. For the events listed, the semantic description of EPILOG's event model includes that a send event S and a receive event R form a pair, i.e., one send event on the sending process matches exactly one receive event on the receiving process. The send event S is recorded before the actual communication is started, marking the earliest time the communication may

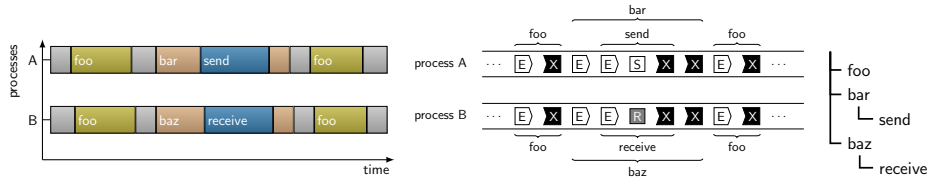


(a) Plain timeline view without events.

(b) Timeline view annotated with events.

(c) Subset of the EPILOG event set.

Figure 2.2.: Timeline view of point-to-point and collective communication calls. Classic timeline visualizations (a) display each process/thread timeline as a segmented rectangle where the segment color corresponds to the function or code region that process was executing at the time. Such displays use the underlying event information in the event stream (b) to determine the extent of regions and the transfer of message. Different events during application execution are represented by specific event records (c).



(a) Timeline view of different functions being called.

(b) Representation as a stream of events.

(c) Joint call tree.

Figure 2.3.: Timeline and event-stream view of control flow and communication events, including a joint call tree.

have started. The receive event R is recorded after the actual communication is completed, marking the latest time the communication may have completed. To compute the time spent within a point-to-point communication, the send and receive events must be enclosed by enter E and exit X events, describing the full duration of the function call used to perform the communication, such as `Send` or `Receive` in Figure 2.2. Figure 2.2a shows a constructed example of a typical timeline view as it is presented to the user by trace-visualization tools. Figure 2.2b shows which events are part in constructing such a view. For each enter event, a corresponding exit event must be present in the trace. The regions modeled by those events must not partially overlap, i.e., they are either non-overlapping or one fully encloses the other. The collective exit CX event matches with its preceding enter event (see Figure 2.2b), as it describes both the collective communication and the end of the function call. Both of these events must be present for all processes participating in the collective call.

A chronologically ordered sequence of events is called an *event trace*. Figure 2.3 shows how

application behavior is represented by such an event trace. Figure 2.3a details a classic timeline view of two processes A and B calling different functions over time. The duration of a call is expressed by the width of the box in the timeline. Calls to functions within functions are expressed by splitting the parent function's box in two with the child function's box in between. In this example, function `bar` calls a communication function `send` on process A, while the function `baz` on process B calls a different communication function, `receive`. Figure 2.3b shows how this scenario would be represented by two streams of events. The measurement system encodes state changes of the application using events of specific types. In this example, these are event types for entering E⟩ and exiting X a function as well as sending S and receiving R a message. The braces show how these events correlate to the individual functions.

The measurement system provides a calling context to individual events by recording changes to the control flow using enter and exit events. By keeping track of these enter and exit events while processing a trace, a performance tool can recover the information about the calling context for the events being processed, without the need to explicitly store it as part of every event record. For each enter, the tool stores an identifier for the current function call onto a stack—the call stack. For each exit, the tool removes the top element of this stack. At any given time, the contents of this stack from the bottom to the top yields the *call path* of an event. By combining all recorded call paths, a performance tool can construct the *call tree* of an application, as shown in Figure 2.3c. Using such call trees, performance tools can present analysis data in a compact and intuitive way. The more detailed the call-path information is, the better it helps distinguish communication in different parts of the application code and identifying the location of execution hotspots and wait states.

Event traces provide a fine grained view of an application's behavior over time. The applications for such a view are manifold and not only restricted to performance analysis. For example, Kranzlmüller used event tracing to build graph structures for debugging message passing communication [83]. Calotoiu et al. [30] use event traces to detect manual implementations of collective functions independent of the communication pattern used. The PSINS [163], Dimemas [88], and Silas [64] simulation frameworks also use event traces for different aspects of performance simulation.

Some performance tools, such as Vampir [112] and Jumpshot [177], display event traces in timeline displays, and allow the detailed visual introspection of application behavior. As no official standardization body for event trace formats exists today, different tools often use their own proprietary trace format tailored to the specific needs of the tool in question. The range of available trace formats includes SLOG2 [35], EPILOG [176], OTF [80], OTF2 [52], STF [75], and ScalaTrace [120]. In many cases, however, the information encoded in such event traces is similar enough that some tool providers offer either the conversion from one format to another (e.g., Scalasca 1.x offered conversion scripts from its then native EPILOG format to SLOG2, OTF, and Paraver's native trace format). However, with growing trace sizes, it becomes more impractical to store multiple copies of the same data in different formats. Therefore, programs like Scalasca and Vampir support multiple trace formats natively through internal abstraction layers, such as PEARL [56] in case of Scalasca. Furthermore, the trace-based performance tools Scalasca, Vampir, and Periscope selected the OTF2 trace format as their main trace format, i.e., they all natively support the traces created by the Score-P measurement system.

2.2. Definition of wait states based on event models

Based on an event model, wait states can be defined using the dependencies between events. For example, in the *Late Sender* wait state, as shown in Figure 2.4a, the receiver is waiting for the sender to send its message. It will not be able to receive the message until the sender has started the communication. Thus, the time spent in the receive call before the sender entered the communication call can be considered as waiting time. The amount of waiting time associated with a wait state is expressed as ω, which is expressed formally through the following equation:

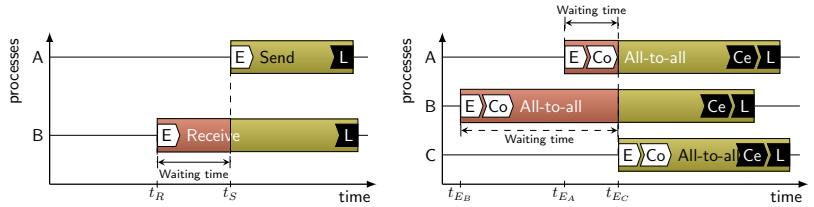
$$\text{Late Sender:} \quad \omega = \begin{cases} t_S - t_R & \text{, if } t_S > t_R \\ 0 & \text{, otherwise} \end{cases} \tag{2.1}$$

Another example for wait states in communication are collective operations with an all-to-all communication pattern: no process can complete the call before the last process started the call. As a result, all processes wait for the last process to join before they return to the user application. The time between the local enter event and the enter event of the last process to join the call is identified as a *Wait at N×N* wait state (shown in Figure 2.4b). Assuming that t_E^i represents the time of the enter event on process i and t_E^l represents the local enter time, then the waiting time can be expressed formally as:

$$\text{Wait at N} \times \text{N:} \quad \omega = max(t_{E_0}, t_{E_1}, \ldots, t_{E_{n-1}}) - t_{E_l} \tag{2.2}$$

The definitions of these wait states only consider the relationship among event information, not where (on which location) this information is locally available. It is the task of the analysis process to communicate this information to the process performing the computation of the waiting time accordingly. For example, Equations 2.1 and 2.2 compute the local wait state using timestamps from multiple locations. For the *Late Sender* wait state, these are the timestamps of the send event t_S, which is recorded on the sending process, and the timestamp of the receive event t_R, which is recorded on the receiving process. For the *Wait at N×N* wait state, the enter timestamps of all processes in the communicator need to be evaluated.

The point in time when such information is communicated may differ among tools. The KOJAK performance analysis tools set—Scalasca's predecessor—also performed an automatic search for wait states [174]. However, it collated all trace-local information at the end of the measurement and combined it to a single event trace holding all global information. This approach bore two disadvantages: (1) the additional I/O to persistent storage to merge the local event traces to a single global event trace and (2) the multiplexing of events belonging to multiple processes is complicating parallel processing. Especially the second disadvantage is due to the fact that this initial approach was never designed for parallel processing. Scalasca therefore follows a completely new approach that requires neither rewriting nor copying event traces after measurement, hence, enabling efficient parallel processing at large scales.



(a) The *Late Sender* wait state. **(b)** The *Wait at N×N* wait state.

Figure 2.4.: Definition of typical point-to-point and collective communication wait states.

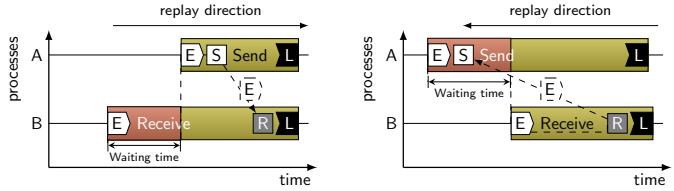
2.3. The communication-replay analysis method

Scalasca communicates the inter-process information just-in-time by replaying the communication pattern recorded in the trace. The general idea of this communication replay approach is very similar to online analysis using piggyback messaging. There, performance-relevant information is sent either included in the application's original message or with an additional message on the same communication path. The post-mortem communication replay technique mimics the latter, sending performance-relevant information on the recorded communication path. Compared to the online analysis approach using piggybacking, the post-mortem communication and computation does not perturb the initial measurement. This enables more complex computations during analysis and even multiple passes over the recorded data. Moreover, as done by the methods presented in this thesis, communication paths other than those explicitly recorded can be used as well.

The heart of Scalasca's parallel event-trace processing is the C++ event-trace interface library PEARL [56]. It provides a high-level interface to event traces comprising three main parts that enable easy parallel processing: (1) An event-class hierarchy that enables an abstract interface to event information, independent of the trace format, (2) a callback infrastructure that allows the execution of specific function calls, and (3) a replay mechanism that enables the parallel processing of event traces.

The class hierarchy enables users to access event information independent of the binary trace format. Basic event information, such as the timestamp of an event, is available for all event types. Specific event information, such as communication information, is only available at the corresponding event objects. Furthermore, it provides access to information not explicitly saved with each individual event, such as the call path a particular event is associated with. The call-path information, for example, is computed ad hoc by tracking enter and exit events during the replay.

The callback infrastructure uses a publish-subscribe mechanism [19] to enable an arbitrary number of callback functions to be called when processing an event. It decouples the PEARL library from any specific functionality of applications using it. The library allows the registration of so-called *callback functions* to be executed on the processing of a certain event. PEARL allows two types of events: *native* and *user events*. Native events correspond to the event types available in the event model used to describe the application behavior, e.g., events for entering and leaving



(a) Late Sender **(b)** Late Receiver

Figure 2.5.: Communication replay for the identification of the *Late Sender* and *Late Receiver* wait state.Triggered by the communication events S or R, respectively, the potentially causing process sends the time of its enter event E to the communication partner, where it is received by a handler of the corresponding communication event and used to compute the waiting time. Note that the two patterns are detected in subsequent replays of opposite direction.

a code region, sending and receiving a message, or entering and leaving a collective operation. These event types are defined within the PEARL library. User events can be defined by the user of a library and are represented by unique, user-defined integer values.

The replay mechanism allows the parallel processing of the individual process-local traces. It starts at one end of the local event trace and advances to the next event until it reaches the other end. Each replay starts and ends collectively for all processes. Further synchronization and communication depends on the callback functions registered for a specific replay. For each event in the event trace, the replay engine triggers the native event corresponding to the type of the event processed. Then, the callback functions are executed in the order they were registered. Each callback function can call other functions or signal further (user) events. Once the execution of a callback—including all of its directly called functions and further triggered callback functions—has finished, the next registered callback is executed. This process is repeated for all callbacks registered for the native event type, before the replay advances to the next event. Scalasca uses replays in both directions of the event trace: a *forward replay* in chronological order of the events in the trace and a *backward replay* in reverse-chronological order. While the forward replay enables efficient flow of information between events in chronological order, using the recorded communication graph, the backward replay enables data to flow in reverse chronological order, using the inverted communication graph of the application. The ability to invert the communication pattern is a key requirement for the detection of root causes and the quantification of their costs as described in Chapter 4, but it also enables the detection and quantification of wait states on the sending process, such as the *Late Receiver*. It is similar to the *Late Sender* wait state, but here the sender waits for the receiver. To detect it, information needs to be passed from the receiver to the sender.

Figure 2.5 shows the communication performed for the *Late Sender* and *Late Receiver* wait state during wait-state detection. The callback function registered for the respective communication events S and R, respectively, in the corresponding replay collects the event information of its corresponding enter event E and sends it to the recorded communication partner of the process. The callback function registered for the corresponding communication events R and

S, respectively, on the communication partner receives the message and comput es the waiting time using the local and the received enter timestamps.

Summary

Scalasca is a performance analysis toolset focussed on the identification of wait states in inter-process communication and synchronization. It uses a trace-based parallel analysis method to identify these wait states. The wait states are defined based on an event model used to describe an application's behavior. The communication and synchronization events in the trace must be obtained by direct instrumentation to ensure all such events are recorded by the measurement system. Using a parallel post-mortem traversal of process-local event traces in either chronological or reverse-chronological event order to communicate inter-process information enables the detection of wait states and their root causes described by such definitions. The high degree of parallelism and the just-in-time communication of inter-process information on the recorded communication paths enables an efficient automatic detection of wait states and their associated waiting time.

3. Wait states in one-sided communication

The knowledge-based identification of wait states in parallel programs is one of the key components of the Scalasca performance analysis toolset. At its original release in 2006, the Scalasca performance analysis toolset supported the detection of wait states in point-to-point and collective communication. While Scalasca's predecessor KOJAK [175] already supported the detection of wait states in active-target synchronization of MPI's one-sided communication [86] and SHMEM [99], Scalasca itself initially did not support these with its replay-based analysis.

This chapter describes the contributions of this thesis to the overall trace-analysis framework. Section 3.1 introduces the OTF2 generic event model for one-sided communication, including examples of its use to model MPI, ARMCI, and SHMEM communication. Using this event model to describe wait states in one-sided communication, Section 3.2 reviews previously published types of wait states for active-target synchronization and introduces previously unstudied wait states of passive-target synchronization. Section 3.3 then describes a novel analysis scheme to discover and quantify these wait states.

3.1. A generic event model for one-sided communication

As explained in Section 2.2, Scalasca defines its wait-state patterns based on an event model. Scalasca's original measurement system uses the EPILOG [176] event model, including a events for one-sided communication [65]. Since version 2.0, Scalasca supports the Score-P measurement system [81] as its primary source of event traces, which uses the OTF2 [52] event trace format. This thesis contributed to the definition of the generic OTF2 event model for one-sided communication [6] and uses it to describe the wait-state patterns in one-sided communication interfaces. As the successor of EPILOG and OTF, it surpasses both in its modeling capabilities using a smaller number of generic event types. The definition of these patterns using a generic event model also emphasizes that they may occur in more than one specific interface.

One-sided interfaces separate communication and synchronization in data transfers. To identify wait states, an analysis therefore has to consider both of them. Individual one-sided communication interfaces, such as the ones of MPI, ARMCI and SHMEM, often define an individual set of RMA operations and synchronization functions. The semantics of individual operations and synchronization functions, although named similarly, may differ across different interfaces. For example, regarding local completion of an operation, ARMCI explicitly declares blocking and non-blocking variants of its RMA operations, whereas OpenSHMEM, as the representative for the SHMEM family of interfaces, only provides blocking operations and MPI defines only operations that are implicitly non-blocking. Regarding remote completion, all three of these interfaces provide functionality to ensure it explicitly. Any function ensuring either local or remote completion is a candidate for containing a wait state.

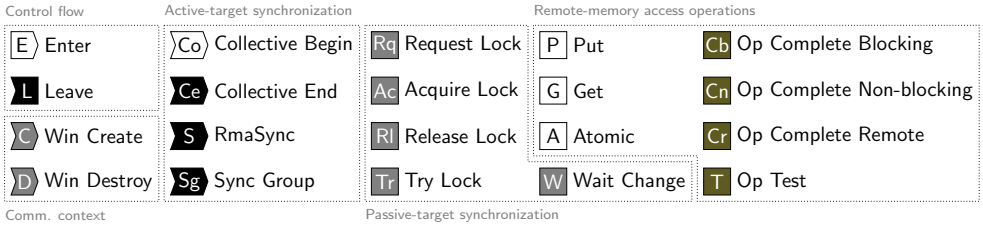


Figure 3.1.: The OTF2 event set for one-sided communication [6]. Different semantic groups are indicated by dotted lines. Control flow events are part of the general OTF2 event model. Communication context, active-target synchronization, passive-target synchronization, and remote-memory access operations are part of the generic one-sided communication event model.

MPI one-sided communication has a very generic one-sided communication interface. The standardization body took much care in the design of the interface not to require a certain memory model for a specific implementation. It uses abstractions to hide specific details of the remote memory access. Although other one-sided communication interfaces do not use these abstractions, but rather expose the specific implementation detail, this thesis will use the name of the corresponding MPI abstraction to ease the understanding of common behavior among one-sided communication interfaces.

Figure 3.1 lists the generic one-sided communication event types of OTF2. A total of five semantic groupings are indicated by the dotted frames: control flow, communication context, active-target synchronization, passive-target synchronization, and remote-memory access operations.

The control flow events are not strictly part of the event model for one-sided communication, but listed here for completeness. The enter E and leave events L model the entering and leaving of a specific code region, such as function calls, and are part of the initial OTF2 specification [52]. As the beginning and end of the one-sided communication and synchronization functions are also modeled by these events, they are included here for completeness. All remaining events in the figure are part of OTF2's generic event model for one-sided communication.

Some communication interfaces, such as MPI, provide multiple independent communication contexts to the developer. As communications, such as collectives, are sometimes tightly coupled to the communication context, it needs to be tracked adequately. As a general abstraction of the different communication contexts provided by different one-sided communication interfaces, the OTF2 event model uses the name *window* for accessible remote memory. OTF2 models the creation and destruction of a common communication context using window-create C and window-destroy events D, respectively. References to such a communication context by remote-memory access operations and synchronizations can only occur after the window-create and before the window-destroy record. For one-sided communication interfaces that provide implicit global communication contexts, OTF2 assumes an implicit window creation during initialization and a corresponding implicit destruction during finalization of the communication.

Creating such communication contexts is usually a collective operation over all processes of the context, often involving process synchronization. As all processes in the communication

context are involved in such a collective operation, it counts to the active-target synchronization schemes. This type of collective communication and synchronization is modeled by explicit event records. The collective-begin event Co is placed directly after the enter event and the collective-end event Ce is placed directly before the leave event of the collective call. Next to collective synchronization on the full window, one-sided communication interfaces often support more lightweight synchronization schemes. The sync event S models pairwise synchronization, while the sync-group event Sg models group-wise synchronization.

In passive-target synchronization, process and memory synchronization is done via shared resources called *locks*. A lock is a token that a process has to acquire before the execution of specific operations can continue. Depending on the one-sided interface, the actual acquisition of a lock by the local process can occur at different times in the execution. The most common and most intuitive behavior in lock acquisition is that the function to acquire the lock returns only after the lock's successful acquisition. The one-sided communication interfaces ARMCI and SHMEM use this behavior for their interface semantics. For these kind of interfaces, the measurement system places the lock-acquisition event Ac at the end of the call to mark the time the acquisition was completed successfully. MPI one-sided communication follows a different approach. It allows its lock function to return before the lock was acquired successfully. Modelling the lock acquisition at the end of such the lock function would therefore lead to inconsistent lock semantics, as multiple processes appear to own the lock. An MPI implementation is allowed to delay the lock acquisition to a later point in the execution, as long as it guarantees that any local remote-memory access operation to the corresponding window are scheduled after the lock was really acquired. This enables MPI implementations to schedule RMA operations in passive-target synchronization internally, at the expense of hiding the acquisition completely from the user view. To mark the earliest time the lock could have been acquired by the local process, the lock-request event Rq is written at the beginning of the lock function. Some one-sided interfaces, such as OpenSHMEM, provide the middle ground between a completely non-blocking lock acquisition and a fully blocking one: a non-blocking call that indicates on return whether the lock was acquired or not. An unsuccessful locking is recorded using the try-lock Tr event, whereas a successful locking results in a lock-acquisition event Ac record. Some communication interfaces allow users to wait explicitly for values in the target window to change, which can be modeled with the wait-change W event.

For remote-memory access operations, OTF2 can model both blocking and non-blocking local completion semantics, as well as remote completion. The start of different types of RMA operations are modeled using the put P, get G, and atomic A events. The corresponding completion events depend on the semantics of the corresponding call. The op-complete-blocking Cb event records the local completion of a blocking operation. The op-complete-non-blocking Cn event records the local completion of a non-blocking operation. Some interfaces allow the applications to query remote completion on an operation, which is recorded by an op-complete-remote Cr event. Similar to locks, interfaces may allow to test for local completion of a non-blocking operation, as known from MPI point-to-point communication. This is recorded using the op-test event T.

To demonstrate the expressiveness of these event types for use with different one-sided communication interfaces, the following sections will discuss how they are used to model MPI, ARMCI and SHMEM communication. The event sub-model for one of these interfaces may not use the

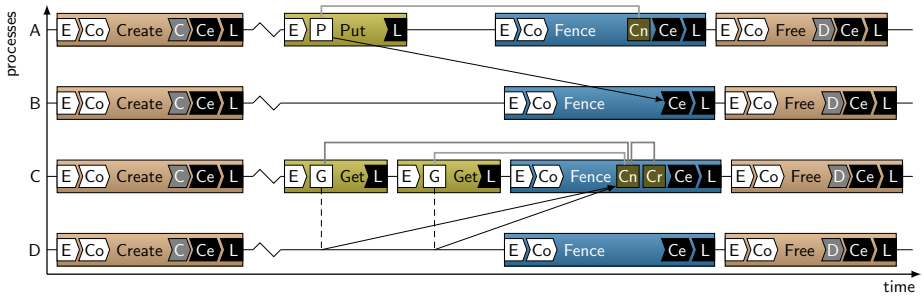


Figure 3.2.: OTF2 event model for MPI collective synchronization. In addition to standard enter and leave events, collective events comprise an additional set of begin and end markers. Also creation and destruction of windows are marked in the respective calls. All RMA operations are implicitly non-blocking, thus they are modeled by the non-blocking event set. The fence call in MPI only guarantees local completion; remote completion is guaranteed by the remote fence call. Therefore, fence only store remote completion events for get operations.

full range of available event types, but rather only uses those applicable to the specific interface. The individual event types keep their semantics across different models, enabling the definition of generic wait-state patterns applicable to multiple one-sided communication interfaces.

3.1.1. MPI

MPI one-sided communication is based on the memory abstraction of a *window*. It emphasizes the fact that often times not the complete memory can be accessed by remote processes, but only a fraction. A reference to such a memory region is a *window handle*. The time between start and completion of remote memory access operations and their synchronization is called an *epoch*. MPI differentiates this further into *access epochs* on the origin, and *exposure epochs* on the target. The latter define the time a window is accessible for remote memory access.

Window handles are always associated with a specific communication context, the so-called communicator. Collective operations on a window are collective in context of the window's communicator. Naturally, users of MPI one-sided communication can define window handles using any valid communicator. Furthermore, users can also create multiple distinct windows using the same communicator. Operations on these individual windows are independent of each other, no matter whether the same communicator was used during creation or not. To enable the correct attribution of RMA operations to individual windows, the measurement system can use specific window-definition events to hold all relevant information about the individual windows defined by the application. Figures 3.2 to 3.4 showcase the relationship of the different events used in the event model. The common prefix `MPI_` for RMA operations and `MPI_Win_` for synchronization calls is omitted for clarity.

Completion records for remote-memory access operations

The standard RMA operations defined by MPI are neither explicitly blocking nor non-blocking. It is dependent on the implementation whether an individual RMA operation blocks during application execution or not. A user of MPI one-sided communication should therefore not assume either case. As with explicitly non-blocking operations, all RMA operations must be enclosed by corresponding synchronization calls. In the non-blocking case, these calls perform the necessary synchronization; in the blocking case, they behave like no-ops regarding operation completion. Semantically, the data transfer is only guaranteed to be completed after the corresponding synchronization function. The OTF2 event sub-model for MPI therefore uses non-blocking record types to model calls as non-blocking. It places the start of the operation (i.e., P, G, or A) after the enter E⟩ event of the respective region instance and the end of the operation right before the leave ⟩L event of the corresponding instance of the synchronization-call region. Which type of completion (local or remote) is recorded at a specific synchronization call depends on the RMA operation and the synchronization call and is discussed in the following sections on the different synchronization types in MPI.

Collective operations and synchronization

Processes create and destroy window handles collectively. As shown in Figure 3.2, all processes record those operations in their respective calls. Both operations—creation and destruction of a window—potentially synchronize the processes, but they do not complete pending RMA operations. Note that MPI uses the term *free* for the destruction of a handle. Any RMA operations issued by the application must be completed by a synchronization function prior to the window destruction. The fence synchronization call is collective on the full window. As the synchronization call completes all pending operations, a single completion record can be used to mark the completion record of multiple operations, as shown on process C, where two distinct get operations are completed by a single fence. Note that a fence in MPI only ensures local completion Cn of pending operations. Remote completion Cr is ensured by the corresponding synchronization call on the target. For put-like data transfers, neither process has the full information to mark remote completion. Therefore, remote completion events are not recorded for those transfers. For get operations, however, remote completion can be inferred from the local completion, as the transfer should complete remotely when all data has been fully transferred at the latest. Therefore the measurement system can record remote completion of the pending get operations on process C at the end of its local fence operation.

Group-based synchronization

General active-target synchronization (GATS) also involves multiple remote processes, but it is not collective. It separates the origin and target processes into separate groups, each with distinct synchronization calls to open and close access epochs on the origin processes and exposure epochs on the targets, respectively. Before RMA operations can be issued in general active-target synchronization, an origin process needs to call `MPI_Win_start`. A single call to

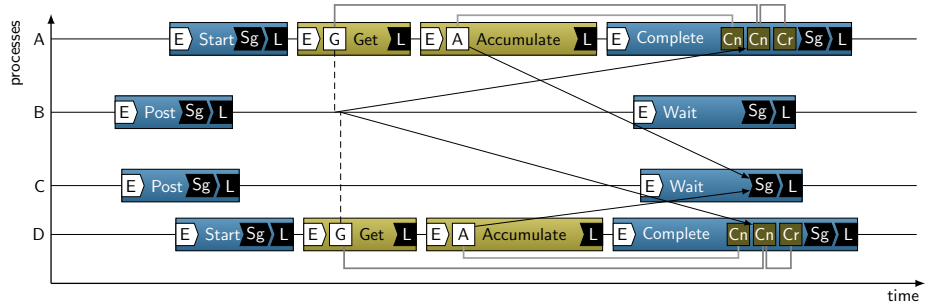


Figure 3.3.: OTF2 event model for MPI general active-target synchronization. Synchronization is modeled via the sync-group events in the corresponding synchronization calls. The groups stored with the event comprise the remote origin and target processes, respectively. The complete call only guarantees local completion of the operation; remote completion is guaranteed by the corresponding wait call on the target. Therefore, complete calls only store remote completion events for get operations.

`MPI_Win_start` may start access epochs for multiple targets for a given window, which are specified as a parameter to the call. The access epochs to all targets of a given window are closed by a call to `MPI_Win_complete`. It is not possible to close only a part of the open epochs on a given window. Therefore, there is a single pair of start-complete calls for multiple targets for a given window. This single pair of start-complete calls is matched by a single pair of calls to `MPI_Win_post` and `MPI_Win_wait`, respectively. All target processes have to provide the ranks of all origin process that specify them as targets when opening their access epoch. Figure 3.3 shows a standard scenario where two targets (processes B and C) open an exposure epoch for the origins (processes A and D). The group relationships for synchronization are recorded using the group-synchronization record Sg. Remote completion is reached at the end of the targets' respective wait calls, however, for other than get operations the target again does not have knowledge about any specific RMA operations of the exposure epoch at hand. Therefore analogous to the collective synchronization, the completion of the accumulate operations of processes A and D are modeled by non-blocking completion records only and the completion of the get operations, as in the previous case in Figure 3.2, are modeled with both non-blocking completion and remote completion records.

Lock-based synchronization

Passive-target synchronization does not have an explicit notion of access and exposure epochs. As the target is passive—it does not actively open the window for access—it is the origin that logically opens both its access epoch and the target's corresponding exposure epoch at the same time. As MPI ensures mutual exclusion in passive-target synchronization through locks, these combined epochs are named *lock epochs* for the remainder of this thesis. To allow MPI runtime systems to schedule concurrent accesses to a window flexibly, the call to acquire a lock may be non-blocking, i.e., the call may return to the application before the lock is actually acquired. All subsequent operations must either also be non-blocking and queued, or block until after the successful acquisition of the lock. The measurement system does not have any

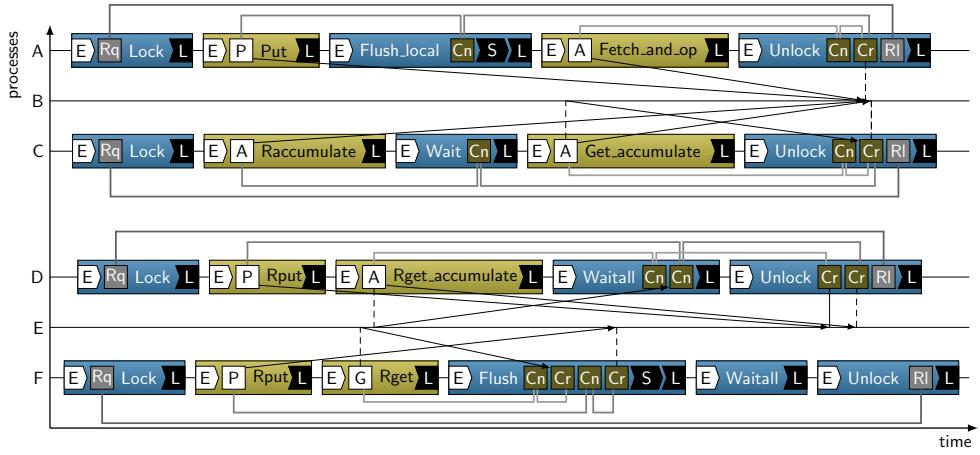
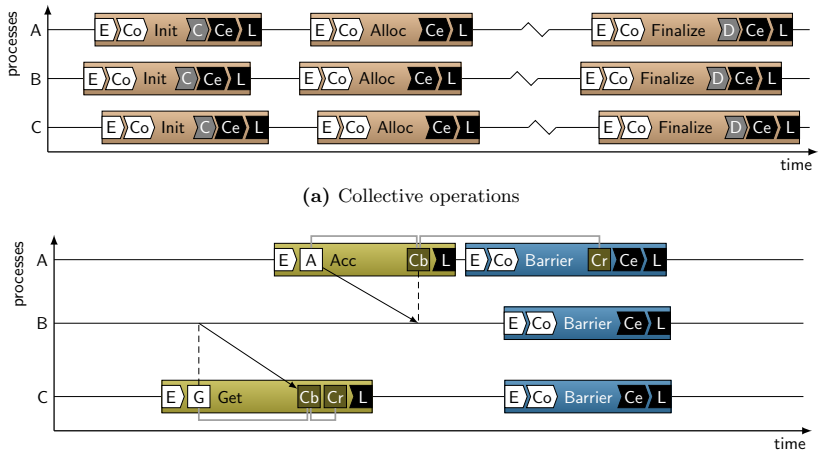


Figure 3.4.: OTF2 event model for MPI passive-target synchronization. Locks model simultaneous access and exposure epochs. The release-lock event also marks remote completion of any operation. Flush calls, as introduced in MPI 3.0, enforce local completion of any operation. As always, get operations complete remotely when they complete locally and all other operations complete remotely at the release of the lock.

data at measurement time to infer whether a lock was acquired. Thus, for MPI, it can only record the request for a lock Rq as shown in Figure 3.4. In this figure, process B and E are the passive targets of processes A, C, D, and F's operations. The lock epoch is ended by an unlock call. This call is guaranteed to block until both local and remote completion of all pending operations. Therefore, the completion of put and accumulate operations can be modeled by both non-blocking and remote completion records.

MPI 3.0 [104] introduced additional synchronization calls for passive-target synchronization. Now, developers can use various flavors of flush to ensure local and remote completion of pending calls during a lock epoch. A flush (process F) ensures local and remote completion of any pending RMA operations. A local variant of flush (process A) ensures only local completion of the operation and remote completion is then ensured at the end of the lock epoch. MPI 3.0 and later also supports explicitly non-blocking one-sided operations, namely the calls `MPI_Rput`, `MPI_Rget`, `MPI_Raccumulate`, and `MPI_Rget_accumulate`. These calls return a request handle as known from non-blocking point-to-point communication to the application that can be used to test or wait for local completion. A corresponding wait call (processes C, D, and F) therefore marks the non-blocking completion Cn for every request separately. Note that these corresponding wait calls are not special one-sided communication synchronization functions, but the same request completion calls as for point-to-point communication. These calls only provide local completion semantics, thus remote completion is marked at the end of the lock epoch.



(a) Collective operations

(b) Active-target synchronization

Figure 3.5.: OTF2 event model for ARMCI collective and active-target synchronization. Initialization, finalization, and memory allocation are collective operations and modeled as such. Neither of these has an effect on ongoing operations. A barrier synchronization invokes an all-fence operation on every process, ensuring remote completion for any pending operation.

3.1.2. ARMCI

The Aggregate Remote Memory Copy Interface (ARMCI) [117] is the underlying one-sided interface of the Global Arrays library. It has been designed by the Pacific Northwest National Laboratory (PNL) to support its PGAS-style distributed data structures and computation calls on them. PNL also provides an implementation of the interface with a library of the same name. While this is the default link target for the Global Arrays library, other implementations of ARMCI using MPI [49] or OSPRI [61] exist. Figures 3.5 and 3.6 show the OTF2 event model for ARMCI using the generic one-sided communication events. A common `ARMCI_` prefix to the calls is omitted for clarity. As the communication context in ARMCI is always global, detailed modeling of distinct memory allocations as separate windows is not performed. Therefore, the collective calls to initialization and finalization therefore record the creation and destruction of the global window definition, respectively (see Figure 3.5a). Allocation of target memory regions are also collective and modeled as such, without creating additional window handle definitions.

ARMCI provides explicit blocking and non-blocking operations. Specifically, it provides a blocking and a non-blocking interface for its put, get, and accumulate operations. Examples of the different RMA operations supported are shown in Figures 3.5b and 3.6. The blocking operations are modeled using the start event for the operation at the beginning of the call and the respective complete-blocking event at the end. The non-blocking operations are modeled using the corresponding start event record in the starting call and the non-blocking completion record in the call ensuring successful completion of the operation. For its non-blocking operations, ARMCI supports explicit and implicit requests. Explicit requests can be used to test or wait for completion of a particular operation, as done for the request-based operations discussed in the MPI

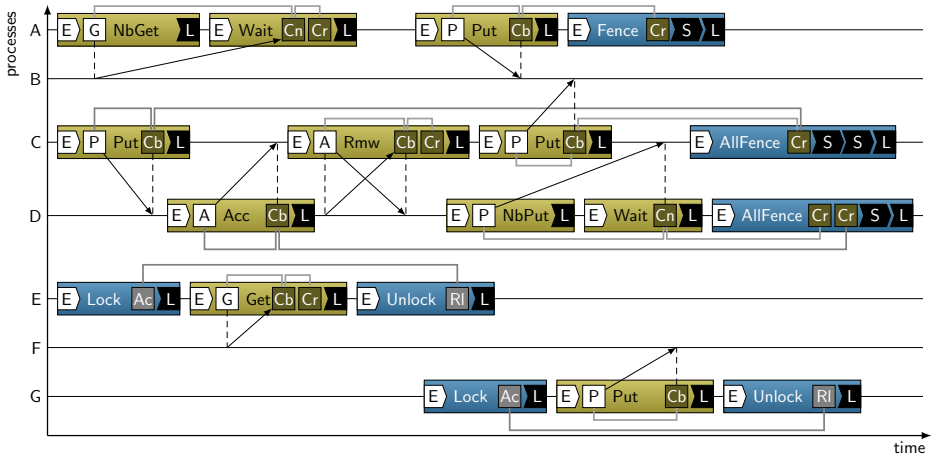
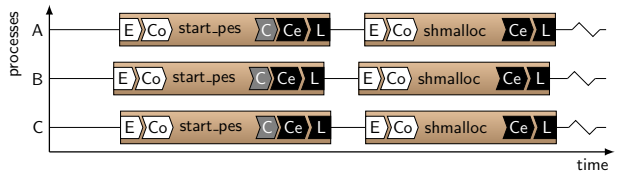


Figure 3.6.: OTF2 event model for ARMCI passive-target synchronization. A fence operation ensures remote completion of any RMA operation to a specified target prior to the call. An all-fence operation ensures the same for all targets of RMA operations prior to the call. The use of locks ensures mutual exclusion of processes, yet does not have any effect on message ordering or completion.

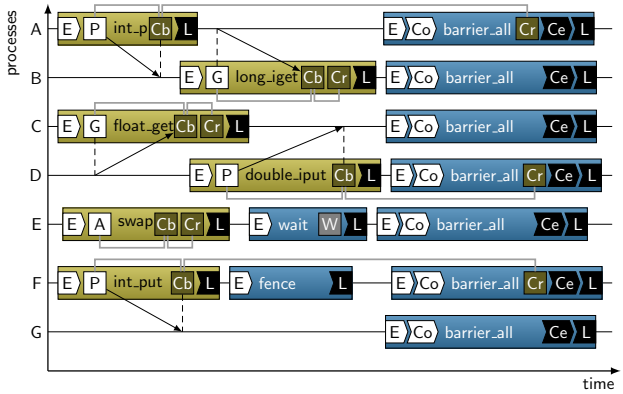
sub-model. Implicit requests are used to wait for the completion of multiple requests at once, either per target or for all pending requests. For put operations, developers can ensure remote completion of pending requests using fence calls. Unlike the similar synchronization function of MPI, the fence in ARMCI is a passive-target call. That is, an origin process can call it independently of the target or any other process. A call to `ARMCI_Fence` will ensure remote completion on a single target (process A), a call to `ARMCI_AllFence` (process C) on all targets since the last synchronization. A call to the collective `ARMCI_Barrier` implies calls to `ARMCI_AllFence` on all processes. Thus, additionally to synchronizing all processes in the communication context, all put operations are ensured to have completed remotely as well. The atomic operations—fetch-and-add and swap—are defined with a blocking interface only. Therefore, a specific operation is modeled by records for the atomic operation start and the corresponding blocking completion (processes C and D). The lock-based mutual exclusion routines ensure that the lock is acquired before they return to the application. The measurement system therefore records the lock acquisition right before leaving the function call. The unlock function places the lock-release event after successful release before returning to the user application.

3.1.3. SHMEM

Unlike MPI, the SHMEM interface is not standardized. Initially designed by Cray for their T3D systems, it was copied over the years by other vendors and experienced API changes as it was ported to different platforms. As a result, a variety of different flavors of SHMEM exist today. The most prominent are Cray SHMEM, SGI SHMEM, and OpenSHMEM. The former two are actively supported communication libraries on CRAY and SGI platforms, respectively.



(a) Initialization and memory allocation.

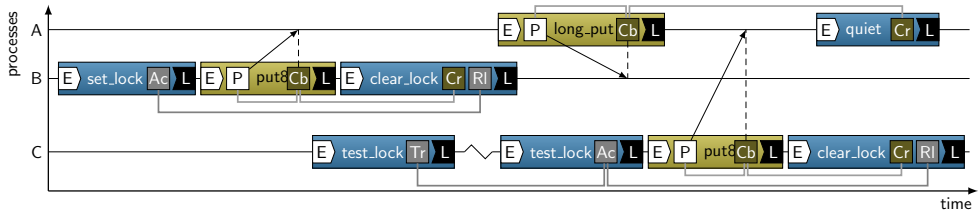


(b) Active-target synchronization.

Figure 3.7.: OTF2 event model for OpenSHMEM collective and active-target synchronization. Initialization and memory allocation is a collective operation and modeled as such. The global communication context is created during initialization, yet no destruction event is written, as finalization is implicit on program termination. OpenSHMEM does not define non-blocking operations, thus operations are modeled by a start event and a corresponding completion event. Barriers enforce remote completion of prior RMA operations.

The latter is a recent community-driven development to consolidate the different functionalities provided by the varying SHMEM interfaces with the objective of providing a single standardized interface to SHMEM that incorporates all capabilities of the different SHMEM flavors assuring application portability across different platforms.

Similar to ARMCI, SHMEM only supports a single global communication context. A single window handle is sufficient and implicitly created during initialization, as shown in Figure 3.8. Processes communicate based on *symmetric objects*. Symmetric objects have the same type, size, and address across all connected processes. With symmetric objects, a process can compute addresses of remote objects, purely based on the properties of their local counterparts. SHMEM allows applications to dynamically allocate and resize symmetric objects using collective function calls. The current OpenSHMEM 1.1 specification [36] only defines a blocking interface to its RMA operations. This means that a remote-memory operation returns upon local completion of the call. For put operations, the specification explicitly states that this does not guarantee remote completion and the message ordering of subsequent put calls without further synchronization is not guaranteed.



(a) Passive-target synchronization.

Figure 3.8.: OTF2 event model for OpenSHMEM passive-target synchronization. A call to `shemem_quiet` ensures remote completion of RMA operations issued prior to the call. Locks ensure mutual exclusion and also complete pending operations remotely.

To enable internal optimizations, SHMEM provides separate function calls for specific native datatypes of C/C++ and Fortran as well as generic remote memory access calls operating on type-opaque buffers. For brevity, Figure 3.8 only shows representative function calls of their respective flavor and omits the common prefix `shmem_`. As part of the general communication routines, SHMEM defines three such flavors each for put and get routines, respectively. First, it defines calls for single element buffers, named `shmem_TYPE_(g|p)` for different `TYPE`s. These calls are only available for native types in the C/C++ interface excluding the `char` type. Second, it defines calls for contiguous buffers, named according to the pattern `shmem_TYPE_(get|put)` for different `TYPE`s. These calls are also available for C/C++ `char` as well as Fortran native types. Finally, calls for strided buffers exist, named according to the pattern `shmem_TYPE_i(get|put)` for different `TYPE`s. According to their blocking behavior, each call region contains a put or get record, respectively, as well as an event recording blocking completion, as shown in Figure 3.8. No additional synchronization is needed to ensure local completion of the respective calls.

To ensure remote completion, SHMEM provides two different types of barriers. First, a call to `shmem_barrier_all` ensures all prior put operations are completed remotely and also synchronizes the processes in that no process will continue execution before the last process joined the call. Second, a call to `shmem_barrier` enables the same on a subset of the processes. The subset is defined by a tuple of the lowest rank in the synchronizing processes, a logarithmic stride (base 2) between consecutive processes, and the total number of synchronizing processes. Calls to `shmem_fence` enforce ordering of put calls relative to the synchronizing calls, but do not guarantee remote completion. Calls to `shmem_quiet` enforce the remote completion. Puts issued before the call are guaranteed to be written to the symmetric memory before puts issued after the call. The first call, `shmem_fence`, ensures this ordering on a per-destination basis, the second, `shmem_quiet` across all programming elements. This is a weaker claim than remote completion, and therefore no remote completion events are recorded in these calls.

Atomic operations in SHMEM are guaranteed to directly complete remotely. That is, an atomic function call will ensure that subsequent accesses of the modified memory on the target process will see the modification. SHMEM also provides a function to observe modification to local memory by remote processes. Using `shmem_wait` or `shmem_wait_until` blocks a process until the corresponding condition is satisfied. For both functions, a wait-change record stores the time the condition was satisfied.

SHMEM also enables users to guarantee mutual exclusion using locks. Similar to the locking semantics of ARMCI, the calls to `shmem_set_lock` return when the lock was successfully acquired. Figure 3.8a shows how in this case an acquire-lock event is placed before leaving the set-lock call (process A). Like in the other interfaces discussed, a call to unlock ensures local and remote completion of all pending operations. The call to `clear_lock` therefore contains remote-completion events for the transfers issued during the lock epoch. Additionally to the classic lock and unlock functionality, OpenSHMEM also defines a test-lock functionality. As demonstrated in the figure on process B, the call returns to the user when the lock is currently held by another process, in which case a try-lock event Tr is placed before the subsequent leave event of the call. In case the lock is not held by another process at the time of the test, the lock is acquired by the process and an acquire-lock event Ac is placed before the subsequent leave event.

3.2. Defining wait states

Wait states in inter-process communication occur when one process waits for a specific action of another process before it proceeds. As one-sided communication involves only a single active process, wait states may not be as obvious as with other communication paradigms. Nevertheless, wait states do occur and can be formally defined using an event model, such as the previously detailed OTF2 model. While the communication in one-sided interfaces does not involve the target process explicitly per definition, the synchronization of the data transfers may or may not involve the target explicitly. These two main classes of synchronization in one-sided communication are called *active target* and *passive target* synchronization. As not all one-sided communication interfaces define both active and passive target synchronization, the wait-state pattern described in this section only apply to interfaces with the corresponding functionality.

The remainder of this section introduces the terminology used to define wait-state patterns in general and provides and an overview of all wait-state patterns in one-sided communication as shown in Figure 3.9. In particular, it recapitulates the wait-state patterns for active-target synchronization (Wait-State Patterns 1 to 5, shown with white background) as described in prior work [63, 86, 99]. Furthermore, it introduces additional wait-state patterns for passive-target synchronization (Wait-State Patterns 6 and 7, shown with gray background) and a new synchronization-complexity indicator contributed by this thesis.

3.2.1. Terminology

To describe complex phenomena in the execution of a parallel program and their impact on the overall application performance, it is necessary to define basic building blocks that can subsequently be used and combined to define more complex interactions. The most fundamental building block of event-based performance analysis is the *event* itself.

Definition 1 (Event) With $p \in P$ identifying a process p out of the set of all processes P, an event $e_p^i \in E_p \subseteq E$ models the ith change of a state in the execution of a program on process p, with E_p being the set of all state changes recorded on process p and E the set of all state changes recorded. Examples of such changes are the entering or leaving of a function call and

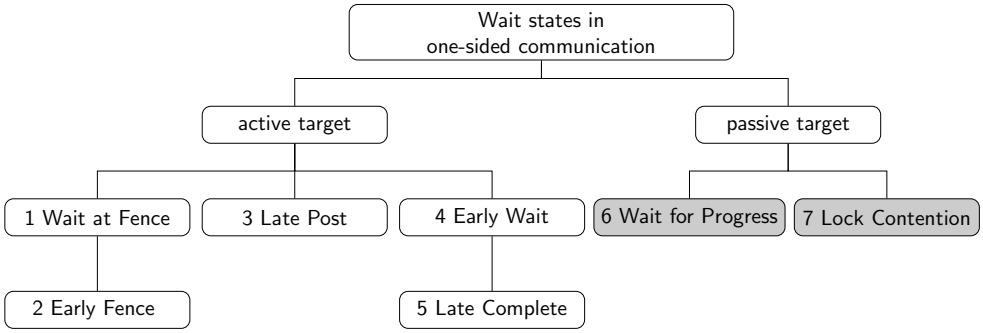


Figure 3.9.: An overview of wait-state patterns in one-sided communication. Patterns marked in gray constitute previously unstudied wait-state patterns.

sending or receiving a message. The distinct time of any specific event in the program execution is denoted by the function

$$\text{Time} : E \to \mathbb{R} \tag{3.1}$$

□

The execution of a program on a process p can thereby be modeled as an ordered sequence of n events $E_p = (e_p^1, e_p^2, \ldots, e_p^i, \ldots, e_p^n) \in E$, where the events are ordered according to their time in the execution. To create a contiguous view on the application execution over time using this sequence of discrete events, the time between two consecutive events is defined as an *activity*.

Definition 2 (Activity) The ith activity a_p^i on a process p describes the time between two subsequent events e_p^i and e_p^{i+1} in the process-local, ordered sequence of events E_p. This implicitly defines the process-local, ordered sequence of activities A_p of size $n - 1$. As common in high-performance computing, each process $p \in P$ shall execute only a single activity on a single compute element at any given time. Each activity is tied to a specific contiguous region in the application code called the *call-path*, which is referenced by the function

$$\text{Callpath} : A \to C \tag{3.2}$$

Furthermore, for activities defined by subsequent enter and leave events to a specific code region, the start and end time, respectively, is denoted by the functions

$$\text{Enter} : A \to \mathbb{R} \tag{3.3}$$
$$\text{Leave} : A \to \mathbb{R} \tag{3.4}$$

Finally, the duration of an activity is defined by

$$\text{Duration} : A \to \mathbb{R} \tag{3.5}$$

□

Evidently, Equations (3.3) to (3.5) are just short-hand for more elaborate usage of Equation (3.1), as the following examples show.

$$\begin{aligned} \text{Enter}(a_p^i) &= \text{Time}(e_p^i) \\ \text{Leave}(a_p^i) &= \text{Time}(e_p^{i+1}) \\ \text{Duration}(a_p^i) &= \text{Time}(e_p^{i+1}) - \text{Time}(e_p^i) \end{aligned}$$

In the event model used in this thesis, the communication routines measured are atomic or opaque in the sense that the model does not record any additional call-path information below the captured communication interface function in the measurement. This means that the time spent in a communication activity is defined directly by the timespan between its enter and leave event and is not interrupted by other activities. In other words, the communication routines are by definition leaf nodes in the call tree.

Definition 3 (Wait state) An activity a_p^i suffers a wait state, if and only if both of the following two conditions are met:

1. The completion of a_p^i depends on a remote event e_q^k denoting the kth event on process q.
2. The time of event e_q^k lies within the timespan of activity a_p^i, i.e.,

$$\text{Enter}(a_p^i) < \text{Time}(e_q^k) \leq \text{Leave}(a_p^i)$$

The amount of waiting time in the ith activity a_p^i on process p is expressed by the amount of time spent waiting for a specific event e_q^k on process q and is denoted by the function

$$\omega : A \to \mathbb{R}$$

which is further defined by

$$\omega(a_p^i) = \text{Time}(e_q^k) - \text{Enter}(a_p^i) \tag{3.6}$$

□

3.2.2. Active target synchronization

Active target synchronization explicitly involves the target process in the completion of the remote memory operation on the target. The term was coined by the MPI standard, which also defines the most comprehensive active-target synchronization schemes. Any of these synchronization schemes is a potential candidate for wait states, as processes explicitly wait for remote events to occur. This can mean that either an origin process may wait for a target to allow the remote memory access, or the target may wait for an origin to complete it.

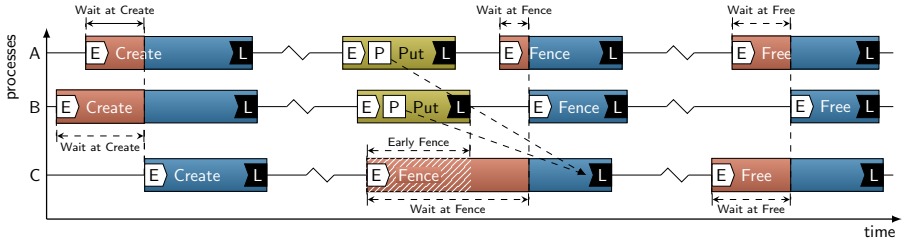


Figure 3.10.: The *Wait at Create/Fence/Free* and *Early Fence* patterns. Waiting time is marked in red with the corresponding wait-state pattern name. Due to process synchronization inherent in the operations, all participating processes have to wait for the last one to enter the call. The *Early Fence* pattern indicates time that a target process spends waiting in a fence synchronization while origins still have to complete their operations [2].

Wait at Create/Fence/Free and Early Fence

The most general active-target synchronization scheme is the use of collective memory fences. That is, all processes using a specific window handle have to perform this synchronization and cannot complete the call before every other process started as well. As such, this scenario is very close to the *Wait at N×N* wait-state pattern discussed in Section 2.2. Additionally, this wait-state pattern also occurs at other collective calls of one-sided communication interfaces, namely the collective allocation and deallocation of memory segments suited for remote memory access.

Wait-State Pattern 1 (Wait at Create/Fence/Free) Let a_p^i be the activity of a collective RMA synchronization call on process p (i.e., creating, freeing, or synchronizing a window). Further let the activities $a_0^{\tilde{i}}, a_1^{\tilde{i}}, \ldots, a_{n-1}^{\tilde{i}}$ be the corresponding activities on the processes 0 to $n-1$ sharing the window handle.

Then the local waiting time ω_p on process p is the difference of the latest enter time among the processes 0 to $n-1$ and the local enter time [86].

$$\omega_p = \max\left(\text{Enter}(a_0^{\tilde{i}}), \text{Enter}(a_1^{\tilde{i}}), \ldots, \text{Enter}(a_{n-1}^{\tilde{i}})\right) - \text{Enter}(a_p^i)$$

□

Figure 3.10 shows the different scenarios of this wait-state pattern. During window creation, process C arrives late and processes A and B accumulate waiting time. Later, during a collective fence operation, process B arrives late while A and C wait. Finally, process B arrives late again during window destruction.

The cause of wait states at collective operations are work or communication imbalances among the processes, as will be discussed in more detail in Chapter 4. As a sub-pattern of the *Wait at Fence* pattern, the *Early Fence* pattern is an additional indicator for imbalance. It identifies those parts of the wait state, where the local process already started the completing synchronization call, but one or more origin process have not issued their respective accesses of this epoch.

Wait-State Pattern 2 (Early Fence) Let a_p^i be the activity of a collective fence synchronization call on process p. Further let the $a_q^{\text{Last Op}}$ the activity of the last remote-memory operation of process q to the local process p. Then, the last remote-memory access to the local process in that epoch is determined by the maximum leave time of all individual last remote memory access operations across all corresponding processes. Furthermore, the waiting time ω_p is the difference between the end of the last remote-memory access operation and the start of the local synchronization activity a_p^i, if the end of the last remote-memory access operation is later than the start of the local synchronization activity [86]:

$$\omega_p = \max\left(\text{Enter}(a_p^i), \text{Leave}(a_0^{\text{Last Op}}), \ldots, \text{Leave}(a_{n-1}^{\text{Last Op}})\right) - \text{Enter}(a_p^i)$$

□

The premature epoch completion of *Early Fence* can be resolved in different ways. First, the origin process could issue the access operation earlier, by changing the order of function calls in the epoch. This might also be beneficial to the overlapping the communication with the user function's computation, but is only possible if there actually is time spent in other calls during the epoch that are independent from the RMA operation in question. Second, instead of changing the origin side of the epoch, the target may delay the start of the synchronization by executing useful independent functions prior to the synchronization. Third, as part of a general load balancing strategy, the work between origin and target could be balanced. This would resolve the complete wait state and usually is the best way to address the underlying problem.

Late Post

Besides the collective synchronization, MPI also defines a more fine-grained *general active-target synchronization*. Here, synchronization is not performed among all processes of a communication context, but only among a runtime-defined set. Each origin process specifies the target processes it will access in a given access epoch and subsequently only synchronizes with the corresponding targets. Similarly, each target specifies the origin processes it allows to access its memory on start of the exposure epoch. In a correct MPI program, the two groups have to complement each other: for each origin process p that has a target process q in the target group starting the access epoch, target process q has to have p in its origin group starting the exposure epoch. Although this synchronization is much more lightweight than the collective synchronization, wait states can still form at different points of this synchronization scheme.

MPI does not strictly define which calls will block until the remote process sends an acknowledgement, thus, the analysis algorithm has to apply a heuristic to determine which of the calls involved suffered a wait state. During pattern detection, it first identifies the call that led to the process synchronization before it actually computes the waiting time.

Wait-State Pattern 3 (Late Post) Let a_q^{Post} be the activity of a post call on a target q in a general active-target synchronization. Let a_p^i the activity of a corresponding synchronization or communication function on the origin process p. If and only if, the start of a_q^{Post} falls between

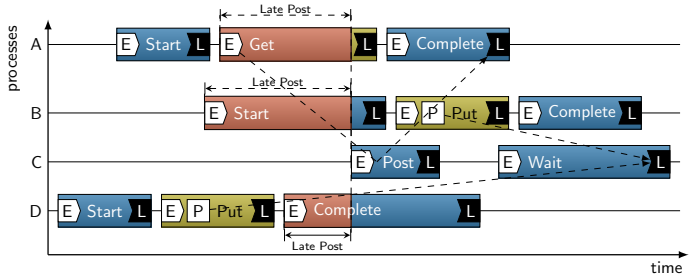


Figure 3.11.: The *Late Post* pattern. Origin processes may block in different calls when accessing the exposed window on the target. Process A does not block when starting the access epoch, but blocks in the subsequent RMA operation. Process B blocks directly when starting its access epoch. Process D does not block the starting of the access epoch and buffers the RMA operation. It then blocks before it can complete the access epoch [2].

the start and end of the local activity a_p^i on process p, the waiting time ω_p is determined by the difference between the start of remote activity a_q^{Post} and the start of the local activity a_p^i [86].

$$\omega_p = \begin{cases} \text{Enter}(a_q^{\text{Post}}) - \text{Enter}(a_p^i) & \text{, if Enter}(a_p^i) < \text{Enter}(a_q^{\text{Post}}) \leq \text{Leave}(a_p^i) \\ 0 & \text{, otherwise} \end{cases}$$

□

Figure 3.11 shows the different synchronization scenarios possible with general active-target synchronization. In this example, processes A, B, and D wait for process C, which starts its exposure epoch too late, causing waiting time on the origins. An access epoch can generally not be completed before the exposure epoch has started. The MPI standard does not explicitly define when a process has to wait for the start of the exposure epoch. Implementations may buffer the start and any subsequent operations and perform all necessary actions in the complete call. Note that while an implementation usually only uses one of those strategies (rendering the concrete scenario in Figure 3.11 unlikely), it is not guaranteed that implementations restrict themselves to a single strategy. The analysis algorithm therefore applies a heuristic to infer the actual point of blocking from the recorded event traces. The different potential locations for wait states are: (1) the start call of the access epoch (process B), (2) one of the remote memory access operations (process A), or (3) the complete call of the access epoch.

Kühnal et al. originally also defined the *Early Transfer* wait state. It described the special case of a *Late Post* wait state, when the wait state occurred in an RMA operation (process A). The explicit name for this special case is purely based on reasons of internal classification of wait states as communication and synchronization-related, not explicitly on differences in the semantics of the pattern definition. To avoid confusion and ease the interpretation of analysis results, the *Early Transfer* wait-state pattern is therefore now assimilated into the *Late Post* pattern.

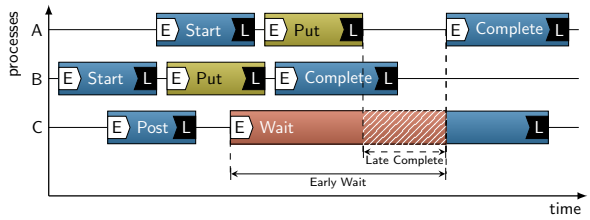


Figure 3.12.: The *Early Wait* and *Late Complete* pattern. Process C enters the wait early, before processes A and B signaled the completion of their access epoch. Process A is the last process to signal completion. The time between the last remote memory operation and the complete call is marked as *Late Complete* time [2].

Early Wait and Late Complete

Target processes in general active-target synchronization may potentially wait for origin processes to complete their access epochs. To ensure that modifications to the window memory become visible in the target's memory consistently, the target needs to perform a memory fence after closing the exposure epoch. To ensure that all modifications of the exposure epoch occur before the memory fence is issued, the target needs to receive acknowledgements from all origins that they have completed their accesses.

Wait-State Pattern 4 (Early Wait) Let a_p^{Wait} be the activity of a wait call on target process p. Further, let a_q^{Complete} be the activity of the corresponding complete call on an origin process q. Then, the waiting time of the target ω_p is the difference of the latest start of completion activities across all corresponding origins and the start of the targets wait activity a_p^{Wait}, if the end of the last completion activities is later than the start of the local synchronization activity [86]:

$$\omega_p = \max\left(\text{Enter}(a_p^{\text{Wait}}), \text{Enter}(a_0^{\text{Complete}}), \ldots, \text{Enter}(a_{n-1}^{\text{Complete}})\right) - \text{Enter}(a_p^{\text{Wait}})$$

□

Figure 3.12 shows the *Early Wait* wait state in a scenario with two origins. The two origin processes A and B complete their access epochs while the target process C is already waiting for their acknowledgement.

The *Late Complete* wait state is a sub-pattern of *Early Wait*. It is an indication that waiting time may be reduced by moving the complete closer to the last remote memory operation to enable the target to complete the exposure epoch earlier. However, this resolution cuts both ways, as the separation of a non-blocking start of a communication and its completion may enable communication-computation overlap. Moving the completion closer to the operation itself may therefore reduce the possible overlap, reducing the overall gain from this optimization. Schneidenbach et al. [141] pointed out this weakness in the MPI definition and proposed the separation of data transfer completion and the acknowledgement of the completion of the current access epoch.

Wait-State Pattern 5 (Late Complete) Let process q be the last process to access the target process p. Further, let $a_q^{\text{Last Op}}$ be the activity of the last remote-memory access operation from process q to p and a_q^{Complete} be the activity of the corresponding completion call on q. Then, the waiting time ω_p on process p due to a delayed complete on process q is the difference between the end of the last remote-memory access operation and the start of the complete [86].

$$\omega_p = \text{Enter}(a_q^{\text{Complete}}) - \text{Leave}(a_q^{\text{Last Op}})$$

□

In Figure 3.12, the *Late Complete* is indicated by the hatched area in the *Early Wait* wait state on process C. As can be seen, the *Early Wait* wait state may be reduced by reducing the gap between the put on process A and the corresponding complete.

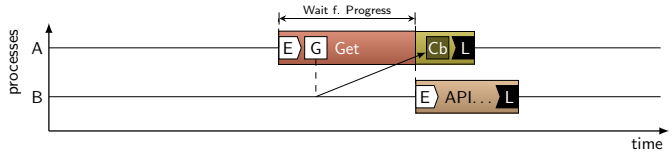
Pairwise RMA process synchronizations

MPI and other one-sided communication interfaces often define several synchronization functions for pending one-sided communication operations. Some often synchronize pending operations to multiple targets or ensure operations from multiple origins complete on the target. Depending on the defined synchronization semantics, this may involve a large number or remote processes and even synchronize processes that did not exchange data. Such synchronization may be necessary, as a target cannot distingiush whether a remote origin still has pending RMA operation that have not arrived yet or whether the are none in the first place, until some form of information exchange clarifies this. For example, if the synchronization function on the target has to guarantee completion of remote accesses to its window, it has to collect information from all potential origin processes.

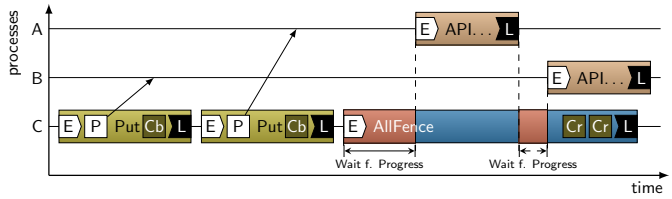
The *RMA pairwise synchronizations* metric is a performance metric that counts such synchronizations in a pairwise manner. This means, for each target the origin needs to synchronize with (and vice versa) one synchronization is counted. , leading to multiple counts within single synchronization functions if itfor multiple origins and targets. The sub-metric *Unneeded pairwise synchronizations* counts the number of such synchronizations that were not needed to complete an RMA operation, indicating that there is potential work to be saved. The user needs to evaluate the this metric and put it into perspective of the time needed for the synchronization. A high number of unneeded synchronizations correlates may indicate the cause for a high time spent in the synchronization.

3.2.3. Passive target synchronization

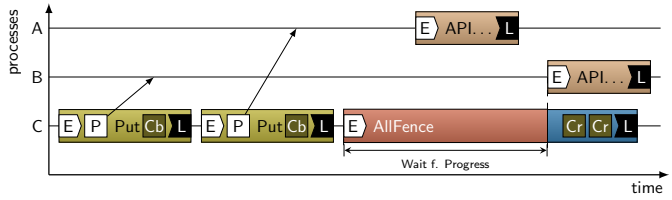
In passive-target synchronization, the target process does not actively participate in the synchronization. Necessary synchronization is either performed by hardware on the target without explicit software interaction, taken care of by additional communication threads, or implicitly performed when the target calls into the communication library. Points of synchronization are therefore not as obvious as with active-target synchronization. Nevertheless, synchronization points may still exist implicitly. To infer whether a wait state occurred, inter-process information needs to be evaluated.



(a) Depending on a single event.



(b) Depending on multiple events with *no-overlap* heuristic.



(c) Depending on multiple events with *last-call* heuristic.

Figure 3.13.: The *Wait for Progress* pattern. Calls may either depend on a single remote event to occur, such as the `Get` operation (a), or on multiple remote events, such as the `AllFence` synchronization using two different heuristics (b and c). The *no-overlap* heuristic only classifies those times that do not overlap with progressing remote regions as waiting time, which is a conservative lower bound. The *last-call* heuristic classifies the time up to the start of the last remote call as waiting time, serving as an upper bound.

Wait for Progress

When hardware support for one-sided data transfers is not available, an additional thread—a so-called *progress thread*—can be used to ensure communication progress. However, not all one-sided communication libraries use progress threads, as they may have a negative impact on communication latency in general, as locking of critical regions in the messaging library will be needed to ensure consistency. Furthermore, it may lead to cache thrashing, with further negative impact on messaging but also on application performance in general. On the other hand, not relying on a progress thread to advance the communication, but rather advancing any pending communication every time the application calls into the communication library avoids these issues. However, this may lead to waiting times when blocking operations on the origin process wait for communication progress on the target. The question of whether to employ a progress thread or not usually depends on the communication patterns and phases expressed by a specific application. For one-sided communication runtime implementors as well as users of one-sided

communication interfaces it is therefore important to identify the costs of not using a progress thread to make an informed decision on which path to choose for a specific application and platform.

Direct information on message progress is mostly not available from the communication library. Therefore, the decision whether remote progress occurred or not has to be made following specific heuristics. Two different approaches can be used to estimate the waiting time of a local process: (1) the *no-overlap* heuristic and the (2) *last-call* heuristic.

The no-overlap heuristic only considers the times where the local activity does not overlap with any of the remote activity it depends on. The motivation for this assumption is that some unknown portion of the overlapping time is productively used for data transfer completion. Not to falsely classify productive time as waiting time, this heuristic will usually underestimate the real waiting time in this pattern and serves as a lower bound. The last-call heuristic considers the full time from the beginning of the local activity to the start of the latest progressing remote activity as waiting time. The motivation for this assumption is that no matter how much time was spent productively with other remote processes, the local activity will always have to wait for the start of the last remote activity it depends on. Therefore, this heuristic may overestimate the real waiting time, but in that serves as an upper bound for the real (unknown) waiting time.

Wait-State Pattern 6 (Wait for Progress) Let a_p^i be an activity on process p that requires communication progress on one or more remote processes to complete. Further let $\mathcal{D}$ be the set of remote activities a_q^k that provide progress ($\rightsquigarrow$) for a_p^i. $\mathcal{D}$ does not contain more than one activity of the same process, implying that each activity $a_q^k \in \mathcal{D}$ completely fulfills its part of the required progress for activity a_p^i.

Then, the waiting time ω for the *no-overlap* heuristic is defined as the sum of all time spans that do no overlap with any of the activities $a \in \mathcal{D}$, whereas the waiting time for the *last-call* heuristic is defined as the difference between the start of the latest activity a_q^k in $\mathcal{D}$ and the start of the local activity a_p^i:

$$\mathcal{D} = \left\{ a_q^k \mid a_q^k \rightsquigarrow a_p^i \right\}$$

$$\forall a_q^k, a_{q'}^{k'} \in \mathcal{D} : (a_q^k \neq a_{q'}^{k'} \Rightarrow q \neq q')$$

$$\text{No-overlap heuristic} : \omega_p = \text{Duration}(a_p^i) - \text{Duration}(\left(\bigcup^{\mathcal{D}} a_q^k \right) \cap a_p^i)$$

$$\text{Last-call heuristic} : \omega_p = \max_{a_q^k \in \mathcal{D}} \left(\text{Enter}(a_q^k) \right) - \text{Enter}(a_p^i)$$

□

The calls to the library API are tracked on the target side, but not explicitly associated with any origin-side call. The target therefore does not know which of the API calls provided remote progress and for which origin potential progress was granted. To identify these, the target needs the time of the enter and leave event on the origin.

Figure 3.13 shows two typical situations for the *Wait for Progress* wait state. Figure 3.13a shows how the get operation on process A cannot complete without remote progress given by process B. Such a scenario is also common to remote-memory access operations that perform computations at the target, where the operation is not directly supported by the communication hardware. Figures 3.13b and 3.13c show a scenario where the two put operations of process C complete locally, but the application also needs to ensure remote completion calling an allfence operation. Process C therefore has to wait for confirmation from processes A and B that can be given at the earliest when the respective remote process calls into the communication runtime after the start of the synchronization activity on process C. The two figures provide a visualization of how the two heuristics—*no-overlap* and *last-call*—classify the waiting time differently.

Lock Contention

As one-sided communication may be capable of directly modifying remote memory without target intervention, synchronization mechanisms are needed to ensure data consistency. For such purposes, one-sided communication interfaces usually provide lock-based mutual exclusion. In lock-based synchronization, critical code sections can only be entered if a corresponding lock is acquired. With exclusive locks, only a single process can acquire a lock at any single moment; with shared locks, multiple origin processes can acquire a lock concurrently. As shared locks only block other exclusive locks until release but allow concurrent shared locks to be acquired, they present less chance of wait states and should be preferred in scenarios where the target memory is not modified.

Naturally, the acquisition of any type of lock may lead to a wait state. Depending on the one-sided communication interface, the specific time of acquisition of the lock may be unknown. For example in MPI, `MPI_Win_lock` may return before the lock is actually acquired. Subsequent remote memory access operations have to ensure the lock is acquired before they can be completed. As those operations are also allowed to return early, the lock acquisition may be delayed until the call to `MPI_Win_unlock`.

A lock can be seen as a shared resource itself, with multiple processes competing for access to it. The state when a process experiences wait states or delays due to other processes' access to the same shared resource is called *contention*. Wait states in the lock-based mutual exclusion mechanisms are therefore a special case of the general *resource contention* that can also be experienced with other shared resources, such as file systems or network devices.

In general, the *Lock Contention* wait state occurs when a process requests a lock that is currently held by another process. It then has to wait for the release of the lock by that process. In general, the information on the synchronization dependencies between the origin processes is not directly available to the individual origin processes.

Wait-State Pattern 7 (Lock Contention) Let a_p^i and a_q^k be the activities of a passive-target synchronization or remote-memory access operation on origin processes p and q. Assume that a_p^i cannot complete before the acquisition of the corresponding lock held by process q. Assume further that q releases the lock at the end of activity a_q^k.

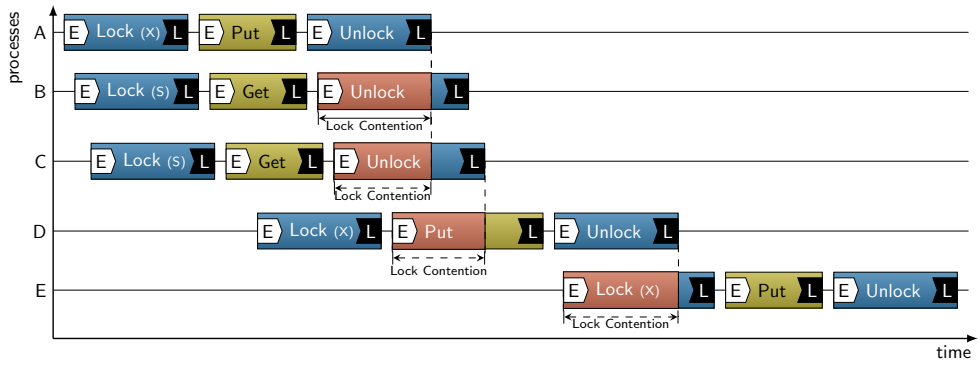


Figure 3.14.: The *Lock Contention* pattern. When multiple processes access the same window on the same location, lock access chains build up. In this example, write accesses are protected by exclusive locks (x), whereas read accesses are protected by shared locks (s). Depending on the one-sided interface, the moment of lock acquisition may not be known explicitly, but can only be inferred by checking the time of release of previous lock owners.

Then, the waiting time ω_p is defined as the difference between the start of a_p^i and the end of a_q^k.

$$\omega = \begin{cases} \text{Leave}(a_q^k) - \text{Enter}(a_p^i) & \text{, if Enter}(a_p^i) < \text{Leave}(a_q^k) \leq \text{Leave}(a_p^i) \\ 0 & \text{, otherwise} \end{cases}$$

□

Figure 3.14 shows the different acquisition scenarios for MPI passive-target synchronization. Similar to the relaxed blocking semantics of MPI general active-target synchronization, MPI passive-target synchronization only requires the unlock to guarantee completion of all pending RMA operations, as long mutual exclusion requirements of the requested locking types are met. This implies that implementations may not block a locking call (as shown with processes B and C) or subsequent RMA operations (as shown by process D), as long as their access to the target is postponed until the actual access is granted by the target. Process A is the first process to acquire an exclusive lock. As it did not have to wait, it is not relevant for the analysis process to know which of the calls actually acquired the lock. In this example, processes B and C both request a shared lock. They can acquire the lock concurrently, but have to wait until it is released by process A. Process D requests an exclusive lock, and has to wait for process C, being the last process to release the shared lock. Additionally, it acquired the lock in the remote memory operation (Put) within the lock epoch. Finally, process E requests the lock, yet directly blocks until the lock is released by process D.

3.3. Scalable detection of wait states in one-sided communication

Chapter 2 introduced the scalable replay-based analysis capabilities of the Scalasca toolset. This section introduces the extensions to this infrastructure necessary to facilitate the identification

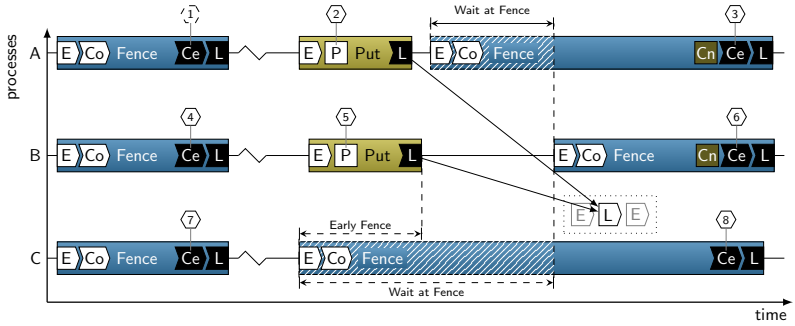


Figure 3.15.: Communication during the detection of wait states in collective operations. The honeycomb markers indicate specific events that are involved in the analysis process; the numbering is arbitrary but unique. At every collective-end event of a fence operation, any previous access and exposure epochs are opened and new ones are opened on all participating processes. Available memory in the corresponding window of process C is indicated by a dotted rectangle. Transfer of information on the end of RMA operations is indicated by solid arrows.

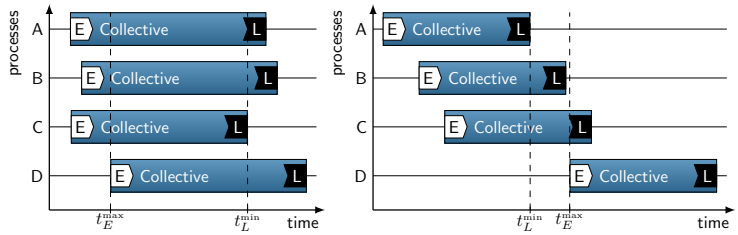
and quantification of the wait states in one-sided communication discussed in the previous section, specifically the scalable detection of wait states in active-target synchronization [2] and progress-related wait states of passive-target synchronization [4]. Furthermore, it discusses the detection of wait states due to lock contention.

3.3.1. Active-target synchronization

Wait-state patterns in active-target synchronization comprise five individual patterns. Three of these are distinct patterns, and the remaining two are sub-patterns that allow a deeper insight into the state of its corresponding parent pattern. Active-target synchronization comprises collective and group-based synchronization.

Collective synchronization

The wait states in collective memory allocation and synchronization are similar to the *Wait at N×N* pattern discussed in Chapter 2. As synchronization or allocation information needs to be exchanged among all processes a full synchronization of the window's communication context is implied. As the operations under measurement were collective, the analysis can also use collective communication during the replay. Figure 3.15 shows an example scenario involving fence synchronizations. The honeycomb markers indicate events that are involved in the detection of waiting time in collective active-target synchronization. Assuming a semantically correct application, the analysis algorithm can assume that all processes will eventually process the local events modeling the same instance of the collective synchronization. The analysis algorithm can therefore also dispatch collective communication calls at collective events (①,③,④,⑥,⑦, and ⑧) without the risk of deadlock. Similar to the detection of wait states in collective communication,



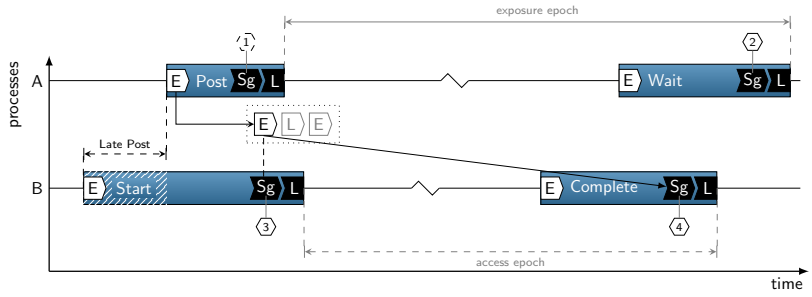
(a) Full overlap of processes. **(b)** Partial overlap of processes.

Figure 3.16.: Heuristic for process synchronization detection in implicitly synchronizing collective calls. Waiting time is only computed in cases of overlap of all region instances across the processes.

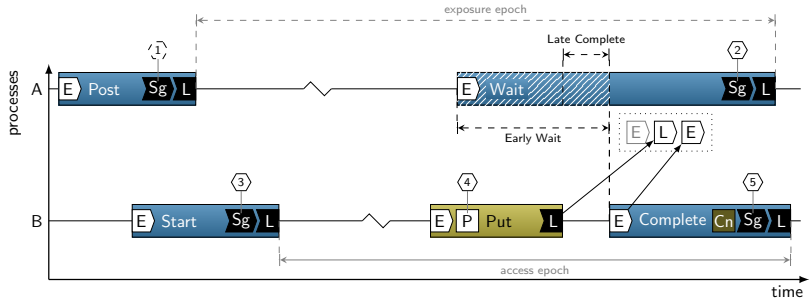
a reduction determines the maximum enter and the minimum leave timestamps of all corresponding calls. Combining these values with local enter and leave timestamps, each process can then compute its local waiting time.

However, process synchronization in these collective calls is implicit. That is, its main purpose is to complete all pending RMA operations locally and remotely. If no additional information is provided through function parameters, the collective operation needs to complete a full information exchange (N×N) or a barrier operation to obtain the missing information—implying full synchronization of all participating processes. Some one-sided communication interfaces, such as MPI, allow the user to provide additional information about the synchronization and dependencies needed by the runtime. This may result in partial instead of full process synchronization. There is no interface-level way to know whether the call was fully synchronizing or not. Thus, the analysis uses a heuristic as shown in Figure 3.16 to decide whether a full synchronization took place. If and only if the maximum enter time $t_E^{\max}$ is smaller than the minimum leave time $t_L^{\min}$, all processes share a common time span in this call and the call is considered to be synchronizing. If the time of the last process entering the collective call is earlier than the time of the first process leaving the call, the call is considered to be fully synchronizing and waiting time is computed for the individual participating processes. If it is later, no waiting time is computed.

While the wait state of collective synchronization uses collective communication, the amount of waiting time in the sub-pattern *Early Fence* uses one-sided communication to transfer the necessary data. During an access epoch, each origin process tracks the last leave time of its accesses to the different targets at RMA operation events (⟨2⟩ and ⟨5⟩). Reaching the end of the access epoch at the collective-end events (⟨3⟩,⟨6⟩, and ⟨8⟩), before closing the access epoch using a fence synchronization, each origin uses one-sided communication—namely an atomic accumulate operation—to send the timestamp of the last operation's leave events to the appropriate window location of the corresponding targets. A fence synchronization allows access to and from every process sharing the same window, thus, the target does not have to know which origins will send their access timestamps. The *accumulate* operation, as defined by MPI, allows the use of reduction operators with the one-sided operation. Using the maximum operator, the atomic operation will replace the value in the target window if the new value is larger than the value already residing there. After all one-sided operations have been dispatched, another fence syn-



(a) *Late Post*



(b) *Early Wait & Late Complete*

Figure 3.17.: Communication during the detection of wait states in general active-target synchronization.

chronization ensures the completion of all pending operations. Each target can then evaluate its window buffer to obtain the maximum leave time of any one-sided operation accessing its target buffer during the original application run.

Group-based synchronization

Similar to the detection of wait states in fence synchronization, the analysis recreates access and exposure epochs of the original application to detect the *Late Post*, *Early Wait*, and *Late Complete* wait states. Figure 3.17 shows example synchronization scenarios involving general active-target synchronization. Triggered by synch-group Sg events, origin and target processes issue calls to the same type of call it was recorded in. In Figure 3.17a, the target process A issues a call to `MPI_Win_post` at event ① and `MPI_Win_wait` at event ②, while origin process B issues a call to `MPI_Win_start` at event ③ and `MPI_Win_complete` at event ④. All necessary information to issue the respective calls is stored in the corresponding sync-group event. In Figure 3.12, processes A and B proceed similarly using events ①, ②, ③, and ⑤, respectively. This way, each target process explicitly exposes a window—shown as a dotted rectangle—that origin processes can then use to put and get data to and from, respectively.

The *Late Post* wait state (see Figure 3.17a) needs to identify the latest beginning of an exposure epoch on potentially multiple targets. To do so, each target copies the local post enter timestamp into its local window memory before it starts the exposure epoch (①). The origin processes can then issue a get operation (③) on this memory location to collect and compare all necessary timestamps. After the access epoch is closed upon processing the corresponding events of the complete call on each origin process (④), it can determine the largest timestamp locally. Then, each origin process searches for a possible overlap of this timestamp with local operations of the corresponding access epoch. If found, the algorithm assumes that the overlapping call waited for the exposure epoch of the target to start and computes the waiting time. If not found, the algorithm assumes that none of the potential synchronization points lead to a wait state and no waiting time is computed.

To detect the *Early Wait* pattern (see Figure 3.17b), the target needs to obtain the largest enter timestamp of the completion call across the origin processes. In contrast to the get operation used for the *Late Post* detection, each origin process transfers its enter timestamp of the `MPI_Win_complete` call using an accumulate operation to the target window. Using the maximum reduction operator, the one-sided operations can compute the largest enter timestamp in place.

The *Late Complete* is a sub-pattern, similar to the *Early Fence* wait state discussed earlier. The analysis needs to identify the time between the last RMA operation targeting the local process and the corresponding complete. Again, each origin process uses the accumulate operation with the maximum reduction operator to compute the largest leave timestamp in the target memory. The resulting time of the last RMA operation to the local process and the last complete operation timestamp may not stem from the same origin. This is still an intended behavior, as the *Late Complete* pattern is an indicator for potential waiting-time reduction by moving the completion of the epoch closer to the last operation. If the completion of an access epoch on one origin can be moved prior to the last RMA operation of another origin, the completion on that other origin process would be the remaining cause for an *Early Wait* wait state. However, moving the completion of a non-blocking call closer to its start may effectively prevent computation-communication overlap. Therefore, the amount of waiting time classified as *Late Complete* must be regarded as an upper limit to the optimization potential of closing the gap between the last operation and the corresponding complete.

3.3.2. Passive-target synchronization

In the message-replay algorithm used by Scalasca for point-to-point and collective communication, the events available in the process-local event traces enable triggering communication as well as inferring implicit information about the individual communication, e.g., that data received at a receive event contains timestamp information of the corresponding send on the remote process. As the measurement system only records explicit involvement of a process using instrumented library calls, and the target process in passive-target synchronization does not perform any of that kind, it becomes evident that the target process cannot generate any viable events during measurements. This implies that the target process is unable to trigger specific actions on certain events to aid the origin's detection of wait states. Using active messages, the replay-based analysis can perform the data exchange necessary to identify wait states

in passive-target synchronization scenarios. Specifically, the key requirements for the extended communication infrastructure are the support of (1) inter-process communication not relying on specific target-side event records, (2) communication on paths not explicitly recorded, (3) asynchronous information exchange to enable runtime optimizations during event processing, and (4) the support of target-side execution of arbitrary tasks based on the communicated message. To enable the detection and quantification of wait states in passive-target synchronization scenarios, this thesis contributes active-message extensions to Scalasca's initial event processing infrastructure that fulfills these requirements to explicitly support the exchange of inter-process information in the absence of explicit event records to trigger specific communication and also enable communication between arbitrary process pairs.

Active-message analysis infrastructure

Two-sided and collective communication are often used as the data exchange layer in cooperative algorithms where the receiver actively receives a message. It knows and decides how the received data needs to be processed locally. The knowledge of how the data needs to be processed emerges from the context containing the explicit reception of the data. However, for unexpected messages, the receiver does not have such a context, and therefore does not know how to process them in the application. Any target-side processing of the data therefore needs to be part of the content of the message. As already briefly touched in Chapter 1, *active messages* encode the context with the message, enabling target-side execution of code after the one-sided transfer succeeded. For specific message types, a *message handler* can be registered that will process a message ad hoc at the receiver. The sender, knowing for which context it provides data in the message, also sends the appropriate handler selection with the message. This effectively decouples the message from its receiving context, as the receiver can provide the appropriate message context by calling the handler selected by the sender.

To enable this, all processes need to agree on a specific set of message handlers to be used for communication and how they are encoded. The complexity of actions that can be encoded into a message largely depends on the communication interface and framework used. Some interfaces have a rather restricted set of message handlers that focus on the notification of the data and sending an acknowledgement of transfer completion back to the sender. Others allow more complex message handlers, such as remote procedure calls.

Three classes form the cornerstones of the active-message framework: (1) A *runtime* class, which defines the abstract framework messaging interface; (2) *request* classes, which define how data is to be transferred between processes; and (3) *handler* classes, which define the data to be packed by the sender, and how it is processed by the receiver.

The runtime class is designed as a singleton object, which is accessible throughout the application using it. It is agnostic to the concrete actions that need to be taken to transfer or process messages. It provides a high-level interface to send and receive requests, and executes the correct handlers on message reception. The active-message runtime class provides an interface to advance communication independently of the current execution context. This enables the use of a variety of progress engines at the target. The current implementation uses explicit polling on the target, which is integrated into the event replay mechanism. This means that the target explicitly calls

into the runtime system to advance any pending requests. It polls at least once per event, which enables communication progress as the analysis progresses through the event trace. Additionally, it provides capabilities to continuously advance the communication while waiting at collective synchronization points.

Request classes define all concrete actions needed to transfer data between processes. For each communication interface used by the active-message framework, a distinct request class needs to be implemented. In its current state, only a request class for MPI messaging is implemented, yet, support for further communication interfaces can easily be achieved by implementing further request classes. Note the MPI-based requests can also be used to analyze applications that do not use MPI themselves, such as ARMCI- and SHMEM-only applications. Additional request classes are therefore only necessary in cases where MPI is not available or a different implementation is desired.

Handler classes define, independent of the message transfer, which data is packed at the sender and how it is unpacked and processed at the receiver. An application using the active-message framework, such as Scalasca's parallel analyzer, needs to derive specific handlers for each distinct task on the receiver side. Each handler provides an interface to `pack` all necessary data, as well as an `execute` method, which is executed by the active-message runtime class on request reception.

Using this flexible active-message framework, Scalasca's parallel analysis supports two previously unstudied wait states in passive-target synchronization. Each of these wait-state scenarios relies heavily on the framework's one-sided interface as described in the following sections.

Wait-state detection

In passive-target synchronization, two distinct wait-state patterns are detected: (1) the *Wait for Progress* and (2) the *Lock Contention* wait-state patterns. This section details the algorithms employed to identify instances of these patterns in the event trace and quantify their severity.

Wait for Progress. Contrary to its initial publication [4], the detection of *Wait for Progress* wait states does not distinguish between single and multiple dependencies. This is due to the fact that, while the original single-dependency algorithm summarized the waiting time already at the target, the more advanced analyses described in Chapter 4 require per-instance data to be transferred back to the origin. As the detection of this wait-state pattern in the case of multiple dependencies also requires per-instance data to be sent back to the origin, the two algorithms were merged into a single algorithm covering both cases.

The detection follows four phases:

① The origin sends a search request to the target.

② The target receives the request and identifies the activity by executing the corresponding request handler.

③ The target sends a response request to the origin, containing the corresponding event data.

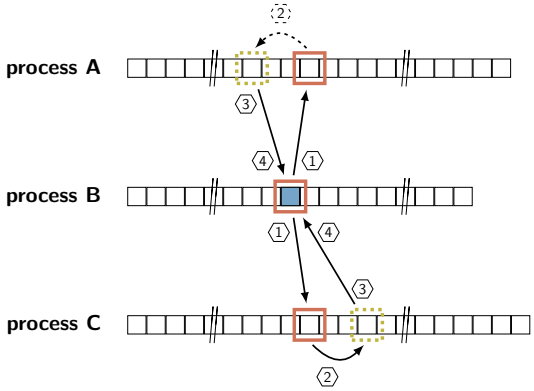


Figure 3.18.: Detection workflow for the *Wait for Progress* wait-state pattern.

④ The origin receives the request and either temporarily stores the event data or computes the waiting time once all responses have arrived.

These four phases are handled by two active-message handlers, as illustrated in Figure 3.18: `WfpRequest` and `WfpResponse`. Process B, as the origin, packs two active-messages using the `WfpRequest::pack` handler and sends them to processes A and C, respectively (①). Processes A and C, receive these messages while processing an arbitrary event. They execute the message handler `WfpRequest::execute` using the message content and search for a corresponding event in their own event stream (②). In this scenario, process A finds the relating event (⋯) prior to the current event position, whereas process C identifies an event that has not been processed by the replay. Upon finding the appropriate events, the `WfpRequest` handler then creates a new active message using `WfpResponse::pack` handler (③). Processes A and C then send those messages back to process B. Process B receives those messages and executes the target side `WfpResponse::execute` handler using the message content. The message handler postpones the wait-state analysis, buffering the information about the remote activities until all responses are received (④). Once the target processes A and C have send their local information to the origin process B, it performs the wait-state analysis using both heuristics. The detection of waiting time with the *last-call* heuristic is straight forward. From all remote activities reported, the last one is identified and the difference between its start and the start of the local activity is classified as the waiting time. The detection of waiting time with the *no-overlap* heuristic is more complex, and the general algorithm is given in Algorithm 1 in pseudo-code. The origin stores all remote activity data until it received data about all dependencies. The data is held in a list of all activities sorted by the enter time of the activity. Once the information about all remote activities is available, the algorithm can investigate each activity and identify possible overlap with the local activity. As all activities are sorted by their enter time, the algorithm computes the overlap in a single pass over all activities. For this it maintains a reference time t^{ref}, which indicates the time up to which overlap with the local activity a_p has been checked. It is initialized with the start time of the local activity. For each activity a_q in the list, the enter time is then compared to the reference time. If the enter time of the remote activity is larger than the reference time, the time span between them is classified as a waiting time. If the

Algorithm 1: Compute *Wait for Progress* using the *no-overlap* heuristic

Input: Local activity a_p^i
Input: List ActivityList of remote API activities, ordered by enter time
Output: Waiting time ω

```
ω := 0;
t^ref := Enter(a_p^i);
foreach activity a_q^API in ActivityList do
    if t^ref < Enter(a_q^API) then
        ω := ω + (Enter(a_q^API) - t^ref);
    end
    if t^ref < Leave(a_q^API) then
        t^ref := Leave(a_q^API);
    end
    if t^ref ≥ Leave(a_p^i) then break;
end
return ω
```

leave time of the remote activity is larger than the reference time, the reference time is updated to the remote activity's leave time. Once the reference time exceeds the local leave time, the computation is done. The total time spent in activity a_p without overlap with the corresponding remote activities is returned as the waiting time ω.

Lock Contention.

The time between requesting or acquiring a lock and its release by a process is called a *lock epoch*. Lock contention leads to so-called *contention chains*, where multiple processes wait in line to acquire ownership of the lock. To identify lock contention in one-sided communication, the analysis needs to process the lock acquisition and release times of a lock. For one-sided communication interfaces with blocking lock semantics, such as ARMCI and SHMEM, this is directly modeled by the respective events, as discussed in Sections 3.1.2 and 3.1.3. For these interfaces, the only activities of the *lock epoch* that need to be considered are the respective activities for acquiring and releasing the lock. For one-sided communication interfaces with non-blocking semantics, such as MPI, the lock acquisition time has to be computed during the contention analysis. For these interfaces, all remote-memory access activities of the lock epoch need to be available to the analysis process.

In a contention scenario, two or more origin processes wait for the access to a specific resource, but do not explicitly know of each other. To identify contention, however, the individual local information on the processes have to be compared to each other to (1) identify the order of accesses to the resource and (2) quantify potential waiting time due to a blocked resource. To enable contention analysis for one-sided communication interfaces, all origin processes need to gather the needed information at a well-known location. It is important to note that any

Algorithm 2: Compute *Lock Contention*

Input: Stack EpochStack of lock epochs ordered by descending lock-release time
Output: Waiting time ω_p

```
currentEpoch ← pop(EpochStack);
while NotEmpty(EpochStack) do
    previousEpoch ← pop(EpochStack);
    a_q ← GetReleaseActivity(previousEpoch);
    a_p ← FindBlockedActivity(currentEpoch, a_q);
    if Enter(a_p) < a_q ≤ Leave(a_p) then
        ω_p ← Leave(a_q) − Enter(a_p);
        SendContentionInfoTo(q);
        SendContentionInfoTo(p);
    end
    currentEpoch ← previousEpoch;
end
```

deterministic location will work, as long as all origin processes locking the same resource choose the same location. A simple first choice is the process that owns the resource being locked.

The analysis follows two phases: (1) gather epoch information; and (2) compute and distribute waiting time information [3]. In the first phase, each origin process caches the relevant lock epoch data until it processes the lock-release event. Then, it creates an active-message request, packed with the lock epoch information, and sends it to the target process. On the target side, the request unpacks the data and stores it for later retrieval. As the active messages coming in from the individual origin processes do not generally arrive in the same order the lock was acquired and released by the application, the target needs to save incoming lock epochs until it reaches a point where it can safely assume to possess the full information on all lock epochs relevant for the contention analysis. Such points are reached at each collective or group-based synchronization point of the window or at collective synchronization points that synchronize at least all processes of the window's communicator. At these points the active-message runtime of Scalasca ensures that all requests are processed before continuing with the analysis. Independent of the locking semantic, all one-sided communication interfaces ensure completion of pending events with the release of the lock. Therefore, the release time of the lock is an indicator for the actual locking order during the application measurement. The target therefore stores the individual lock epochs provided by the origin processes in a data structure sorted by the release time of lock in the respective epoch.

Once the analysis system can assume all distributed lock epochs have been collected and inserted into the queue, it can start its contention analysis as described by Algorithm 2. The pseudo-code given assumes a stack-like data structure to simplify the notation of the algorithm. Furthermore, process p denotes the waiting process, whereas process q denotes the process that p is waiting for. As the epochs are ordered in reverse-chronological lock-release order, the last lock epoch in the contention chain is processed first. The epoch information (currentEpoch) is taken from the stack to initialize the algorithm. Then, while more epoch information is available on the stack, another epoch (previousEpoch) is taken from the stack to compute the waiting time. For

the previous epoch, we identify the activity a_q that released the lock, and a_p the waiting activity within the current epoch. This is done by finding overlap with one of the synchronization or remote-memory access operations within the current epoch with the lock-release activity of the previous epoch. If an overlapping activity is found, the waiting time is computed by the difference between the leave event of a_q and the enter event of a_p, and the respective information is sent to both processes q and p. Then, the algorithm moves on to the next epoch in the stack. The algorithm finishes when no further epochs are in the stack, which means the top of the contention chain is reached; the first epoch never suffers a wait state itself.

Summary

In event-based performance analysis, metrics are computed using an event model. Such an event model defines the records that store the parameters of specific state changes in the application, where they are recorded, and which semantic connections individual records have among each other. This chapter introduces the OTF2 generic event model for one-sided communication, suitable to cover the communication semantic of multiple one-sided communication interfaces in high-performance computing. It demonstrates the applicability of this model by defining specific sub-models for MPI one-sided communication, ARMCI, and OpenSHMEM. Using such event models, wait-state patterns in one-sided communication are specified both for active-target and passive-target synchronization. Furthermore, this chapter demonstrates how these wait-state patterns can be identified in event traces using Scalasca's replay-based analysis method. Wait states in active-target synchronization are identified using one-sided communication transferring the timestamp information needed in a replay of the recorded access and exposure epochs of the application. Wait states in passive-target synchronization are identified using an extension of the event-replay method, inspired by active messages, allowing the flow of information on other than the recorded communication paths.

4. A unified model for critical-path detection and wait-state formation

Interpreting raw performance data can be challenging and time-consuming for developers, diminishing their productivity. The identification and quantification of wait states is already a first step in forming an understanding of the measured data. However, wait states still only represent the symptoms of the actual performance problems and they often materialize in significant temporal and spatial distance to their root cause. Without knowing where a specific performance problem stems from, developers potentially spend a lot of time tracing back a wait state to its root cause. To retain productivity, performance tools need to identify a wait state's root causes automatically, and thus significantly reduce the time needed to optimize parallel applications.

Böhme et al. introduced two methods that proved valuable in identifying optimization potential as well as root causes of wait states in the application. The first method—critical-path detection alongside related performance indicators [24]—identifies those parts of the parallel execution of an application that dominate its overall runtime as well as indicating how much performance gain can be expected through better load balancing. Using this information, developers can easily find suitable optimization targets. The second method—root-cause analysis [26]—spatially and temporally tracks waiting time back to differences in execution time across processes within synchronization intervals. Such differences are also called *load imbalance* or *delays*. The *load* of a process in this context is the time needed to perform intentional tasks—both in computation and communication. By identifying the imbalances that ultimately cause wait states cascading through the execution, both in spacial and temporal terms, developers can focus on the cause rather than the symptom of a performance problem.

Both methods build on the identification of critical points of process interaction, so-called synchronization points. Moreover, both methods rely on complete synchronization information to be available to the analysis. In case of incomplete synchronization information (i.e., a synchronization point remains undetected), both methods may produce results ranging from small inaccuracies to full loss of sensible analysis results. For the root-cause analysis, missing process-synchronization information may lead to identifying the wrong call path as the root cause of one or more wait states. For the critical-path analysis, missing process-synchronization information may lead to identifying the wrong call paths and processes to be on the critical path. It is therefore of prime importance to uncover all points of synchronization among the processes of a parallel application to provide valuable insight into the application behavior. For applications using multiple communication paradigms including one-sided communication, either method can only succeed if one-sided communication is fully integrated.

The contribution of this thesis in this regard is threefold:

1. It defines all critical points of process interaction in one-sided communication by specifying the location of synchronization points and newly introducing contention points.
2. It enables the detection of root causes and the computation of the critical path in one-sided communication scenarios.
3. It defines contention as an additional cause of wait states and extends Böhme et al.'s work to provide a unified cost model for root-cause detection.

The remainder of this chapter introduces a unified cost model to identify root causes for both synchronization-based and contention-based wait states, including a recapitulation of Böhme et al.'s original definition [23] and the necessary extensions contributed by this thesis. Furthermore, it demonstrates how one-sided communication scenarios can be integrated into Böhme et al.'s detection of the critical path. Finally, it provides details on the scalable implementation used to detect both root causes and the critical path in parallel applications using one-sided communication.

4.1. Root causes of wait states in one-sided communication

Böhme et al. investigated root causes for collective and point-to-point communication [26]. All wait states that occur in these communication paradigms are based on process synchronization and caused by load imbalance—in both computation and communication—among the corresponding processes. One essential characteristic of all synchronization-based wait states is that one process is waiting for one or more other processes to begin a corresponding activity. Contention-based wait states differ in this regard, as a process waits for another process to end a certain activity. To understand this fundamental difference, it is necessary to first introduce the definition of delays and their contribution to wait-state formation by Böhme et al.. Definitions that only differ in their notation are marked as a direct citation. Definitions that needed to be extended for the support of contention-based wait states are marked as based on Böme et al.'s work.

4.1.1. Delay

For synchronization-based wait states, Böhme et al. show that wait states occur due to load imbalance between processes [26]. To detect the function calls responsible for those imbalances, the scope of the search needs to be determined. In synchronization-based wait states, a process waits for a remote process to start a corresponding activity, such that both of the processes can proceed with their local execution. Böhme et al. call such a point of waiting *synchronization point*.

Definition 4 (Synchronization point [23]) A synchronization point $\mathcal{S} = (a_p^i, a_q^k)_{\mathcal{S}}$ is a tuple of two activities $A \times A$, where a_q^k is the activity containing the event of process q the completion of activity a_p^i on process p depends on. □

After a mutual synchronization point, two or more processes are regarded as synchronized, that is, they both continue their local execution at the same time. If at the next synchronization point of the same processes one of these processes is waiting for the other, the cause for this lies in the interval between those points, the *synchronization interval*.

Definition 5 (Synchronization interval [23]) A synchronization interval $\zeta_{(\mathcal{S}',\mathcal{S})}$ is defined by two subsequent synchronization points S' and S between the same processes p and q. For the determination of the interval itself it is not important which of the processes p and q is the waiting process.

$$\zeta_{(\mathcal{S}',\mathcal{S})} = \left((a_q^{k'}, a_p^{i'}), (a_p^i, a_q^k) \right)_\zeta = \left((a_p^{i'}, a_q^{k'}), (a_p^i, a_q^k) \right)_\zeta$$

Synchronization intervals may be shared by multiple processes, as is the case with collective communication. In the wait-state formation model these are treated as pairwise superpositions of individual intervals. When computing the pairwise load imbalance in the shared interval, all participating processes are taken into account. □

Within such a synchronization interval, the cause for the delayed arrival of the later process can be either some intentional computation on the part of the later process, or some wait state on that process due to an overlapping synchronization interval with another process. To separate the time spent in intentional computation from the time spent waiting, Böhme et al. compute the *waiting-time-adjusted execution time* of every call path in the synchronization interval.

Definition 6 (Waiting-time adjusted execution time [23]) Let $c \in C$ be a specific call path of all call paths C present in the application. Furthermore, let $a_p^{i'}$ and a_p^i be the first and last activity of an interval of activities on process p, respectively. Then, the waiting-time-adjusted execution time is a function

$$d : A \times A \times C \to \mathbb{R}^{\geq 0}$$

returning the sum of time spent on activities cleared of any waiting time of the given call path within the given interval of activities on the given process.

$$d_{(a_p^{i'}, a_p^i)}(c) = \sum_{\substack{j=i'+1 \\ \text{Callpath}(a_p^j)=c}}^{i-1} \left(\text{Duration}(a_p^j) - \omega(a_p^j) \right) \tag{4.1}$$

□

By definition, the difference between the waiting-time-adjusted execution time d and the overall time spent in the interval is the sum of all waiting time in the interval. Investigating the differences of time spent per call path in one or more activities on corresponding processes, Böhme et al. define a *delay* as a cause of synchronization-based wait states [26].

Definition 7 (Delay [23]) Let $\zeta = \left((a_p^{i'}, a_q^{k'})_{\mathcal{S}'}, (a_p^i, a_q^k)_{\mathcal{S}} \right)$ be the synchronization interval defined by the synchronization points $\mathcal{S}'$ and $\mathcal{S}$ of processes p and q. Then, the excess execution time δ is the positive difference in waiting-time-adjusted execution time between two call paths

on different processes in the corresponding synchronization interval.

$$\delta_\zeta(c) = \begin{cases} d_{(a_q^{k'}, a_q^k)}(c) - d_{(a_p^{i'}, a_p^i)}(c) & \text{, if } \quad d_{(a_q^{k'}, a_q^k)}(c) > d_{(a_p^{i'}, a_p^i)}(c) \\ 0 & \text{, otherwise} \end{cases} \tag{4.2}$$

This excess execution time is called *delay*. □

Böhme's wait-state formation model identifies the cause of a wait state by definition as an overload of the later process, rather than an underload of the waiting process. As a consequence, the model does not consider negative differences in call paths—where the waiting process spent more time than the delayed process—as these do not contribute to but rather mitigate the consequences of a potential delay on process q in the corresponding synchronization interval.

The wait-state patterns of active-target synchronization in one-sided communication are—similar to the wait states investigated by Böhme et al.—all synchronization-based. Also the *Wait for Progress* wait-state pattern can be seen as a special case of a synchronization-based wait state. The original wait-state formation model therefore naturally covers these wait-state patterns as well.

4.1.2. Contention

Although, many of the wait-state patterns in one-sided communication are synchronization-based, the *Lock Contention* wait-state pattern differs in key aspects from its synchronization-based siblings. The event on the causing process does not mark the beginning of a concurrent activity, but the end of an activity that must not be concurrent with the activity on the waiting process, a characteristic that was not covered by Böhme et al.'s wait-state formation model. To express this difference in the wait-state formation model, the points of process synchronization due to contention are defined as *contention points*.

Definition 8 (Contention point) A contention point $\mathcal{C} = (a_p^j, a_q^l)_\mathcal{C}$ is a tuple $A \times A$, where a_p^j is waiting for a_q^l to end. □

This fundamental difference also has implications on the interval of activities investigated during the root-cause analysis. The two essential differences are: (1) the remote activity on the process causing the contention is included with any existing waiting time in the computation of pairwise imbalance, while the blocked activity on the waiting process is not and (2) the time of the blocked activity excluding its waiting time is part of the next interval. To reflect this, the wait-state formation model is extended by the definition of *pre-contention intervals*.

Definition 9 (Pre-contention interval) A pre-contention interval κ is defined either by consecutive synchronization and contention points or two consecutive contention points between the

same processes p and q.

$$\kappa_{(\mathcal{S}',\mathcal{C})} = \left((a_p^{i'}, a_q^{k'})_{\mathcal{S}'}, (a_p^{j}, a_q^{l})_{\mathcal{C}} \right)_{\kappa} \tag{4.3}$$

$$\kappa_{(\mathcal{C}',\mathcal{C})} = \left((a_p^{j'}, a_q^{l'})_{\mathcal{C}'}, (a_p^{j}, a_q^{l})_{\mathcal{C}} \right)_{\kappa} \tag{4.4}$$

Furthermore, κ^- denotes the set of activities on the waiting process p excluding the synchronization activity and κ^+ denotes the set of activities on the causing process q including the full synchronization activity.

$$\kappa^- = \left(a_p^{i'+1}, \ldots, a_p^{j-1} \right) \tag{4.5}$$

$$\kappa^+ = \left(a_q^{k'+1}, \ldots, a_q^{l} \right) \tag{4.6}$$

□

Corollary 4.1 To avoid contention, the pre-contention interval κ^+ on the causing process q needs to spend less or equal time in activities than the waiting process p in the corresponding pre-contention interval κ^-. □

Delays—as the root cause of synchronization-based wait states—resemble imbalances in the execution time of call paths leading up to the wait state. Identifying them with the ultimate resolution strategy to balance the respective call paths works well for the common use case of point-to-point and collective communication—explicit common data exchange phases between multiple processes. This approach, however, is less suited for contention-based wait states.

Searching for delays in a contention scenario would always emphasize the additional function calls present in κ^+ as a root cause due to the inherent asymmetry of the pre-contention intervals κ^+ and κ^-. As an example, consider the first two processes (A and B) in Figure 4.1 on page 73. The example uses a common implementation strategy for MPI one-sided communication, where a process is allowed to postpone obtaining the lock and performing the subsequent RMA operations until the unlock operation, as discussed in Section 3.2.3. Assume that the y-axis is a common synchronization point for all processes and processes A and B enter the corresponding synchronization function $\texttt{Unlock}_0$ and $\texttt{Unlock}_1$, respectively, at the same time. Only the call to $\texttt{Unlock}_0$ in the pre-contention interval κ^+ on process A would be identified as the root cause, as all other call paths are perfectly balanced between the processes. However, this would be stating the obvious—i.e., the unlock function as the culprit—without any additional insight for the user.

As stated by Corollary 4.1, the resolution for contentions of this kind of wait state is to reduce the length of the pre-contention interval κ^+ in a way that the causing process A can release the lock before process B tries to acquire it. The unified wait-state formation model therefore considers contributions of all activities in the pre-contention interval to the waiting time caused in the same proportions of those activities' contributions to the overall time spent in the interval. Large contributions thus identify higher potential of a function as a viable resolution target for the contention.

4.1.3. Propagation

Wait states can be caused directly by some intentional activity of the application. However, synchronization and pre-contention intervals between different processes may partially overlap and thus influence each other. As such, a wait state in a synchronization interval may delay a process to cause a wait state at the interval's end. Likewise, a wait state in a pre-contention interval may push the activities of a process to overlap with those of another process, causing the contention. Böhme classifies wait states as either *direct* or *indirect* and either *propagating* or *terminal* [26]. This section discusses these definitions and shows how they apply to contention-based wait states.

Definition 10 (Direct vs. indirect wait state [23]) A direct wait state is caused by intentional computation or communication (i.e., load), which does not include waiting time. An indirect wait state is caused by other direct or indirect wait states. □

Seen in isolation and from the end to the beginning, an indirect wait state is part of a chain—the *propagation chain*—that has terminal wait states at the end, potentially leading over indirect wait states to a direct wait state, and eventually some intentional computation or communication at the beginning. A wait state causing further wait states is said to propagate.

Definition 11 (Propagating vs. terminal wait state [23]) A propagating wait state is a wait state that causes other wait states. A terminal wait state is a wait state that does not cause further wait states. □

The following properties of propagation chains follow directly from those definitions:

- At the beginning of any propagation chain are one or more activities with intentional computation and communication—the causes.
- The first wait state in the propagation chain is direct, all following wait states are indirect.
- The last wait state in the propagation chain is terminal, all preceding wait states are propagating.

As single wait states can propagate to multiple other wait states, the combination of multiple propagation chains results in a tree-like structure. More precisely, it is a cycle-free graph with multiple source nodes (the causes) and multiple sink nodes (the terminal wait states). As discussed later in Section 4.3.3, the classification in direct and indirect is done from the cause in the direction to the terminal wait states, while the classification into propagating and terminal wait states is done from the wait states in the direction of their cause. Böhme et al. originally defined these relationships for synchronization-based wait states and they transfer seamlessly to contention-based wait states without further adaptation.

Figure 4.1 shows these relationships in a scenario containing only contention. In this example, all four processes A through D request exclusive access to the same resource and there are no further wait states than the ones shown. The resource accessed by the processes is not shown. Processes A and B concurrently request the resource and A is the first to gain access. Thus, process A does not suffer a wait state. Process B is only second to gain access and has to wait

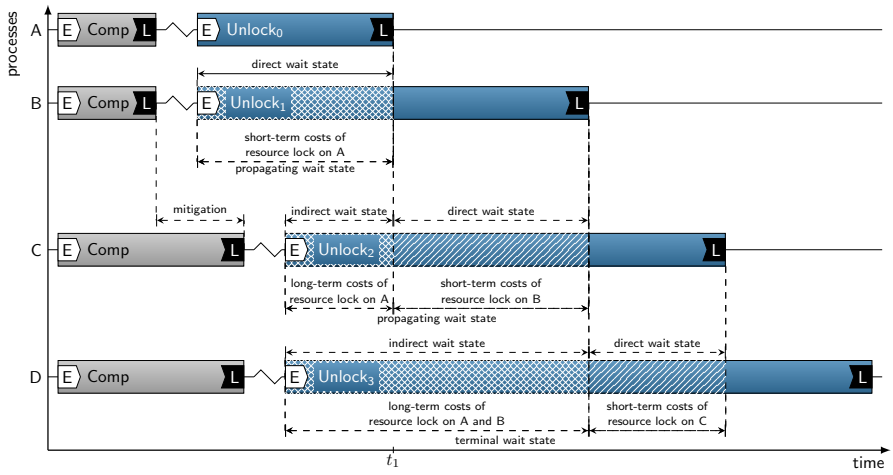
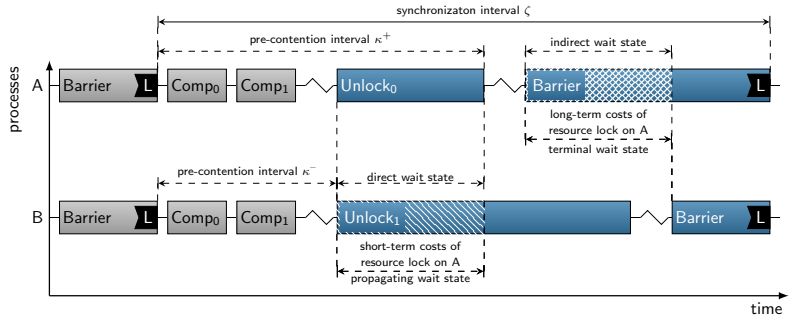


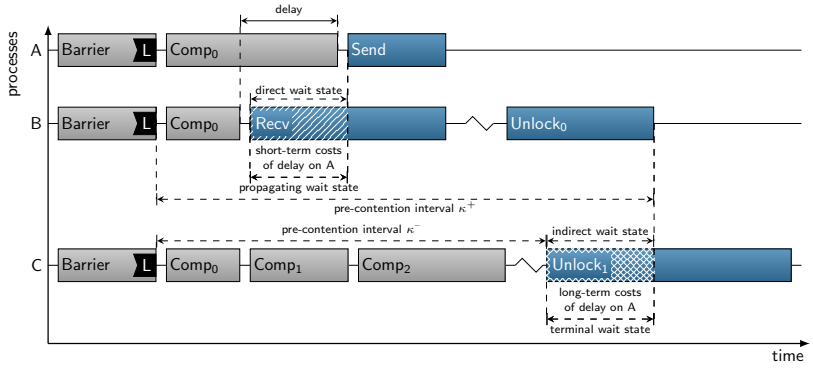
Figure 4.1.: Direct and indirect wait states in resource contention scenarios. Direct wait states materialize as short-term costs. Indirect wait states materialize as long-term costs. Suitable imbalances mitigate short-term costs first, then long-term costs in the order of the distance in the contention chain.

the full span of time process A needs to complete its access to the resource. As process A, in this scenario, did not suffer a wait state since the last synchronization, the cause of process B's wait state needs to lie in activities on process A prior to the resource release—this includes both the lock-protected resource access as well as the activities leading up to it in the pre-contention interval. Process B's wait state is therefore direct. Processes C and D both spend more time in the user function `Comp`, therefore arriving after processes A and B at the contention point, yet still at a time where A and B access the resource. If process B would not have waited for process A, it would have released the resource at time t_1. Any time process C waits for the contended resource prior to t_1 is therefore direct waiting time, as this is caused by intentional execution on process B. The amount of direct waiting time is mitigated by the imbalance between processes B and C in `Comp`. However, process C has to wait even longer than process B to acquire the resource. This additional time is caused by process B waiting for the resource to become available. It is waiting time that process B is not responsible for and therefore classified as indirect. Process D, as the last process in the contention chain, is waiting for process C to release the resource. It has no additional mitigation of contention time compared to process C, as it arrives at the same time at the contention point. Therefore, the direct waiting time is the full duration of process C holding the resource. Additionally, process D also suffers indirect waiting time radiating down in the contention chain from processes A and B. The important property of contention scenarios shown here is that imbalances can mitigate the waiting time.

As an example of how imbalance can mitigate contention, consider a company with a large workforce and a cafeteria. Consider further, that all employees start and end their work at the same time in the morning and evening, respectively—similar to a global barrier synchronization



(a) *Lock Contention* propagating to a *Wait at Barrier* wait state.



(b) *Late Sender* propagating to a *Lock Contention* wait state.

Figure 4.2.: Cost accounting with synchronization and contention-based wait states.

and the beginning and end of a code section. If all employees start their lunch break after four hours of work, long queues will occur at the cafeteria. If some employees start their lunch break earlier and others later, the overall concurrency reduces and less employees have to wait. However, the employees starting the lunch break earlier will have to work longer after the break, and vice versa. Effectively, the employees need to shift work between the before-lunch and after-lunch parts of their work to retain an overall balanced load per day. If some of the contention remains during the lunch break, some of the employees will have to work longer to make up for the time lost waiting.

As seen in this example, contention-based and synchronization-based wait states can influence each other and the resolution of wait states may have non-local effects. Figure 4.2 shows two scenarios where wait states propagate in mixed scenarios of contention and synchronization. Figure 4.2a shows how contention propagates further and causes wait states in the enclosing synchronization interval. The contention-based wait state on process B spans the full length of `Unlock`$_0$ on process A. Assuming the load on processes A and B is equal after their respective unlock function, the waiting time in the barrier on process A closing the synchronization inter-

val is caused directly by the contention present in the synchronization interval. However, the contention on process B is in turn caused by process A blocking the resource, therefore the wait state in the second barrier on process A is indirectly caused by an activity in the pre-contention interval κ^+ on the process A itself.

Just as the contention indirectly caused the wait state at the barrier, a delay can indirectly cause contention. Figure 4.2b shows how a delay first causes a *Late Sender* wait state, which in turn causes a contention wait state later on. In the synchronization interval of processes A and B, the user function $\texttt{Comp}_0$ is imbalanced, and the delay present on process A leads to a wait state on process B. The pre-contention intervals κ^+ and κ^- on processes B and C, respectively, would be perfectly balanced. The waiting time for process C corresponds to the time of the *Late Sender* wait state, delaying the synchronization activity and causing the contention. Note that process C uses additional activities $\texttt{Comp}_1$ and $\texttt{Comp}_2$ as a controlled imbalance to ensure the equal amount of waiting-time-adjusted execution time in the pre-contention interval with process B. In this example, the additional functions act as placeholders to illustrate the general technique of inducing a controlled imbalance to avoid contention. The time spent in the call to $\texttt{Comp}_1$ is equivalent to the non-waiting time of the `Recv` on process B, while the time spent in the call to $\texttt{Comp}_2$ is equivalent to the time needed for the synchronization activity $\texttt{Unlock}_0$. These functions do not necessarily have to be separate function calls, but could also be represented by an imbalanced $\texttt{Comp}_0$ (which is balanced between B and C in this example). Furthermore, it is important to note that not all wait states at the end of contention chains are terminal. For the sake of simplicity in this example, the lock contention in Figure 4.2b is regarded in isolation. The waiting time classified as terminal wait state on process C may very well propagate in a larger scope. However, this cannot be assessed from the scenario depicted in the figure alone.

When discussing the interaction of contention-based and synchronization-based wait states, it is further important to note that any controlled imbalance that is introduced to resolve a contention, may have to be matched by another controlled imbalance inverse to the original to balance the overall load in the encompassing synchronization interval, as described in the cafeteria example.

4.1.4. A unified cost model

Böhme et al. use a cost model to quantify the impact of individual root causes of wait states. The costs represent a call path's contribution to the formation of wait states during application execution. It is computed for a specific wait state from the aggregated time of activities in that call path within the wait state's synchronization interval. Contributions of wait states within the interval are propagated to their sources, whereas contributions of waiting-time-adjusted activities resemble a root cause. As synchronization-based wait states are caused by delays, Böhme et al.'s original model only handles these as root causes. As the cause of contention does not fit the original definition of delays, this thesis extends the original cost model to support both types of wait states in a single unified cost model.

To distinguish call paths with mostly local influence on wait-state creation from those with far reaching effects, Böhme et al. define *short-term* and *long-term* costs. Short-term costs comprise

only the contributions to direct wait states. Long-term costs comprise all contributions to both direct and indirect wait states.

Definition 12 (Short-term costs [23]) The short-term costs of a process-call-path tuple are the sum of waiting-time contributions of that process and call-path to direct wait states throughout application execution.

$$\text{Short-term costs} : P \times C \to \mathbb{R} \tag{4.7}$$

□

Definition 13 (Long-term costs [23]) The long-term costs of a process-call-path tuple are the sum of waiting-time contributions of that process and call-path to direct and indirect wait states throughout application execution.

$$\text{Long-term costs} : P \times C \to \mathbb{R} \tag{4.8}$$

□

Delay costs

The definition of short-term and long-term costs of delays are part of Böhme et al.'s initial cost model. They first specify how contributions to direct wait states are computed as part of the short-term costs and then specify how the costs of propagating wait states are computed as part of the long-term costs.

Commonly, a synchronization interval can comprise several causes of wait states. Additionally, delays are indicators of imbalance across processes, and the costs need to be proportional to the amount of imbalance caused by the delays. Therefore, Böhme et al. define a scaling factor for the computation of costs for a specific delay or wait state that ensures that costs are attributed according to the respective delay's or wait state's size. This way, long delays and wait states receive larger costs than short delays and wait states. This scaling factor is computed specifically for each investigated synchronization interval and involves the aggregated time of all delays and wait states in the interval.

Definition 14 (Delay-cost scaling factor [23]) For a synchronization interval $\zeta = \left((a_p^{i'}, a_q^{k'}), (a_p^i, a_q^k)\right)_\zeta$, let the sum of all delays $\hat{\delta}_\zeta$ and the sum of all waiting times $\hat{\omega}_\zeta$, be defined as:

$$\hat{\delta}_\zeta = \sum_{c \in C} \delta_\zeta(c) \qquad \hat{\omega}_\zeta = \sum_{a_q^{k'} < a_q^l < a_q^k} \omega(a_q^l) \tag{4.9}$$

Then, the scaling factor for any delay $\delta_\zeta(a_q^l)$ in the synchronization interval ζ is defined by:

$$\text{Delay-cost scaling factor} := \frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta} \tag{4.10}$$

□

Using this scaling factor for each individual delay present in the synchronization interval, Böhme et al. define the short-term delay costs.

Definition 15 (Short-term delay costs [23]) The total short-term delay costs of a call path c on process q is defined by the sum of all contributions to wait states $\omega(a_p^i)$ over all synchronization intervals ζ where call path c constitutes a delay (i.e., $\delta_\zeta(q,c) > 0$).

$$\text{Short-term delay costs}\,(q,c) := \sum_{\zeta=\left((a_p^{i'},a_q^{k'}),(a_p^i,a_q^k)\right)_\zeta} \frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta}\delta_\zeta(q,c)\,\omega(a_p^i) \tag{4.11}$$

□

It is important to note that the short-term costs are computed only for the delays δ and not for any wait states within the synchronization interval. This means that in the presence of wait states in the synchronization interval, a portion of the waiting time is not distributed as costs. Therefore, the overall short-term costs attributed to call paths may be smaller than the overall waiting time detected for an application.

The costs for wait states are computed as part of the long-term costs, which comprise the full contributions to direct and indirect wait states. As such, the overall long-term costs are equal to the overall waiting time detected for an application.

Definition 16 (Long-term delay costs [23])

$$\text{Long-term delay costs}\,(q,c) := \sum_{\zeta=\left((a_p^{i'},a_q^{k'}),(a_p^i,a_q^k)\right)_\zeta} \frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta}\delta_\zeta(q,c)\,\varphi(a_p^i) \tag{4.12}$$

□

As with short-term delay costs, the long-term delay costs are only computed for call paths with delays δ. To reflect the contributions to indirect wait states, Böhme et al. introduce the propagation costs φ, which, through its recursive definition, comprise all contributions to further wait states in the execution.

$$\varphi(a_q^l) = \sum_{\substack{\zeta=\left((a_p^{i'},a_q^{k'}),(a_p^i,a_q^k)\right)_\zeta \\ a_q^{k'}<a_q^l<a_q^k}} \frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta}\omega(a_q^l)\left(\omega(a_p^i) + \varphi(a_p^i)\right) \tag{4.13}$$

As delays can contribute indirectly to both synchronization-based and contention-based wait states, the propagation costs need to reflect this appropriately. Therefore, the re-definition of the original propagation costs to *unified propagation costs*, including costs of both synchronization and contention, is deferred until after the introduction of the interval-delay costs.

Interval-delay costs

Owing to their definition—where an activity is not waiting for a remote activity to start but to end—contention-based wait states need to be handled differently from synchronization-based wait states. The definition creates an inherent inequality of the pre-contention intervals κ^+ and κ^-—the former containing more activities of the lock epoch than the latter. With Böhme's original heuristic assigning costs to per-call-path imbalance (delays) within the synchronization

interval, the cost distribution applied to pre-contention intervals would overemphasize those additional call paths as optimization targets. However, these are often the least viable activities to change, as they may only contain the necessary communication and synchronization for the data exchange and the user has little chance of changing them. As contention is resolved by controlled per-call-path load imbalance within the pre-contention interval, attributing costs of contention-based wait states as part of the delay-costs metric is counter-inductive. The unified cost model therefore introduces the additional metric *interval-delay costs*. Just as the delay costs indicate which call paths need to be balanced, the *interval-delay costs* indicate which call paths are candidates for introducing controlled imbalance. Where the original cost model only strives to achieve balanced execution times for all call paths in the synchronization interval, the unified cost model integrates the special handling for pre-contention intervals that moves the focus from a per-call-path balance to a more general per-interval balance. As such, the unified cost model distributes the costs for a contention-based wait state among all call paths of activities in the pre-contention interval κ^+ on the process causing the wait state relative to their duration in the interval. This heuristic emphasizes longer activities as optimization targets, as these may pose better opportunities for introducing a controlled per-call-path imbalance.

To facilitate a contention-free access, the interval κ^+ on the causing process q has to be shorter or equal to the interval κ^- on the waiting process p, as p only needs the requested resource to be available and does not require it to be released by q directly before p's acquisition. The length (or duration) of κ^+ is defined by two factors: (1) the sum of wait-state-adjusted execution times $\hat{d}_{\kappa^+}$ and (2) any wait states in the interval that skew the activities in a way that leads to the conflicting access of the two processes. If the sum of all wait-state-adjusted execution times in the pre-contention interval $\hat{d}_{\kappa^+}$ is smaller than the corresponding sum of wait-state-adjusted execution times $\hat{d}_{\kappa^-}$ on the waiting process, the load between the processes would already be appropriately imbalanced for contention-free access to the shared resource. In those cases, the contention is only caused by the presence of wait states in the interval. Consequently, the activities should not be considered as contributing to the contention and should not be part of the distribution of interval-delay costs. Wait states, in contrast, are always considered as causes in the cost distribution, as they constitute unwanted application inefficiencies. The interval-delay costs reflect this algorithmically, using separate scaling factors s_d and s_ω for the contributions of waiting-time-adjusted execution time and waiting time, respectively.

Definition 17 (Interval-delay-cost scaling factor) For a pre-contention interval $\kappa = \left((a_p^{i'}, a_q^{k'}), (a_p^i, a_q^k)\right)_\kappa$, let the sum of all waiting-time-adjusted execution times $\hat{d}_{\kappa^+}$ and the sum of all waiting times $\hat{\omega}_{\kappa^+}$, be defined as:

$$\hat{d}_{\kappa^+} = \sum_{c \in C} d_{\kappa^+}(c) \qquad \hat{\omega}_{\kappa^+} = \sum_{a_q^{k'} < a_q^l < a_q^k} \omega(a_q^l) \tag{4.14}$$

Furthermore, let the ratio of the sum of waiting time present in the pre-contention interval κ^+ to the waiting time caused on the waiting process ω_{κ^-} be defined by:

$$r = \begin{cases} \frac{\hat{\omega}_{\kappa^+}}{\omega_{\kappa^-}} & \text{if} \quad \hat{\omega}_{\kappa^+} \leq \omega_{\kappa^-} \\ 1 & \text{otherwise} \end{cases} \tag{4.15}$$

Then the scaling factor s_ω for any waiting time present in the pre-contention interval κ^+ is defined by

$$s_\omega = \frac{r}{\hat{\omega}_{\kappa^+}} \tag{4.16}$$

and the scaling factor s_d for any waiting-time-adjusted execution time $d_{\kappa^+}(c)$ in the pre-contention interval κ^+ is defined by

$$s_d = \frac{1-r}{\hat{d}_{\kappa^+}} \tag{4.17}$$

□

Using this scaling factor for each individual activity present in the pre-contention interval, the short-term interval-delay costs can be defined as follows:

Definition 18 (Short-term interval-delay costs) The total short-term contention costs of a call path c on process q is defined by the sum of all contributions to wait states $\omega(a_p^i)$ over all pre-contention intervals κ^+ where call path c is part of an interval $\kappa^+ > \kappa^-$.

$$\text{Short-term interval-delay costs}\,(q,c) := \sum_{\kappa=\left((a_p^{i'},a_q^{k'}),(a_p^i,a_q^k)\right)_\kappa} \frac{1-r}{\hat{d}_{\kappa^+}}\, d_{\kappa^+}(c)\, \omega(a_p^i) \tag{4.18}$$

□

Similarly, the long-term interval-delay costs can be defined using the corresponding value for activities in the pre-contention intervals. Analogous to the long-term delay costs, φ refers to the unified propagation costs.

Definition 19 (Long-term interval-delay costs)

$$\text{Long-term interval-delay costs}\,(q,c) := \sum_{\kappa^+=\left((a_p^{i'},a_q^{k'}),(a_p^i,a_q^k)\right)_\kappa} \frac{1-r}{\hat{d}_{\kappa^+}}\, d_{\kappa^+}(c)\, \varphi(a_p^i) \tag{4.19}$$

□

Unified propagation costs

It is in the nature of propagating wait states that they contribute to multiple other wait states later in the execution. Such indirect wait states can be synchronization-based or contention-based, as was already illustrated by Figure 4.2 on page 74. Böhmes's original definition of propagation costs only considers synchronization-based wait states and was fully defined in terms of delays within synchronization intervals. Contention-based wait states may also propagate, therefore the unified propagation costs extend Böhme's original definition to reflect this additional property. The extension can be incorporated into the definition of the propagation costs $\varphi(a_q^l)$ of a specific activity a_q^l using an additional operand. Analogous to Böhme's original definition of the propagation costs, the unified propagation costs of a specific call path are the sum of separate recursive accumulations of the propagation costs of subsequent synchronization-based wait states and the costs of subsequent contention-based wait states. Through this recursion, a propagating wait state's contribution to both synchronization-based and contention-based wait states is achieved.

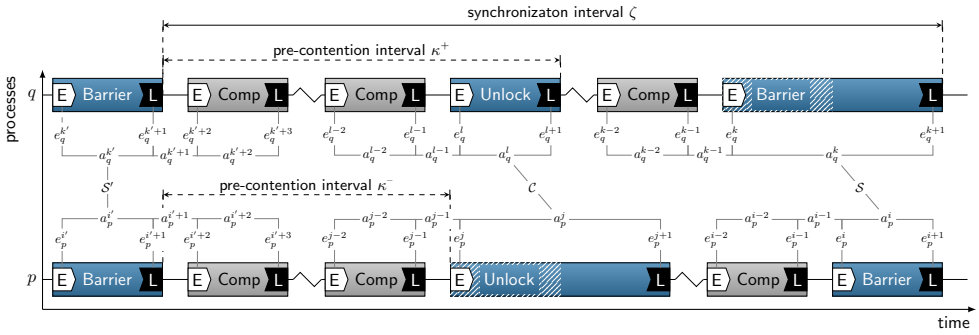


Figure 4.3.: Example timeline diagram, showcasing the symbols used to define the unified wait-state formation model.

As noted, the initial definition receives an extra operand summing up all contention-based contributions to the long-term costs, resulting in the following definition of $\varphi(a_q^l)$. Note that the propagating costs include the short-term and long-term costs of a wait state where it constitutes a propagating wait state.

Definition 20 (Propagation costs, based on [23])

$$\varphi_{\text{sync.}}(a_q^l) = \sum_{\substack{\zeta=\left((a_p^{i'},a_q^{k'}),(a_p^i,a_q^k)\right)_\zeta \\ a_q^{k'}<a_q^l<a_q^k}} \frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta}\omega(a_q^l)\left(\omega(a_p^i)+\varphi(a_p^i)\right) \tag{4.20}$$

$$\varphi_{\text{cont.}}(a_q^l) = \sum_{\substack{\kappa=\left((a_p^{i'},a_q^{k'}),(a_p^i,a_q^k)\right)_\kappa \\ a_q^{k'}<a_q^l<a_q^k}} \frac{r}{\hat{\omega}_{\kappa^+}}\omega(a_q^l)\left(\omega(a_p^j)+\varphi(a_p^j)\right) \tag{4.21}$$

$$\varphi(a_q^l) = \varphi_{\text{sync.}}(a_q^l) + \varphi_{\text{cont.}}(a_q^l) \tag{4.22}$$

□

To aid the understanding of the symbols used in the formulae and their relationships in the unified wait-state formation model, Figure 4.3 shows a timeline diagram with the basic entities labeled as they are used throughout this section. The first barrier (on the left) marks the synchronization point $\mathcal{S}'$ at the beginning of the synchronization interval ζ, which connects the activities $a_p^{i'}$ and $a_q^{k'}$. The second barrier (on the right) marks the synchronization point $\mathcal{S}$ at the end of the synchronization interval ζ. It connects the two activities a_p^i and a_q^k. Within this synchronization interval, the two processes p and q access the same resource concurrently. The scenario shown uses exclusive locks, leading to a contention point $\mathcal{C}$ at the unlock activities a_p^j and a_q^l. The pre-contention intervals κ^+ and κ^- span between the synchronization point $\mathcal{S}'$ and contention point $\mathcal{C}$ on processes q and p, respectively. Note that κ^+ also contains the activity a_q^l that is causing the contention. Furthermore, Figure 4.11 on page 94, demonstrating the imple-

mentation of the overall cost accounting, provides an additional example scenario with including synchronization-based and contention-based wait states.

4.2. The critical path in one-sided communication

Originally developed by Kelly et al. in 1959 as a project planning tool in engineering [77], to coordinate concurrent activities in complex projects, critical-path analysis was soon adapted to other scheduling scenarios, such as the scheduling of tasks in a distributed system. The general idea of this approach as adapted by computer science is to model the execution of a parallel application as a so-called *program activity graph* (PAG). Such a graph is a directed, cycle-free graph that describes the activities of a (parallel) application and their relationships. The vertices of the program activity graph are the activities. Subsequent activities on the same process are connected through *sequence edges*; process interaction is indicated by *communication edges*. Sequence edges point from the temporal predecessor to the successor; communication edges from the sender to the receiver. Figure 4.4 shows an example program activity graph. To highlight the difference between sequence and communication edges, the former are drawn as solid arrows and the latter are drawn as dashed arrows.

Definition 21 (Critical path [23]) The critical path is the longest path through the program activity graph of an application. The activities part of the critical path are called *critical activities*. □

The critical-path analysis does not model specific data or execution dependencies between process-local activities but models the measured sequence of execution as the only local relationship between activities. Process-local activities are therefore represented as a single sequence connected by sequence edges only. Communication edges connect distinct sequences of activities. While for a serial application in this model every activity contributes to the critical path and is therefore considered a critical activity. In a parallel application, the communication between activities introduce dependencies where one activity can only complete with the start of another an the potential for wait state arises. To enable the critical-path analysis, Böhme et al. use the communication events available in the trace to model communication edges, while the enter and leave events present in the trace are used to model the sequence of activities on each process. The process-local set of activities $A_p = (a_p^0, \ldots, a_p^n)$ on a process p (see Def. 2 on page 45) represents the activities connected by sequence edges. As Böhme et al. point out, the activities on the critical path do not contain waiting time, as waiting implies that a set of remote activities is taking longer than the local activities up to the synchronization point. Shortening a non-critical activity has no influence on the overall execution time, as it only increases waiting time at a subsequent synchronization point. Shortening a critical activity, however, reduces the overall execution time up to a synchronization point and therefore reduces the waiting time of dependent processes.

Based on the infrastructure provided by Scalasca's event-based trace analysis, Böhme et al. implemented the replay-based critical-path detection for parallel applications [24]. Synchronization and pre-contention intervals represent sub-graphs in the program activity graph where the sequence of activities on one process is the longest of all processes in a specific interval. For

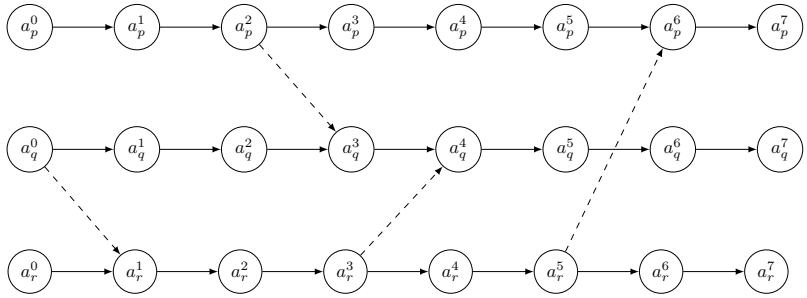


Figure 4.4.: Example program activity graph. Vertices are activities. Solid arrows depict sequence edges between activities on the same process and dashed arrows depict communication edges between activities on distinct processes.

each synchronization interval, the process arriving last at the synchronization point determines the time taken for that synchronization interval for all participating processes. In a parallel application, however, many concurrent and partially overlapping synchronization intervals among processes exist. A sequence of activities of one process causing a wait state on another process at the end of a synchronization interval may also be part of an overlapping synchronization interval with a third process where the first process is waiting. By definition, the activities leading up to a wait state cannot be part of the critical path. Thus, not all synchronization intervals have activities on the critical path, and it is the task of the analysis to identify those that do.

The Scalasca parallel performance analyzer creates an analysis report that contains data summarized over the runtime of the application. To present the critical path information in a summarized form next to other performance metrics in the analysis report, Böhme et al. define two metrics related to the critical path: (1) the *critical-path profile* and (2) the *critical-path imbalance indicator* [24].

Definition 22 (Critical-path profile [23]) The critical-path profile is a function that maps a process $p \in P$ and call-path $c \in C$ onto a positive real number. It represents the time spent in critical activities by each process broken down into individual call-path contributions.

$$\text{Critical-path profile}(p, c) : P \times C \to \mathbb{R}^{\geq 0} \qquad \square$$

The critical-path profile indicates the call-path and location of the critical activities. High values on individual processes do not directly indicate strong imbalance problems; they have to be seen in the context of the critical-path imbalance indicator.

Definition 23 (Critical-path imbalance indicator [23]) The critical-path imbalance indicator ι is the difference between the time of a call path c spent on the critical path $d_{crit}(c)$ and the average time spent in that call path over all processes. As an imbalance only affects the overall runtime if the time on the critical path is larger than the average time across all

processes, Böhme et al. only include positive values.

$$\iota(c) = \max\left(d_{crit}(c) - \text{avg}(c), 0\right)$$
$$\text{avg}(c) = \frac{1}{P}\sum_{p=1}^{P} d_p(c)$$

□

To enable critical-path detection for one-sided communication, only two steps are necessary: (1) the identification of wait states in one-sided communication or synchronization constructs and (2) the specification of the critical-path ownership transfer from one process to another.

How to detect wait states in one-sided communication was already covered in Chapter 3, leaving only the specification of the correct transfer of the critical path open at this point. Directly resulting from Definition 21, a sequence of critical activities never contains waiting time. In accordance with its initial definition by Böhme et al., the transition points for the critical path in one-sided communication scenarios must therefore be at synchronization and contention points in the application. For the synchronization-based wait states in one-sided communication, the integration with Böhme et al.'s initial implementation is straight-forward [5]. For contention-based wait states, Definitions 8 and 9 need to be taken into account to identify additional transition points for the critical path. In conjunction with the specific description of the replay-based implementation of the critical path detection, Section 4.3.2 will feature a more detailed discussion of each individual wait-state scenario possible in one-sided communication and how it is handled in a specific example case starting on page 88. A detailed visual representation of how the critical path is computed is shown in Figures 4.7 to 4.9 on pages 89 and 91. From the formal perspective, once all synchronization and contention points are identified, the computation of the critical-path profile and imbalance falls into place. Further performance indicators based on the critical path defined by Böhme, such as the *performance-impact indicators* [23], can be computed similarly.

4.3. A scalable analysis framework

As introduced in Chapter 2, a *parallel replay*—in Scalasca terminology—describes the concurrent, consecutive processing of a series of events in local event streams, where the processing of individual events can trigger specific actions, including inter-process communication. Scalasca uses the recorded communication information—represented as specific events in the event streams—to pass information among processes in the direction of the replay. Initially, Scalasca used only the *chronological* or *forward replay* for its parallel wait-state search. To enable information to be passed in *reverse-chronological* order Becker et al. introduced the *backward replay* [13]. Böhme et al. used this infrastructure in the implementation of both the root-cause and critical-path detection [26].

Figure 4.5 shows how the two types of replays are orchestrated in Scalasca to enable the computation of higher-level performance metrics beyond simple wait states, specifically the critical-path

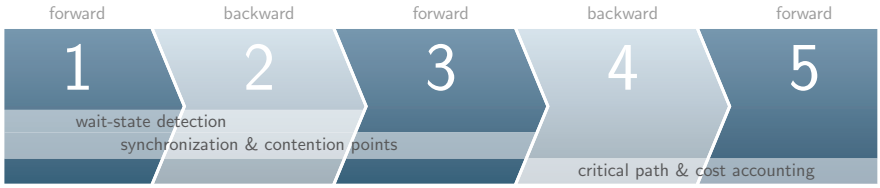


Figure 4.5.: Five consecutive replay phases used in Scalasca. Each phase can use data structures built up in prior phases. The replay direction is indicated by the heading and color of the individual phase. Wait states are detected in the first two phases. Contention point data is computed in the first phase, while synchronization point data is computed in phases one through three. Once synchronization and contention point data is computed, phases four and five assess the overall costs and the critical path.

and cost metrics. Each replay phase can build on the information gathered and computed in earlier phases. The first phase identifies most of the wait states supported by the Scalasca analyzer. Depending on the algorithm detecting a wait state, synchronization and contention point information can already be computed directly for all processes involved. For one-sided communication scenarios, the detection of progress-related wait states and lock contention already exchanges the full synchronization information during the first phase. Also synchronization points in collective functions can be determined fully during waiting-time detection in the first phase. For the other wait-state patterns, synchronization point information is still incomplete. Each process with a wait state knows about its synchronization point with the remote process, however, the causing process does not know directly that it was involved in a wait state. The second replay phase lets the synchronization information flow back to the causing processes.

One pattern, the *Late Receiver* wait state of point-to-point communication, occurs at the sender—the source of information in a forward replay. To detect it, the timing information must flow into the opposite direction—the sender needs information from the receiver. By using the backward replay, the information flow for waiting time detection is natural in the direction of replay. There, the original receiver can send data to the original sender and enable the computation of potential waiting time. Just like for the wait states identified in the first replay, the synchronization partner causing the *Late Receiver* wait state needs to be notified of its involvement in the wait state during the subsequent replay phase. Thus, the third replay phase—a forward replay—is needed to conclude the identification of the synchronization points. After the third replay phase, all synchronization and contention points are known to all the processes involved. Now, the critical path detection and cost accounting for root causes of wait states can be performed in the last two phases. The remainder of this section will discuss the communication needed in each replay phase to perform the outlined tasks.

4.3.1. Detection of synchronization and contention points

One of the key ideas of the backward replay is the inversion of the communication roles. If information needs to flow from the receiver to the sender, their communication roles are inverted—a sender posts a receive while the initial receiver sends the information. Becker et al. show how

this can easily be achieved for point-to-point and collective communication, enabling efficient and deadlock-free data flow in reverse-chronological order along the recorded communication paths [13]. However, the inversion builds on the fact that communication information is present on both sides of any point-to-point or collective communication. One-sided communication is lacking a matching function call on the target to allow an easy inversion of the communication paths. For one-sided communication, the existing approach therefore needs to be adapted.

One of the characteristics of one-sided communication is the separation of data transfer and its corresponding memory and process synchronization. While the data transfer itself often does not create synchronization points, its synchronization may do so. The Message Passing Interface defines three synchronization schemes for one-sided communication [106], each with the potential to block a local process until a certain action on a corresponding remote process is taken, as discussed in Chapter 3. The synchronization schemes can be classified as either active or passive target. In active target synchronization, the target process does not have to issue matching communication, but matching synchronization functions. For the collective version of active target synchronization, the wait-state detection algorithm already exchanges all necessary synchronization point information, thus no additional exchange is needed. This is the case for the *Wait at Fence* family of wait states described in Section 3.2.2 on page 47. However, the information about synchronization points in general active-target synchronization—namely *Late Post* and *Early Wait* wait states—still needs to be communicated after the wait state's detection. Similar to the *Late Sender* wait state pattern in point-to-point communication, information to complete the synchronization point detection needs to be passed in the opposite direction of the initial synchronization information. The information is therefore communicated during the second replay phase. In contrast to point-to-point communication, however, the groupwise synchronization of general active-target synchronization poses additional challenges owing to (1) the requirements for a correct nesting of access and exposure epochs and (2) dynamic group sizes for each epoch.

General active-target synchronization separates access and exposure epochs. Each is opened and closed by a distinct set of calls on both origin and target of the remote memory operations. The synchronization is group based; a target process needs to specify all origin processes it exposes memory to when opening an exposure epoch, while an origin process needs to specify all target processes it plans to access. It is erroneous for a target process to specify an origin process that does not specify the same target process in its provided group. When starting an exposure and an access epoch on the same process, the blocking semantics of the synchronization calls require a strict order: the calls for the exposure epoch need to embrace the calls handling the access epoch on the same window. The reason for this is simple. To ensure consistency, a target process can open an exposure epoch purely based on local information—it does not need to wait for any other process. It will therefore not wait for any origin processes to send an acknowledgement for starting a corresponding access epoch. An origin process, however, can only start the real access, after the respective target process has opened the exposure epoch, which means it will have to wait for an acknowledgement of the corresponding target that the exposure epoch has started.

If a process is both origin and target in the same epoch, it therefore has to first perform the call with no external dependencies (post) and then the one with potential external dependencies

(start). In such a scenario, naïvely inverting the communication direction using general active-target synchronization for a backward replay would have the nesting of the necessary calls inverted, potentially leading to a deadlock. Guaranteeing deadlock-freeness is not impossible, however, yet it adds an additional level of complexity to ensure proper function in all possible scenarios.

Furthermore, to ensure that all origin processes know about the location of valid remote memory blocks on the target processes, one-sided communication in general, and MPI one-sided communication in particular, often requires collective allocation of buffer space. As the data that needs to be transferred during backward replay can be dynamic in size between access epochs, and general active-target synchronization is group-based and not collective, ad-hoc allocation of additional buffer space is not possible. Using a worst-case heuristic, allocating distinct memory locations for all processes would be needed but may be prohibitive at large scales.

As an alternative, point-to-point communication provides (1) loose synchronization using its non-blocking interface and (2) ad-hoc allocation of buffer space. However, it requires both the sender and the receiver of information to actively take part in the communication. As the processes receiving the synchronization information do not know yet whether a synchronization point exists—it is the purpose of this communication to distribute this information—the data exchange needs to include all potential synchronization partners. Compared to the normal point-to-point communication, where only two processes interact, general active target synchronization has an $n : m$ relationship. As a consequence, all processes in the access groups need to communicate with all corresponding processes in the exposure groups and vice versa. In a worst case scenario, when the program uses general active-target synchronization to synchronize a window over almost all processes in an all-to-all-like communication pattern, every process would have to send $p - 1$ messages. However, in practice, the collective fence synchronization is much better suited for all-to-all like synchronization and general active-target synchronization is used for exchanges with pre-known and fairly static neighborhood topologies. In such scenarios, the communication and memory requirements per process can be regarded as constant, as the size of the neighborhood is independent of the overall number of processes.

Considering the advantages and disadvantages of the two communication approaches for the backward replay discussed, the use of point-to-point communication wins over one-sided communication owing to its simplicity and ease of implementation. Figure 4.6 illustrates the message exchanges needed to complete the synchronization information in *Early Wait* and *Late Post* wait-state patterns. In the example shown, process D and B as well as C and A share a synchronization point. Process B causes an *Early Wait* wait state, as it is the last process to complete its access epoch and the completion is started after process D starts to close its corresponding exposure epoch. As the origin processes do not know whether they caused a wait state, each target sends a message to every origin process. The message to the causing process contains the actual waiting time ω_w, all others zero. For the *Late Post* wait state the message transfer is similar. Here, each origin process sends a message to every target. Process A, which is regarded as causing the wait state on process C receives a message with the detected waiting time ω_s. Process B receives a message indicating zero waiting time and hence does not create a synchronization point.

To handle wait state patterns of passive target synchronization—*Wait for Progress* and *Lock Contention*—more changes to the original implementation are necessary. As an exception among

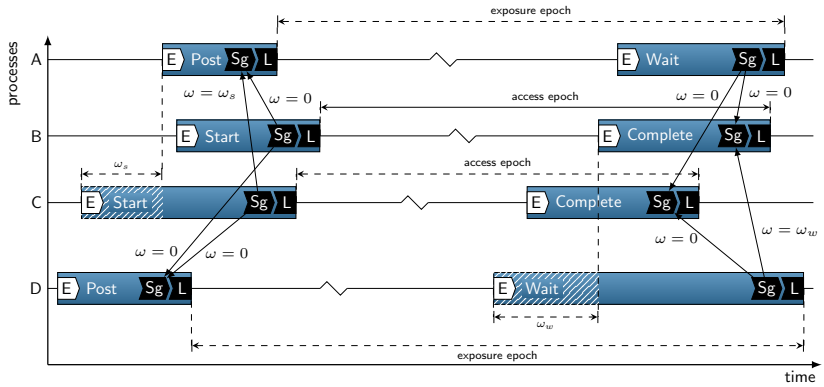


Figure 4.6.: Communication during the synchronization point detection for the *Early Wait* and *Late Post* wait-state patterns. In the backwards replay, the target (and potentially waiting) process sends messages to all origins. Only the origin process causing the wait state receives the actual waiting time, all others zero.

the synchronization-based wait states, the *Wait for Progress* pattern does not contain full synchronization information at the events of the corresponding synchronization points. While the waiting process can use the communication information stored in the local event of the origin—comprising information about the communication context and the target—the corresponding synchronization point on the target does not contain such information. On the target, a call potentially providing progress may not even be a communication or synchronization call at all. Therefore, all the necessary information is associated with the enter event of the respective activity on the target. Moreover, the origin process includes all necessary communication parameters with its request and the target stores additional information separately. This communication information is then used in subsequent replays to exchange the necessary information with respect to that synchronization point. Furthermore, a single function on the target may provide progress to multiple RMA operations from distinct origins, resulting in multiple distinct synchronization points associated with the same event.

Currently, the only contention-based wait state pattern detected by Scalasca is *Lock Contention*. Its detection and quantification, as it is described in Section 3.2.3, is also done in the first analysis phase. What makes resource contention stand out from the other wait state patterns is that the waiting processes have no explicit connection in the event stream. That is, they have no explicit event information that tells them which other process is waiting for them or which process they are waiting for; they only know the resource they want to access. After the detection of a lock contention wait state, as described in Section 3.3.2, the origin processes know which process they cause to wait and which they are waiting for. During the fourth replay phase, each process waiting due to lock contention determines the waiting-time-adjusted execution time of all call paths in the pre-contention interval and sends it to the causing process. The causing process receives this information and compares it to its corresponding pre-contention interval. It then determines whether its local activities get partial costs attributed of the waiting time caused. If

its own waiting-time-adjusted execution times—including the additional time needed to release the lock—is less or equal to the corresponding times sent by the waiting process, the costs are distributed among the wait states of the causing process in the pre-contention interval as described by eqs. (4.18) and (4.19) on page 79.

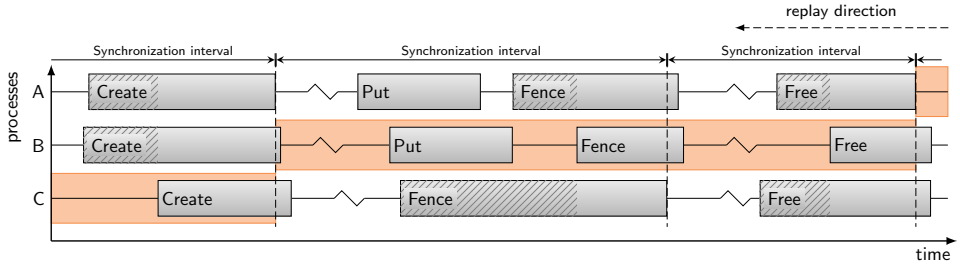
4.3.2. Computing the critical path

The critical path constitutes the stream of activities—the critical activities—that determine the length of a parallel execution. The activities in a synchronization interval on the process causing the wait state at the interval's end also determines the length of the interval. However, not all of those activities are also critical activities. They are only candidates for critical activities, as explained in Section 4.2. It is therefore the task of the critical-path analysis to identify those synchronization intervals that actually contain critical activities.

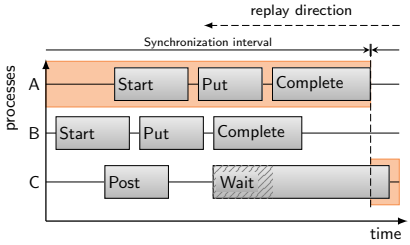
MPI, SHMEM, and ARMCI, as common HPC communication libraries, need to be initialized before their first use and shutdown explicitly before program termination to ensure system resources are handled properly. For the critical path, the explicit shutdown before program termination is an important factor. Such a function is commonly collective and synchronizes all processes. This means a process can only terminate if all processes reached this point of explicit shutdown. It therefore constitutes the end of the last synchronization interval of the parallel execution. Clearly, any process arriving last at this point dictates when the overall program exits at the earliest. For MPI, this routine is `MPI_Finalize`. The critical-path analysis, as introduced by Böhme et al., therefore starts with identifying the process causing wait states during finalization for all other processes, which resembles the last synchroniztion point of the execution. The activities on that process prior to this last synchronization point are therefore critical activities. As the critical path does not contain any waiting time, any activity containing waiting time cannot be a critical activity. As a consequence, the last waiting activity on the process causing the last wait state in the programm cannot be part of the critical path. This activity—more precisely its successor—therefore marks the beginning of the last interval of critical activities—the last segment of the critical path. If a process' activities directly after a wait state are critical activities, the activities prior to the corresponding synchronization or contention point on the process causing the wait state are also critical activities; their length directly dictates when the critical activities on the waiting process can start and therefore influences the length of the overall execution.

Using this characteristic of the parallel execution, the critical activities can be identified moving in reverse-chronological (backward) direction through the trace. A flag indicating the current ownership of the critical path is passed between processes at synchronization points. The analysis starts with critical-path ownership assigned to the process causing the wait states during finalization at end of the last synchronization interval of the processes. Processing the trace from the end to the beginning, the ownership of the critical path changes from the process owning the critical path at any wait state to the process causing it. Using this approach, only a single process owns the critical path at any given time of the analysis.

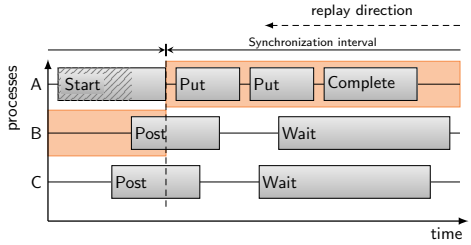
For every synchronization and contention point, the ownership information is exchanged from the waiting process to the one causing the wait state. If the waiting process is not owning the



(a) Wait at Create/Fence/Free



(b) Early Wait



(c) Late Post

Figure 4.7.: Computation of the critical path in active-target one-sided communication. As the detection is done during backward replay, the change of ownership has to be tracked from right to left. Wait states are indicated by hatched areas. The ownership of the critical path for a time interval is indicated by an orange rectangle. The ownership may change at events in the respective regions identified as synchronization or contention points.

critical path at that time, it cannot pass it to the causing process; it therefore passes a zero value to the causing process, indicating that the causing process is not taking over the ownership of the critical path. This way, the ownership of the critical path passes among the processes, finally arriving at the beginning of the execution.

Figures 4.7 to 4.9 show the different types of wait states in one-sided communication and how the critical path changes ownership in such scenarios. In the individual timeline figures, the critical path is indicated by the highlighting orange rectangle potentially spanning multiple activities on individual processes. Note that the direction of the timelines is in chronological order from left to right, the computation of the critical path, however, is from right to left, as the analysis is performed in reverse-chronological order. As the critical activities are determined during a backward replay, the discussion of the critical-path ownership also follows the replay direction—from right to left. For the purpose of these examples, one process is assumed to own the critical path initially.

Figure 4.7a shows the critical path behavior for the collective wait-state patterns of the *Wait at Fence* family. In the beginning of the analysis, the scenario depicted assumes process A to own the critical path, as shown on the right edge of the timeline. In the collective free

function it waits for process B, which enters the collective call late, causing wait states on processes A and C. Therefore, process A transfers the ownership of the critical path to process B. On the wait state at an intermediate fence call, process B is the culprit, yet, it already owns the critical path, thus the critical path does not change ownership. In the third wait state scenario in the figure, this time at the collective creation of the window handle, process C is the process causing wait states on all other processes including process B, which is on the critical path at that time. Therefore the critical path turns to process C. As these wait-state patterns form at collective synchronization calls, the replay algorithm can use collective communication to determine the ownership of the critical path. Böhme's original implementation for similar wait states at collective N×N operations use the communicator associated with the collective operation. For one-sided communication, the communicator needs to be determined through the associated window of the synchronization operation. Apart from this additional indirection, the computation of the critical-path ownership is the same compared to the original collective detection algorithm as introduced by Böhme et al. and explained below. All processes perform a reduce operation with the process causing the wait state as the root. Each process provides either the value 0, if it is not on the critical path at the time, or the value 1, if it is. The collective operation uses the `MPI_MAX` operator to check whether any participating process is in fact on the critical path. If the result of the reduction is 1, then one of the waiting processes owned the critical path, and the causing process has to take over ownership. If the result of the reduction is 0, none of the waiting processes owned the critical path, therefore the causing process cannot take over any ownership.

Figure 4.7b shows how the critical path changes in case of an *Early Wait* wait state. As explained in Chapter 3, in general active-target synchronization scenarios the target process has to wait for every origin process to close its access epoch before it can close its corresponding exposure epoch. In this example, the target (process C) owns the critical path prior to the evaluation of this condition. Process A is the last process to complete the access epoch causing process C to wait. The data structures storing the synchronization point information for the corresponding events already contain mutual information—i.e., both processes of the synchronization point know which process to send the information to and which to receive it from, respectively. Process C therefore sends the critical path flag indicating 1 to process A, which then takes over the ownership of the critical path until its next wait state in replay direction.

Figure 4.7c shows the transition of the critical-path ownership in the case of a *Late Post* wait state. Here, origin processes wait for the target process to start the exposure epoch. In the specific scenario shown in the figure, the start of the access origin blocks until the last target process (B) opens its pending exposure epoch. Therefore, assuming process A is on the critical path during its access epoch, it changes to process B at the mutual synchronization point.

Figure 4.8 shows a communication scenario with a *Wait for Progress* wait state. The `AllFence` function call on process C has to wait for remote progress on processes A and B, with B being the last process to provide remote progress. It is therefore identified as the process causing the wait state on process C. During its detection in the first replay phase, both processes already obtain all necessary information to communicate the critical-path ownership. Similar to the scenarios of the general active-target synchronization, the waiting process C sends the critical-path flag set to 1 to process B. Upon receiving the flag, process B takes over the ownership of the critical path.

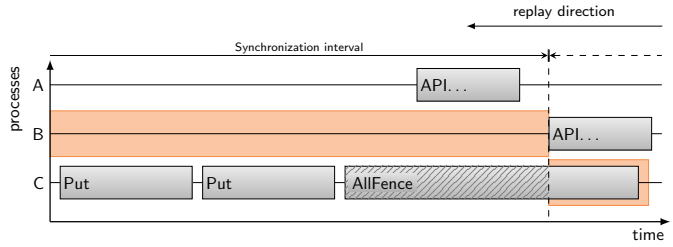


Figure 4.8.: Computation of the critical path in *Wait for Progress* scenarios. As the detection is done during backward replay, the change of ownership has to be tracked from right to left. Wait states are indicated by hatched areas. The ownership of the critical path for a time interval is indicated by an orange rectangle. The ownership may change at events in the respective regions identified as synchronization or contention points.

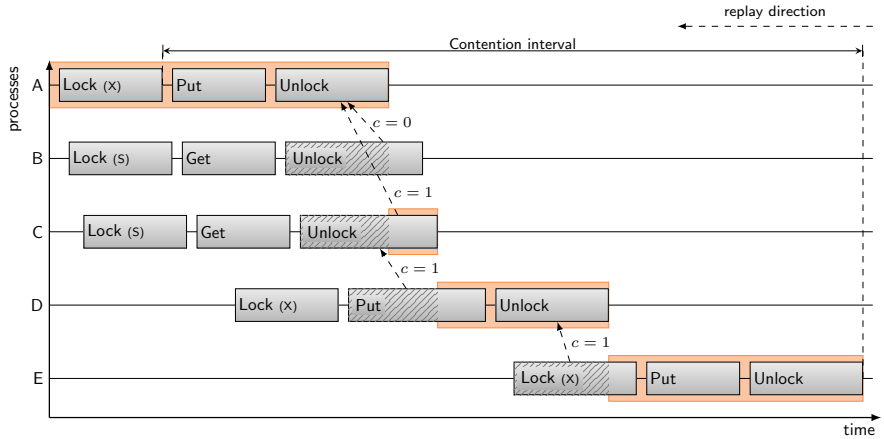


Figure 4.9.: Computation of the critical path in *Lock Contention* scenarios. As it is done during backward replay, the change of ownership has to be tracked from right to left. Wait states are indicated by hatched areas. The ownership of the critical path for a time interval is indicated by an orange rectangle. The ownership may change at events in the respective regions identified as synchronization or contention points.

Figure 4.9 shows the critical-path transition for contention-based wait states. Although the root causes for contention-based wait states are determined differently in comparison to synchronization-based wait states, the transition of the critical path is computed similarly. The scenario shown is the same as the initial contention scenario discussed in Section 3.2.3. Processes B and C use a shared lock (S), whereas all other processes use exclusive locks (X). Assuming process E owns the critical path when the replay encounters this locking scenario, all activities of its lock epoch up to the activity that experienced the wait state are still identified as critical activities.

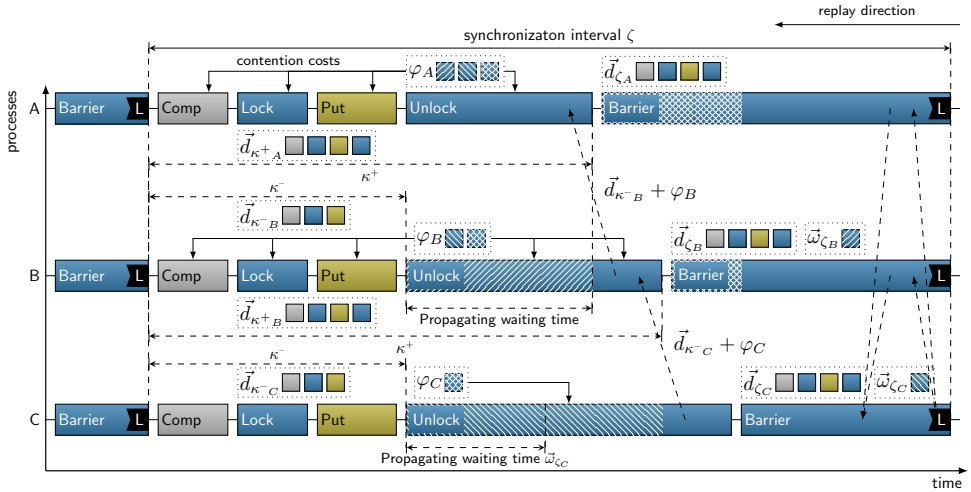


Figure 4.10.: Computation of delay and interval-delay costs with contributions to both synchronization-based and contention-based wait states. Dotted rectangles represent vectors of data; squares represent a contribution of a specific activity to that data. Dashed arrows indicate data exchange between the processes. Solid arrows connected to the propagation cost vectors *PropCostSym* indicated which activities are considered when computing the individual contribution.

Process E blocked in the `Lock` call—one of three different possible implementation scenarios. Using the information exchanged during the wait-state detection, process E sends the critical-path flag set to 1 to process D. Process D uses a second possible implementation scenario for lock acquisition in MPI one-sided communication, it blocks during the RMA operation scheduled for the target window. Again, all activities in backward direction from obtaining the critical path up to the contended activity containing the wait state (indicated by hatched areas) are identified as critical activities. Process D then sends the critical-path flag set to 1 to process C. Processes B and C use a shared lock on the target window, so they can access it concurrently once process A releases the lock. Consequently, both processes send the value of their critical-path flag to process A. Process C, owning the critical path at that time sends a value of 1, whereas process B sends a value of 0. After evaluating the messages from both processes B and C, process A knows that it received ownership of the critical path.

4.3.3. Computing costs in the unified wait-state formation model

Figure 4.10 exemplifies the data flow needed to compute delay and interval-delay costs for call paths on specific processes. In the scenario shown, a lock-contention scenario is embraced by two barrier calls. Note that delay and interval-delay costs are computed in a backward replay, thus the diagram is read from right to left, including the message exchange indicated by dashed arrows. Dotted rectangles denote process-local data structures that correspond to vectors of

per-call-path contributions to time spent in the interval, as in the case of the waiting-time-adjusted execution time $\vec{d}_{\kappa}$; or scalar values, as in the case of the propagation costs φ. The boxes within those rectangles denote the activities that contribute to the values stored in those data structures. The solid arrows emerging from the propagation-cost rectangles of φ_A,φ_B, and φ_C indicate which wait states contribute to the propagation costs. In the scenario shown, the analysis starts and ends at barriers—synchronization points for all processes. Waiting time in the barrier function, denoted by the cross-hatched areas, is stored with the synchronization point information. Also, all processes involved know the process directly responsible for this wait state. Böhme's cost calculation for collective functions operates slightly different from the point-to-point algorithm in that it uses multiple collective operations to compute delays. The causing process C is the root of those collective operations and broadcasts its vector of activities $\vec{d}_{\zeta_C}$ and the vector of waiting times $\vec{\omega}_{\zeta_C}$ in the synchronization interval ζ to all participating processes, here processes A and B. The waiting processes A and B then compare the vector to their own vector of activities, $\vec{d}_{\zeta_A}$ and $\vec{d}_{\zeta_B}$ respectively. Doing so, each process identifies individual contributions and the respective scaling factor and delay. In this specific scenario, no delay can be identified; thus the waiting time $\vec{\omega}_{\zeta_C}$ on process C is fully responsible for the waiting time in the barrier of processes A and B. The computed delay from each waiting process is accumulated using a collective reduction with the causing process being the root again.

Process C suffers from a contention-based wait state with the waiting time $\vec{\omega}_{\zeta_C}$. To enable a proper cost distribution on the causing process B, process C computes and sends the vector of waiting-time-adjusted execution times in the pre-contention interval, $\vec{d}_{\kappa^-{}_C}$, along with the total costs φ_C accumulated so far to process B. Process B then compares the received vector of times in process C's pre-contention interval $\vec{d}_{\kappa^-{}_C}$ with its own time vector $\vec{d}_{\kappa^+{}_B}$. In this scenario, the waiting-time-adjusted interval $\vec{d}_{\kappa^+{}_B}$ is longer than the corresponding interval $\vec{d}_{\kappa^-{}_C}$ on process C. The activities in the different call paths in the pre-contention interval of process B are therefore partially causing the wait states observed so far. To compute the correct ratio, first, the excess time spent in $\vec{d}_{\kappa^+{}_B}$ in comparison to $\vec{d}_{\kappa^-{}_C}$ is computed. This excess time reflects the overall imbalance in the absence of waiting time, the pre-contention intervals on processes B and C contain. Owing to additional wait states on process B in the interval, the wait state on process C is larger than the computed excess time. Process B therefore computes the ratio of local contribution (excess time) to remote contribution (propagating wait states) for the direct wait state on process C. The resulting ratio is the factor used to compute the individual contributions of costs for the local activities in the pre-contention interval of process B. The rest of the propagation costs, along with process B's pre-contention interval vector $\vec{d}_{\kappa^-{}_B}$, is sent on to process A, higher in the contention chain.

The costs for process A are computed similarly. Process A receives the waiting-time-adjusted execution time $\vec{d}_{\kappa^-{}_B}$ and propagation costs φ_B from process B. It then compares its own activities in the pre-contention interval with the data received. Process A does not suffer waiting time in this interval, thus none of the costs propagate further and they are completely attributed to the corresponding call paths of the activities in the pre-contention interval according to the activities' individual contribution to the overall time spent in the interval.

Figure 4.11 exemplifies the communication and computation of delay and interval-delay costs in a scenario with synchronization-based and contention-based wait states. Arrows indicate messages sent between processes. Rectangles with dotted borders indicate the payload of these messages.

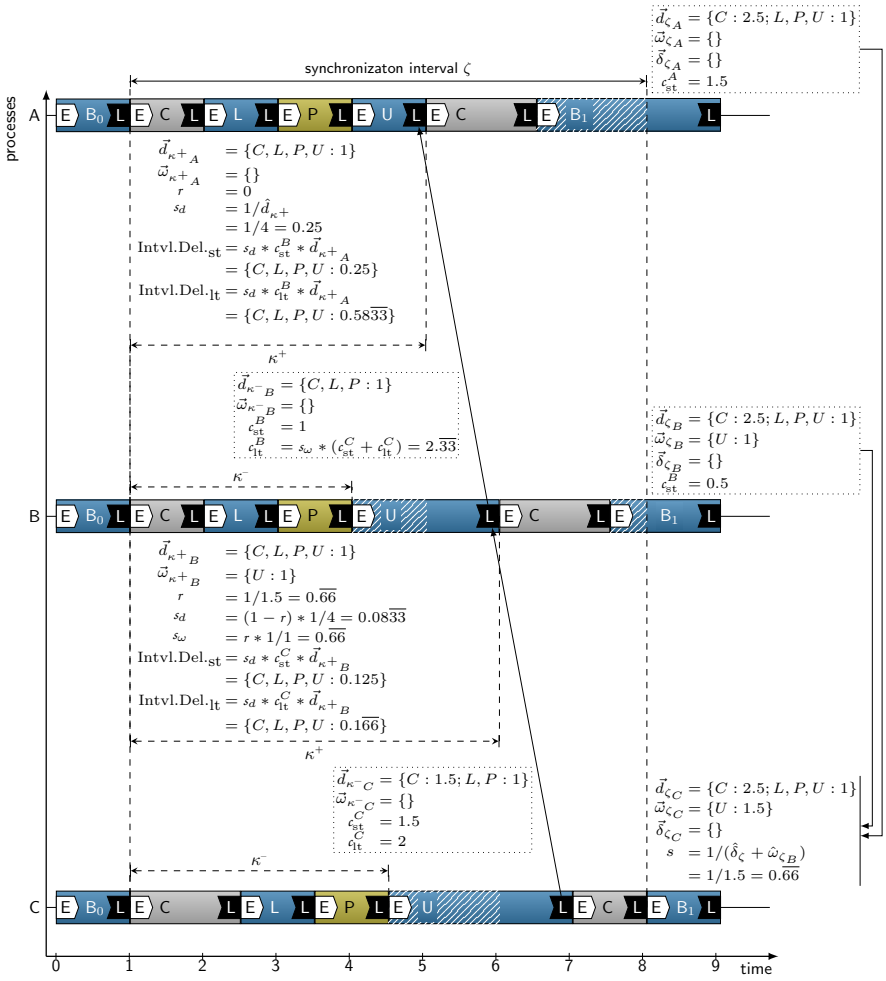


Figure 4.11.: Example computation of contention and delay costs with contributions to both synchronization-based and contention-based wait states. Arrows indicate messages sent between processes. Rectangles with dotted borders indicate the payload of these messages. Values without specific borders are computed using both local and received values. The x-axis uses an artificial time unit to better illustrate the specific costs attributed to the individual call paths.

Values without specific borders are computed using both local and received values. The x-axis uses an artificial time unit to better illustrate the specific costs attributed to the individual call paths. The scenario shows lock contention within a larger synchronization interval between two barriers. As the cost is computed via backward replay, the barrier (B_1) is evaluated first. Process C arrives late, causing a synchronization-based *Wait at Barrier* wait state. Each process determines the synchronization interval ζ consulting its local event stream and information collected during the previous replay phases. In this scenario, the synchronization interval spans to the barrier B_0 for all processes. To retain clarity in the figure, the communication shown for the analysis of the barrier wait state is simplified from the actual implementation that uses a combination of different collective communication calls to compute the delay $\vec{\delta}_{\zeta_C}$ present in the synchronization interval as well as the total amount of waiting time at the barrier B_1. The overall load in the synchronization interval is balanced, i.e., process C does not contain a delay in its activities of the synchronization interval, leaving $\vec{\delta}_{\zeta_C}$ empty. It does, however, contain a wait state in the unlock function U of process C. As this wait state is the sole cause of processes A and B waiting for C at the subsequent barrier, it is attributed with the full long-term costs c_{lt}^{C}, which is the sum of all waiting time at the barrier in this scenario. The wait state itself, with a length of 1.5 corresponds to the short-term costs c_{st}^{C} of this wait state. Short-term and long-term costs are sent to process B along with the vector of waiting-time-adjusted execution times $\vec{d}_{\kappa^- C}$ and the vector of waiting times $\vec{\omega}_{\kappa^- C}$ present in the pre-contention interval κ^-. Process B then evaluates the pre-contention interval of process C, to determine its own contributions. While process C waits for 1.5 units, the sum of wait states in its pre-contention interval κ^+ is 1. The remaining 0.5 units result from the larger sum of waiting-time-adjusted execution time of the activities on B taking longer than the ones on C. The ratio r of wait states in the pre-contention interval on process B to the wait state caused on process C is 1 to 1.5. Consequently, only a third of the overall short-term and long-term costs are attributed to activities on process B using the corresponding scaling factor s_d of $\frac{1}{12}$ ($0.08\overline{33}$). Using this scaling factor the 1.5 units of short-term costs c_{st}^{C} are distributed to the interval-delay costs of process B according to their contribution to the pre-contention interval, each with 0.125 units. The same scaling factor is also used for the 2 units of long-term costs c_{lt}^{C}, each with $0.1\overline{66}$ units. The contribution of the wait state in function U is computed using the waiting-time scaling factor s_ω. It is applied to the sum of short-term and long-term costs c_{st}^{C} and c_{lt}^{C}, respectively, forming the long-term costs c_{lt}^{B}. This is sent to the causing process of the wait state in U, process A, amounting to $2.\overline{33}$ time units. The short-term costs c_{st}^{B} sent along to process A are equal to the wait state caused, in this case 1 unit. Process A does not experience any wait state in its pre-contention interval, the causes for the remaining waiting time lies within the pre-contention interval on process A. Hence, the waiting time ratio r is zero, as is the waiting time scaling factor. The scaling factor for the waiting-time-adjusted execution time s_d is $\frac{1}{4}$ (0.25). Using this factor the interval-delay costs are computed for both short term and long term, being 0.25 and $0.58\overline{33}$ for each call path in the interval, respectively.

Summary

Böhme et al. initially defined a wait-state formation model suitable to identify root causes of wait states in collective and point-to-point communication. All of the wait states present in those

communication paradigms are based on process synchronization where one process explicitly waits for another process to start a specific activity. This chapter integrates the synchronization-based wait states in one-sided communication into Böhme's original wait-state formation model. Moreover, it presents a unified wait-state formation model by integrating lock contention as a new type of wait state, with formation characteristics different from the synchronization-based wait states, and introduces interval-delay as its root cause. Finally, this chapter shows how Böhme's performance indicators, such as the critical path, can be computed in the presence of contention-based wait states.

5. Evaluation

This chapter discusses performance measurements and analysis results obtained with the methods described in the previous chapters and implemented in the Scalasca parallel trace analyzer. As many of these measurements are correlated to a specific publication, not all measurements showcase the full range of methods discussed in this thesis. Nevertheless, in summary these measurements provide a good overview of the scalability and effectiveness of the presented methods, as indicated below:

SOR provides timings of all 5 phases during the analysis of trace measurements of a communication kernel across four different implementations including point-to-point and one-sided active and passive-target synchronization up to a scale of 65,536 processes.

BT-RMA provides results of porting a data-exchange pattern from point-to-point to one-sided communication using active-target synchronization in a well-known benchmark kernel and subsequently optimizing it by using the scalable wait-state detection.

CGPOP provides results of optimizing an available implementation of a data-exchange pattern using one-sided communication with active-target synchronization in a well-known proxy application using the scalable *critical path* and *root-cause* detection.

SRUMMA provides timings of the wait-state detection in passive-target synchronization in the one-sided communication interface ARMCI up to a scale of 32,768 processes.

NWChem provides measurement results for the simulation of the $SiOSi_3$ input with the NWChem simulation framework on 4,096 processes, acknowledging that progress-related wait states were a significant factor for the communication time on the IBM Blue Gene/P and their sparse distribution may fuel further imbalances throughout the execution.

Lock-Contention Microbenchmark provides measurement results for a controlled lock-contention scenario, demonstrating the effectiveness of lock-contention detection for MPI passive-target synchronization and the identification of waiting time and their root-causes.

Together, these case studies provide examples for the detection of wait states in one-sided communication with active and passive-target synchronization across two one-sided communication interfaces (MPI and ARMCI) and for the detection of root causes for such wait states and the critical path in applications using one-sided communication. Furthermore, they demonstrate how this information can be used to optimize the one-sided communication patterns in parallel applications.

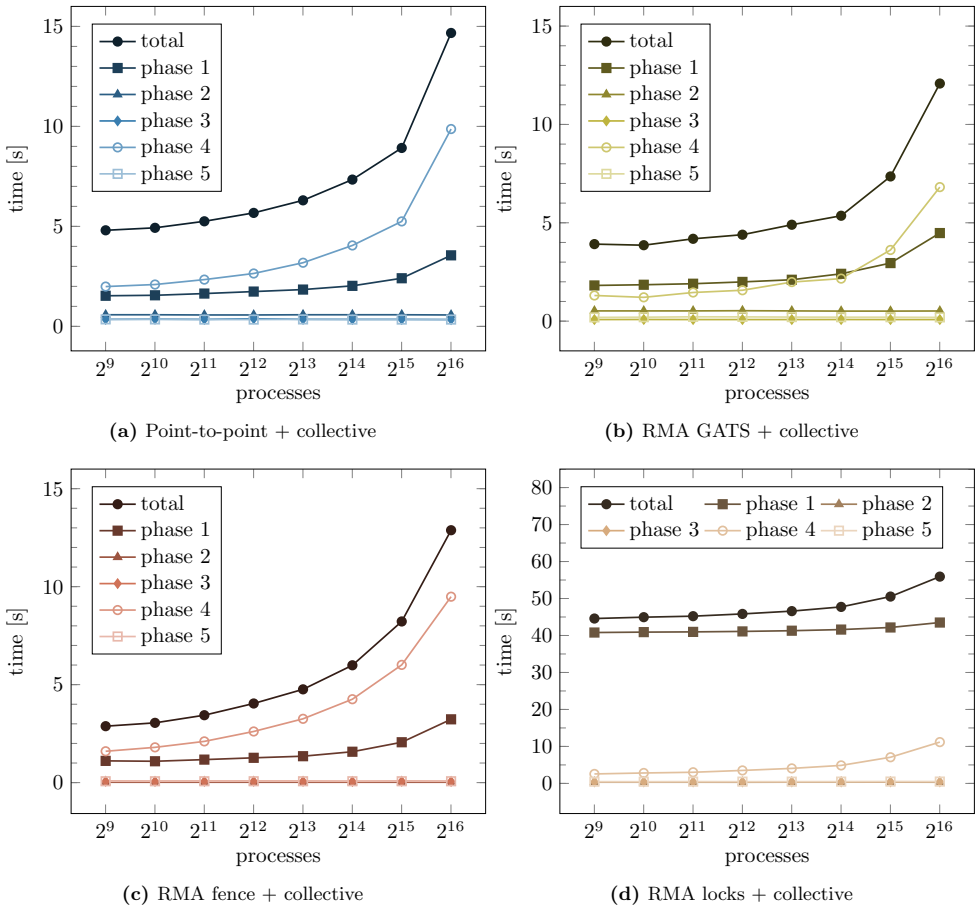
5.1. SOR

This case study focuses on the runtime of the five analysis phases of the Scalasca parallel analyzer as described in the previous chapters for four different implementations of a halo-exchange: (1) *p2p*, using the original point-to-point implementation, (2) *rma-fence*, using one-sided communication with collective active-target synchronization, (3) *rma-gats*, using one-sided communication with the group-based general active-target synchronization, and (4) *rma-locks*, using one-sided communication with passive-target synchronization. The timings for each phase are evaluated across a scale of 512 to 65,536 processes, to expose scale-dependent costs within each phase and demonstrate their general scaling behavior.

The SOR benchmark is a computational kernel that iteratively solves Poisson's equation using a red-black successive over-relaxation method on a two-dimensional grid. The communication pattern includes a nearest-neighbor halo exchange and a collective reduction for each iteration. The halo exchange, originally using point-to-point communication, was re-implemented to use one-sided communication in different synchronization schemes. The collective reduction is performed after each iteration to test for convergence. The benchmark allows both the problem size and the number of processes to be configured for a particular run, enabling both weak and strong scaling.

For the presented scaling measurements, the benchmark was configured for weak scaling, keeping the load per process constant. In principle, the benchmark runs either until the residual is below a given threshold or until the maximum number of iterations is reached. For this case study, the benchmark was configured to perform exactly 500 iterations by setting the residual tolerance threshold to zero. This way, premature convergence is prevented and each measurement performs exactly the configured maximum number of iterations with a significant amount of communication load on the one side and a predictable trace size on the other.

Figure 5.1 shows the time needed to analyze measurements of four different SOR nearest-neighbor exchange implementations: (1) point-to-point (fig. 5.1a), (2) one-sided communication using fence synchronization (fig. 5.1c), (3) one-sided communication using general active-target synchronization (fig. 5.1b), and (4) one-sided communication using passive-target synchronization (fig. 5.1d). Each plot provides graphs for the total analysis time, as well as the time broken down into the five phases. For all measurements, the phases two, three, and five show almost constant scaling across all investigated process counts with minimal influence on the total analysis time. These phases are mostly responsible for the detection of synchronization points and corresponding cost accounting for only a single wait state pattern, the *Late Receiver* in point-to-point communication. With the substitution of one-sided for point-to-point communication in the nearest-neighbor exchange in the application, these analysis phases have even less inter-process communication and only set up and manipulate local data structures. As such, the time needed for these phases are mostly dependent on the number of events in the process-local trace, which in the configuration of this scaling experiment stays mostly constant. The remaining main wait-state detection and cost accounting (phases one and four, respectively, as discussed in Section 4.3) therefore dominate the total analysis time. For Figures 5.1a to 5.1c, the fourth phase, which performs the majority of the cost accounting in backwards direction dominates the total time needed by the analysis. This is mainly due to two reasons: first, the messages exchanged during cost accounting are usually bigger, as they contain the vector of execution times in the



(a) Point-to-point + collective

(b) RMA GATS + collective

(c) RMA fence + collective

(d) RMA locks + collective

Figure 5.1.: Timings of the five analysis phases for measurements of the original SOR implementation using MPI point-to-point and collective communication, as well as modified implementations using various RMA synchronization schemes as replacements for the point-to-point communication. For the point-to-point and active-target synchronization scenarios the overall trend is similar with runtime mostly dominated by the cost accounting in the fourth phase. For the passive-target synchronization scenario the first phase with the detection of lock contention clearly dominates the overall runtime, however, its costs appear mostly scale-independent.

current synchronization interval, as well as the corresponding costs; second, especially the computation of collective waiting time turns out to be a costly operation at scale, as the causing process has to gather data to be distributed to the other waiting processes, leading to scale-dependent runtime costs, which clearly shows on all four figures. The detection of wait states in active-target synchronization constructs as part of the first analysis phase (see section 3.3) performs well, as the analysis can utilize events on both sides of a data exchange.

In scenarios involving passive target synchronization, as shown in Figure 5.1d, the wait-state detection dominates the total execution while showing good scalability overall. This is mostly expected, as the use of the active-message framework to analyze passive-target synchronization incurs additional runtime costs, compared to the analysis of active-target synchronization. First, the active message runtime needs to call a progress routine on every event visited; second, the active-message handlers are called *out-of-order* on the target side, thus the target process needs to identify the correct event in the trace; and third, the active messages are unexpected at the target, thus memory allocation for their reception has to occur ad-hoc and buffers are subsequently not pre-posted. For the nearest-neighbor exchange used in the benchmark, the overall scaling behavior is still very good. Scaling from 512 to $65,535$ processes, the analysis can maintain 93% parallel efficiency, as the work of active-message handlers is almost constant across processes, independent of the execution scale. A small scale-dependent influence is introduced by additional collective synchronization needed to ensure completion of pending active messages after each iteration.

In summary, this case study showed that the analysis of one-sided communication constructs with active-target synchronization performs on a par with the analysis of comparable point-to-point communication constructs. They perform significantly better than the detection of wait states in passive-target synchronization. Here, the case study revealed a much higher scale-independent overhead, due to the additional execution of the active-message runtime. Nonetheless, the overal active-message framework proves effective in the integration with the overall replay approach and shows good scalability. Future implementation improvements may increase its efficiency and bring it closer to the performance shown for active-target synchronization.

5.2. BT-RMA

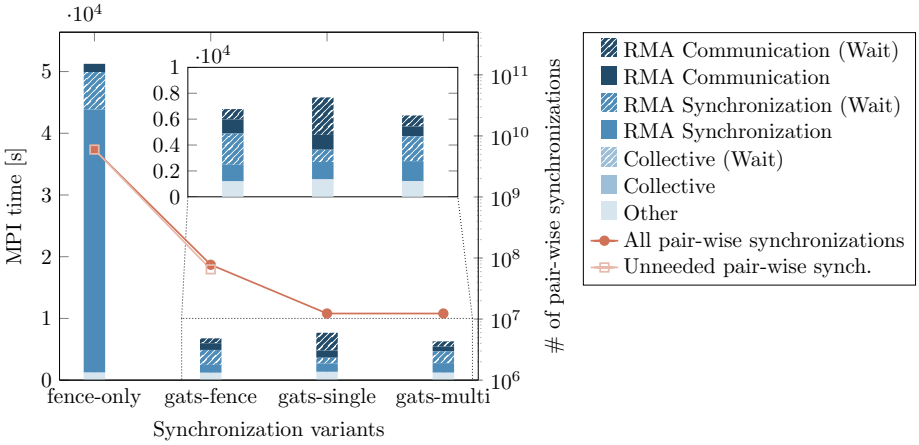
This case study focuses on the developer assistance during the substitution of one-sided for point-to-point communication in an existing benchmark code. It demonstrates how performance metrics obtained for one-sided communication influenced the optimization of the use of one-sided communication in the code. The following description of different implementations therefore follows potential choices of a programmer new to one-sided communication and its synchronization techniques; an expert programmer familiar with one-sided communication may directly choose a particular synchronization scheme, suitable for the target platform. The case study was intially conducted in the context of the initial publication of the scalable detection [2] on a fixed scale on one platform with changing implementations of the nearest-neighbor exchange. Further measurements were added here for different platforms and scales to demonstrate that on a different platform, developers could have come to different conclusions during the optimization procedures, as the severity of the detected wait states depends largely on the MPI implementation.

The BT-RMA benchmark is a modified version of the BT benchmark, part of the NAS Parallel Benchmark Suite 2.4 [11]. The BT benchmark solves three sets of uncoupled systems of equations in the three dimensions x, y, and z. The systems are block tridiagonal with 5×5 blocks. The domains are decomposed in each direction, with data exchange in each dimension during the solver part, as well as a so-called face exchange after each iteration. Those exchanges originally used non-blocking point-to-point communication. The modification replaces the point-to-point communication with one-sided communication using different flavors of synchronization schemes. For measurement purposes, five purely computational subroutines were excluded from instrumentation, lowering the runtime intrusion to about 1% and keeping the trace size manageable.

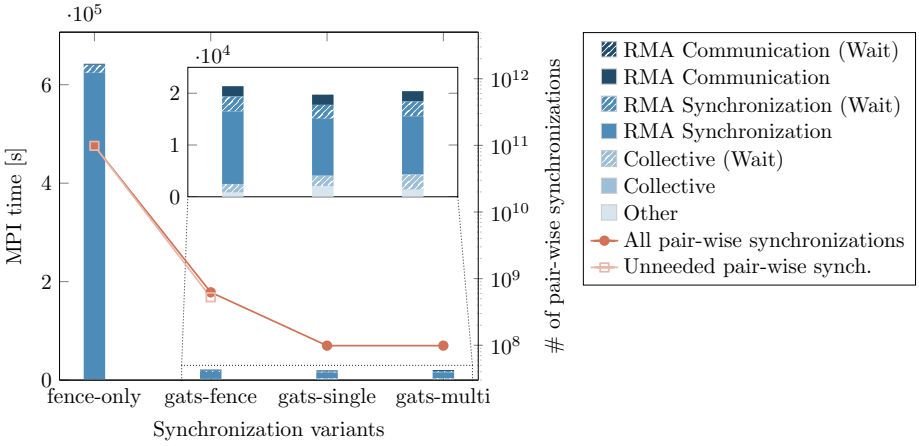
The initial development platform for the port of BT to one-sided communication was the IBM Power6 575 cluster JUMP2 of Forschungszentrum Jülich running AIX and POE. The target scale was 256 cores in ST mode, using the class D problem size. The time spent in MPI is shown in Figure 5.2a, separated into different sub-metrics available in the analysis reports for all four implementation variants. The measurement results used for the plot are printed in Table A.5 on page 121.

From a user's perspective, the simplest form of synchronization with the MPI one-sided interface is using fences. Fences synchronize all processes of a given window and do not need to specify targets and origins explicitly. Thus, we developed our initial version of BT-RMA using fence synchronization for both data exchanges. The analysis results of the *fence-only* implementation clearly reveals that the application spends a large portion of the MPI time (supposedly) productively in RMA synchronization, namely the `MPI_Win_fence` call. One-sided communication in MPI is inherently non-blocking and can be postponed by the runtime system until the next synchronization function. As such, the synchronization call may include the time for the data transfer as well as the process and memory synchronization. Here, the time spent in RMA synchronization accounts for more than 44% of the overall application runtime, while around 6% of the total runtime is waiting time. As only a fraction of the time spent in RMA synchronization is explicitly classified as waiting time, an undiscerning user may think this time to be spent productively advancing the communiucation. Most of this time is spent during the solver exchanges in the three dimensions. Using the performance metric *Pairwise RMA process synchronizations*—which indicated the number of process synchronizations with other processes, as explained in Section 3.2.2—showed that 98.1% of all pairwise synchronizations counted occur in the same synchronization calls that exhibit the excessive use of time. Even more, 99.8% of those pairwise synchronizations were unneeded as the solver only exchanges data to nearest neighbors.

To reduce the number of pairwise synchronizations between processes, we modified the solver to use general active-target synchronization (GATS), while the fence synchronization was left untouched in the face exchange. As Figure 5.2a clearly shows, the use of general active-target synchronization reduces the number of pairwise synchronizations by an order of magnitude. This also positively influences the overall time spent in RMA synchronization in general, leading to a seven-fold reduction in overall time spent in MPI during the application run. Although significantly faster, active target synchronization still accounts for about 4.2% of the application runtime, with Wait at Fence requiring 1.3% and Early Wait about 0.9%. In addition, this variant uses 2.5 times more time for remote access operations compared to the fence-only version, now spending 1.6% of the total time in the Early Transfer wait state. This indicates that in the



(a) IBM Power6 AIX Cluster JUMP2



(b) IBM Blue Gene/P JUGENE

Figure 5.2.: Time spent in MPI, broken down into waiting time and non-waiting time for communication and synchronization in BT-RMA on the IBM Power6 system JUMP2 and the IBM Blue Gene/P system JUGENE at Forschungszentrum Jülich. Bar plot shows time across all processes with hatched areas denoting waiting time. Line plot shows the number of pairwise RMA process synchronizations. For *gats-single* and *gats-multi* no unneeded synchronization were performed and the data points are omitted for clarity.

version using fence synchronization the MPI implementation is progressing more of the overall RMA communication during the fence calls themselves than when using general active target synchronization.

To reduce the waiting time in the collective fence synchronization and to eliminate the remaining unneeded synchronizations between processes—potentially causing some of the unwanted waiting time—we adapted the face exchange to use GATS synchronization with the same single window, originally used for the fence exchange (see *gats-single*). While it further reduced the synchronization time, waiting time in RMA operations increased significantly, ultimately leading to an overall performance regression. This different behavior suggests that during fence synchronization POE uses non-blocking RMA operations, whereas with GATS synchronization, it performs the RMA operations directly, leading to more *Late Post* wait states.

Although, now all synchronizations are needed, it is still performed en bloc for all communication partners, thus we created individual windows for each communication buffer (see *gats-multi*). Furthermore, we rearranged the GATS synchronization calls slightly, starting the exposure epochs as early as possible and shortening the access epochs by moving the start/complete calls close to the RMA transfers, decreasing the overall runtime again. In this configuration BT-RMA is almost 1.7x faster than the first fence-based version, and on a par with the original point-to-point communication.

With the availability of the IBM Blue Gene/P system JUGENE at Forschungszentrum Jülich, the BT-RMA benchmark was run again in all variants on problem size D with an increased scale of 1,024 processes in VN mode. Unfortunately, the general active target synchronization for the Blue Gene/P systems at the time of investigation had difficulties coping with skewed access and exposure epochs during the GATS synchronization of the solver phases, leaving the runtime system exiting unexpectedly. As a workaround, we inserted a barrier call after each solver step in the dimension x, y, and z when changing the synchronization mechanism to GATS. While these measurements are not particularly interesting in terms of a performance comparison to the original measurements on the JUMP2 system, they reveal several interesting changes in behavior when moving to this platform (see Figure 5.2b).

First, with the greater scale, the impact of the unneeded pairwise synchronizations is more severe; moving from fence to GATS synchronization for the solver results in a 30x reduction in time spent in MPI and an 3.6x performance increase overall. This is understandable, as the 4-fold increase in size leads to a 16-fold increase in pairwise synchronizations during the fence synchronization. Second, when using fence synchronization in the solver step (fence only) the inserted barrier calls hardly have any effect on the application behavior, as the fence calls implicitly synchronize the processes. Third, the MPI implementation of the Blue Gene/P seems to use non-blocking RMA operations, as the overall time spent in RMA communication across all synchronization schemes is minimal and the majority of the time is spent in synchronization, suggesting that the actual RMA operation call only initiate transfers, but most of the progress is done during subsequent synchronization. This last aspect suggests that the optimizations done for the multi-window synchronization variant are counter productive, leaving the single window version as the fastest option on this platform.

On a third system of investigation, the IBM Blue Gene/Q system JUQUEEN currently installed at Forschungszentrum Jülich, the behavior of the BT-RMA benchmark is yet again slightly dif-

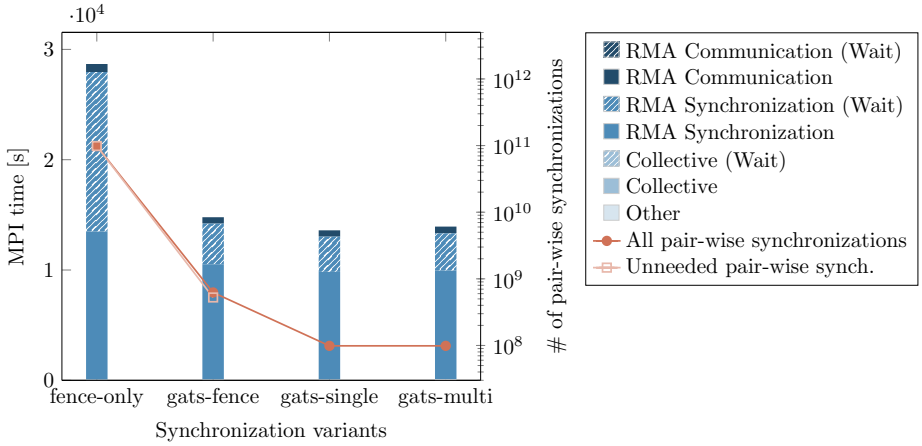


Figure 5.3.: Time spent in MPI, broken down into waiting time and non-waiting time for communication and synchronization in BT-RMA on the IBM Blue Gene/Q system JUQUEEN at Forschungszentrum Jülich. Bar plot shows time across all processes with hatched areas denoting waiting time. Line plot shows the number of pairwise RMA process synchronizations. For *gats-single* and *gats-multi* no unneeded synchronization were performed and the data points are omitted for clarity.

ferent, as shown in Figure 5.3. Like the MPI implementation of the Blue Gene/P, the Blue Gene/Q's MPI is a direct derivative of the popular open-source MPICH implementation. It is therefore not surprising that it follows a similar strategy in the use of non-blocking RMA operations in combination with general active-target synchronization. As a result, the communication time is very small, including only the time to set up the communication, performing the actual transfer either concurrently to the computation or within the subsequent synchronization call. Surprisingly, the MPI implementation deals much better with the fence synchronization in the solver steps, although one has to take into consideration that the Blue Gene/Q has a much higher core density compared to the Blue Gene/P, with 1,024 cores being available on half a midplane. As more of the communication can be performed using shared memory, the communication and synchronization will be significantly faster between the processes. Nevertheless, it remains evident that the high waiting time is connected to the unneeded synchronization with processes other than the direct communication partners.

In summary, this study showed how the detection of waiting time in MPI one-sided communication with active-target synchronization can guide performance optimization engagements across several implementation stages. Furthermore, with measurement results across three different HPC platforms and MPI implementations, it demonstrates that performance results and development decisions based on those results may differ between platforms.

5.3. CGPOP

This case study also focuses on the developer assistance through specific performance metrics during the implementation of a data exchange with one-sided communication, albeit, the performance metrics focused on in this case study are the *critical path* and *delay* metrics. Here, the application already had an initial implementation using one-sided communication, which was abandoned at some point in the development process due to lack of performance. The purpose of this study is to demonstrate the usefulness of the metrics in optimizing applications using MPI one-sided communication with active-target synchronization. The results were initially published as part of the scalable computation of those metrics for one-sided communication using active-target synchronization [5].

The CGPOP miniapp [149] represents the conjugate gradient solver of Los Alamos National Laboratory's Parallel Ocean Program (POP) 2.0, which is the ocean model of the Community Earth System Model (CESM) [164], a major climate code developed at the National Center for Atmospheric Research. CGPOP was created to study the most critical part of the application on different platforms without having to port the whole ocean simulation. It is implemented in several different variants, one of which uses one-sided, point-to-point, and collective communication. As such, it provides an interesting test case for studying inter-paradigm influences. As a test kernel, CGPOP provides different communication drivers to identify the communication scheme that suits a given platform best. Next to 1D and 2D point-to-point variants, it also provides a 1D halo exchange using one-sided communication and general active-target synchronization. The 1D decomposition uses a space-filling curve to partition the data. According to the developers, the one-sided kernel was not investigated deeply, as it did not seem to perform en-par with the two-sided kernels. Experiments to gain more insights on where the time was actually lost were conducted on a Linux/Infiniband cluster at RWTH Aachen University, using the 180×120 tile-set on 60 processes.

Before any performance measurements could be taken, the code needed slight modifications. Initially, group handles were frequently created and not freed, which exceeded the tracking capabilities of the Scalasca measurement system. As the measurement system already gives us detailed insight into the code's performance, we also disabled any application-internal timing calls. This slightly modified version was used as the baseline of our study. Because the I/O time needed to read in the input data was non-deterministic and dominated the overall execution time, we isolated the solver steps in `solver.esolver` within the analysis report to focus our attention on the performed iteration rather than the initialization. The initial measurements revealed two issues: (1) the barrier, called in the solver step right before the one-sided data exchange, experienced severe waiting time, and (2) despite the barrier, some origins experienced *Late Post* wait states (see Table A.8). Initially, the *Late Post* waiting time was not intuitive, as the barrier in front of it should have taken care of any imbalances leading to wait states. However, the root-cause analysis revealed that an imbalanced barrier completion is responsible for the *Late Post* wait state. This is an example of a cross-paradigm wait state, where wait states or imbalances in one communication paradigm influence other paradigms as well. The waiting time in the barrier is caused by delays in the function `matrix_mod_matvec` and its parent function `pcg_chrongear_linear`. For both functions, the delay costs identify two processes as the main contributors to the overall waiting time. The waiting time itself, however, is more

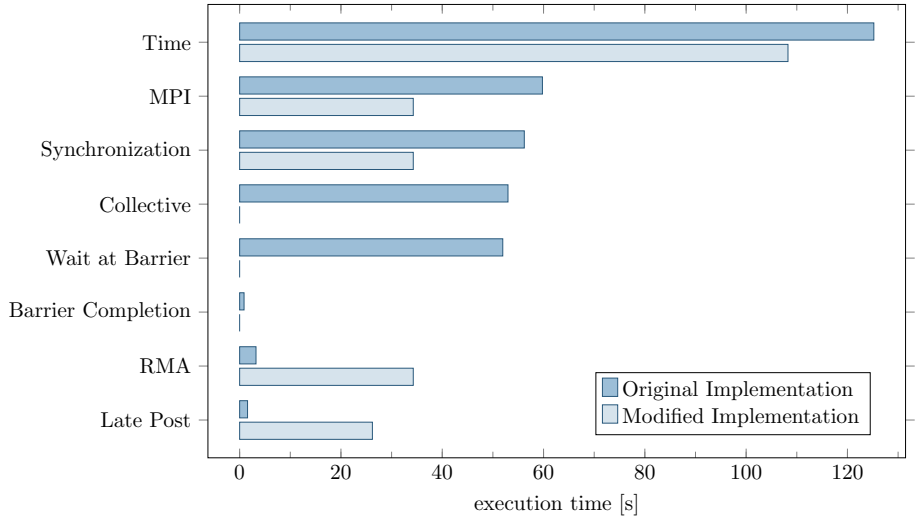
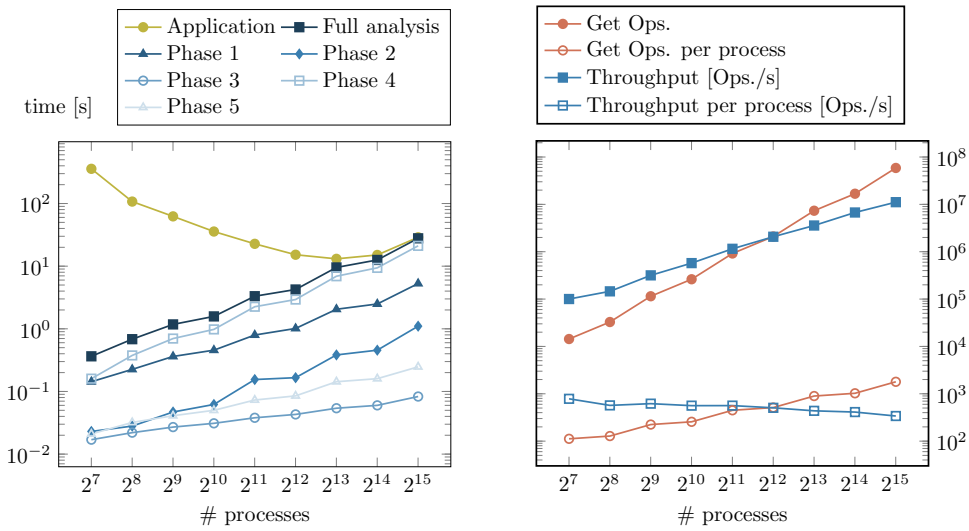


Figure 5.4.: Comparison of runtimes between the original one-sided implementation of CGPOP and the modified version.

wide-spread over the processes. With the underlying nearest-neighbor exchange, this indicates that the barrier synchronization may be too heavy-weight in this case. As the barrier itself is not functionally necessary at this point in the code—the one-sided synchronization itself will take care of consistency—it was removed to see how the waiting time caused by the delay materializes in a more light-weight synchronization. As expected, the modifications partly dissolved wait states due to the lighter-weight synchronization, which partly reappeared as *Late Post* wait states. After all, the actual delay causing the initial barrier wait states was not changed. Overall, the waiting time decreased and the application core already showed a significant runtime improvement. Furthermore, the critical-path profile shows both aforementioned user functions (`matrix_mod_matvec` and `pcg_chrongear_linear`) to be on the critical path and indicates a significant imbalance. A logical next step would now be to find ways of removing this load imbalance, which, however, is beyond the scope of this thesis and should be done in closer collaboration with the original developers.

In summary, this study demonstrated how the performance metrics of *critical path* and *delay* could be used to identify root causes of underperforming communication pattern in one-sided communication with active-target synchronization. Furthermore, it revealed that depending on the implementation, processes might exit a barrier less aligned than expected, leading to imbalance after the barrier. This means, though beyond the direct control of the user, barriers can also be a (usually small) source of delay and their completion time should be accounted during the root-cause analysis.



(a) Analysis phases

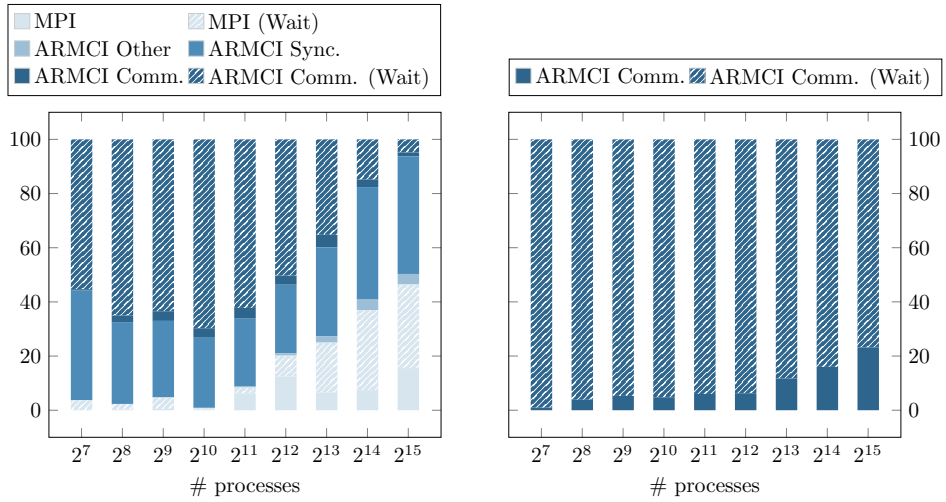
(b) Event throughput

Figure 5.5.: Scaling behavior of the different analysis phases (left) during the analysis of the strong-scaling measurements of the SRUMMA benchmark. The timings for the application (green) reveal that beyond 8,192 processes the decreasing local work and increasing communication leads to a performance regression. The timings for the five analysis phases (blue) also show that the increasing event count per process impacts the overall analysis time. With *Get* events increasingly dominating the overall event counts, the throughput of the active-message framework shows good scalability (right).

5.4. SRUMMA

This case study focuses on the scalability of the active-message framework for analyzing wait states in one-sided communication with passive-target synchronization in a communication kernel of a matrix-matrix multiplication. Demonstrating the applicability of the wait-state detection for one-sided communication beyond MPI, the investigated communication kernel is using ARMCI, the one-sided communication interface used by the well-known partitioned global-address space library Global Arrays.

The SRUMMA algorithm [84] for scalable matrix-matrix multiplication is a test case for the scalability of the active-message framework developed to detect wait states in passive-target one-sided communication. The procedure, which is based on remote memory access, is implemented in Global Arrays to support the multiplication of distributed global arrays. This algorithm, invoked as the `ga_dgemm` call, employs an owner-computes model with each process computing a block of the output matrix. The relevant blocks of the input matrices are obtained through non-blocking get operations. The different block-block products are structured to avoid contention from numerous simultaneous get requests directed at the same target process.



(a) Overall communication/synchronization

(b) ARMCI communication only

Figure 5.6.: Normalized performance metrics of MPI and ARMCI in strong-scaling measurements of the SRUMMA benchmark. Initially, the progress-related waiting time dominates the overall communication (left figure; dark blue hatched area). At larger scales, the waiting time in the collective communication, induced by ARMCI synchronization, takes over the dominant role (left figure; light blue hatched area). Across the scales, the percentage of waiting time in ARMCI communication is slightly reduced (right figure), as computation phases get shorter and more processes are communicating concurrently—providing progress for remote accesses.

The measurements are configured for strong scaling of the multiplication of two 4096×4096-element matrices on different numbers of processes. The square property of the matrix, coupled with the blocked data distribution of the global arrays, results in all processes performing the same number of floating-pointing operations between communication calls. The symmetry is only broken by the differences in the cost of communication due to topological asymmetries. Such regular calculations with seemingly coordinated communication are typically not expected to incur a great *Wait for Progress* penalty. The measurements of this benchmark were performed on the IBM Blue Gene/P system JUGENE at Forschungszentrum Jülich, which was decommissioned at the end of 2012. Figure 5.5a shows the overall strong scaling behavior of the benchmark. It can be seen that beyond 8,192 processes, the application fails to scale any further. This is due to the increasing lack of computational work performed by the individual processes and the benchmark-internal synchronization using `ARMCI_AllFence()` and `MPI_Barrier()` starting to dominate the time of the benchmark run. Executed on 8,192 processes, about 79% of the total benchmark time is spent in either ARMCI or MPI functions, with about 42% of the total benchmark time being waiting time. Investigating the delay costs for the *Wait at Barrier* wait states reveals that their root cause lies in delays induced by `ARMCI_AllFence()`. As the synchronization time of the benchmark was not the primary focus of these experiments, they were not investigated any further.

While the initial implementation of the *Wait for Progress* detection was based on an implementation using ARMCI [4], the scaling timings of the different analysis phases shown in Figure 5.5a are based on the new MPI-based implementation. Figure 5.5a shows that (1) the scaling behavior of the analysis is independent of the applications scaling behavior and (2) the analysis time increases across all scales. Figure 5.5b reveals that the scaling behavior of the analysis depends on the number of one-sided operations performed by the application, or in other words, the number of requests that need to be handled in the system. In the data-decomposition scheme present in the benchmark, the number of communication calls grows linearly with the number of processes with an estimated coefficient of 2.5, leading to the observed growth in analysis time because the local request-response handling of the local processes is saturated. In the new implementation, receive buffers are created ad hoc, as opposed to the pre-allocated dedicated communication buffers per process of the original implementation. While this is much more scalable in terms of memory requirements, it means that every active message is *unexpected* in terms of the MPI communication layer. Too many unexpected messages at any given time may negatively impact the per-message processing time.

The benchmark application has a balanced load of remote-memory-access operations per process. For the analyzer, this means the number of requests necessary to analyze the complete behavior is also quite balanced across all processes. With the largest measurement using 32,768 processes, the analyzer was able to maintain a processing speed of 11 million get operations per second, thus handling an overall workload of almost 59 million get operations. In general, communication patterns that overburden a single process with data accesses tend to reduce the overall performance of the parallel analysis, as the overburdened process will dominate the overall analysis time. Request coalescing may further improve such situations, however, pathological communication patterns may continue to influence the analysis performance.

Figure 5.6a shows an excerpt from the application's performance metrics, indicating that with the decomposition of the matrices across more processes, the execution is increasingly dominated by synchronization. This can be explained by the complexity of the all-fence operations. Additionally, the figure shows a significant amount of waiting time in ARMCI communication—i.e., *Wait for Progress* wait states—which is contrasted with actual communication in Figure 5.6b at different scales. The dominance of waiting time in comparison to actual communication is significant. On the given platform, additional progress threads would automatically use a full core, not allowing multiple progress threads to share a core. Therefore, the overall fraction of waiting time is still low enough that the use of a dedicated core to run a helper thread is not justified in this scenario. The severity of the pattern is attenuated slightly at larger scales, as the computation load decreases and the probability of a target process to immediately provide progress increases. Although it is expected that the impact of remote progress to be observable for communication patterns as used by this benchmark, its dominance of the ARMCI communication profile is nonetheless surprising for fairly regular communication phases. For applications with more irregular communication patterns, where individual processes communicate in irregular intervals, the impact of progress-related wait states may therefore be even higher, as the likelihood of concurrent communication decreases.

In summary, this case study shows that progress-related wait states can significantly impact the communication performance and induce further imbalance into a code. With the detection and quantification of such waiting time as presented here, developers can obtain the information for

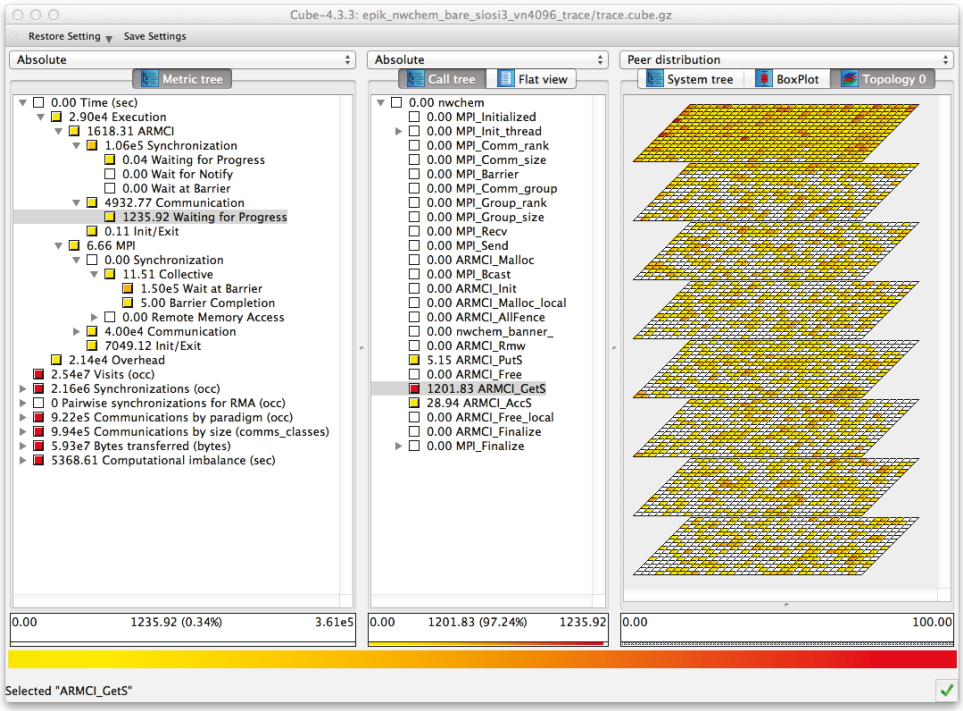


Figure 5.7.: Performance analysis report of a simulation run of NWChem using the $SiOSi_3$ input conducted on 4096 cores on the IBM BlueGene/P system JUGENE of Forschungszentrum Jülich.

an informed decision on how to configure and run their simulations on platforms without explicit software or hardware progress. While offering potential for further improving communication performance and scalability, the presented analysis framework showed acceptable performance on measurements up to 32,768 processes, proving the overall design to be feasible. Furthermore, it demonstrates the applicability of this approach to investigate the performance of other one-sided communication interfaces besides MPI, such as ARMCI.

5.5. NWChem

This case study focuses on the presence of progress-related wait states in a real-world simulation scenario. It shows how the performance analysis can help quantify the overall loss in performance of PGAS-style computations on a platform lacking remote communication progress.

The NWChem [166] framework for computational chemistry serves as a real life example for the influence of progress-related wait states on the overall performance of a parallel application. The

framework allows users to chose from multiple different algorithms common to computational chemistry. The benchmark configured for these measurements performs a density functional theory (DFT) calculation on the $SiOSi_3$ input. DFT is a widely used single-determinant approach to the many-electron problem [82, 127, 129]. All integral evaluations were performed using the direct method, and the Fock matrix was constructed using the distributed data approach [53].

The NWChem framework exhibits a vast amount of call paths that would normally raise the memory requirements significantly during measurement. Moreover, it comprises a lot of short functions called with a high frequency, which would impact both memory requirements and measurement overhead, potentially leading to drastically perturbed measurements, rendering such measurements useless. To minimize the runtime overhead, the NWchem binary was only instrumented at the main routine—to provide a root call-path node. This binary was then linked with a pre-instrumented ARMCI library. As a result, the executable only contained minimal instrumentation. The measurement was conducted on the IBM Blue Gene/P system JUGENE at Forschungszentrum Jülich on 4,096 processes using the $SiOSi_3$ input. Figure 5.7 shows in the middle pane that all ARMCI calls are directly connected to the root cnode `nwchem` and no other call path information is present. In the left pane, the major source of waiting time is computational imbalance at global synchronization points. Although the *Wait for Progress* inefficiency pattern is not a major part of the overall waiting time in the presented NWChem measurement—the overall waiting time is dominated by *Wait at Barrier*—it still has a significant influence on the one-sided communication employed by Global Arrays. In this example, it accounts for 20 percent of the ARMCI communication time that is used for the one-sided data accesses. Figure 5.7 also reveals that the attributed waiting time is sparsely scattered across the processes. It is likely that these highly irregular waiting times induce imbalances which themselves lead to further waiting times—a possibility that should be subject of a more detailed investigation of NWChem performance, which is out of the scope of this work.

This case study demonstrates that progress-related wait states do occur in the real-world applications. Especially with higher-level frameworks building on one-sided communication, such as PGAS runtimes, these may introduce chaotic, noise-like delays throughout the application execution that are non-trivial to mitigate. In this context, it exemplifies that the wait-state detection can be helpful for end user but also implementors of such frameworks to quantify performance loss in parallel applications, as the basis for further action.

5.6. Lock Contention Microbenchmark

This case study focuses on the correct detection of lock-contention wait states, their root causes, and the identification of the critical path in applications using MPI one-sided communication with passive-target synchronization. It verifies the detection and cost accounting algorithms presented in Sections 3.3.2 and 4.3, respectively, and shows how users need to interpret the resulting performance profile.

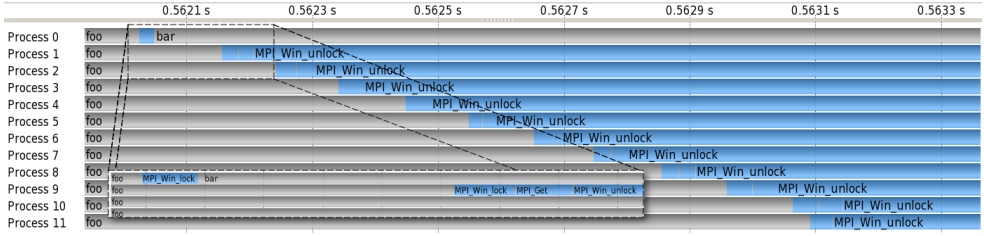
The overall scenario is similar to the one shown in Figure 4.2 on page 74: a lock contention scenario enclosed in collective barrier synchronization. Processes are partitioned into process 0 acting as the target for all RMA operations, and the rest of the processes, scheduling RMA operations to update the window on the target process. After an initial barrier synchronization

of all processes, all processes call the function `foo()` to simulate work with process-individual workloads. The simulated workload is the lowest on target rank 0 and increases with rank, thus the processes return from `foo()` in rank order. As the target has the lowest workload, it is the first to return from the call to `foo()` and is guaranteed to lock its local window before any of the other processes requested the lock. Locks on the local window are never postponed but block until the lock is successfully acquired, as a local lock epoch needs to ensure that local loads and stores to the window are appropriately protected. While the target holds the lock, it executes the function `bar()` for 2 seconds to simulate local updates to the window before releasing the lock again. The skew in the workload simulated by `foo()` ensures that the workers request the lock after it has been acquired by the target rank 0. They form a contention chain waiting for rank 0 to release the lock. Each process calls `foo()` again for a duration of 100 microseconds after its release of the lock. Finally, all processes are synchronized by another barrier operation.

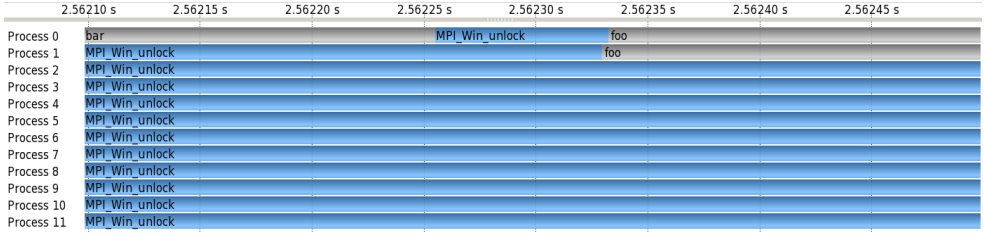
The skew of the processes after completing the remote memory access leading to a subsequent *Wait at Barrier* wait state is independent of the initial skew induced by the calls to `foo` on the different processes; it only depends on the time needed to complete the RMA access and to pass the lock ownership to the next process.

The benchmark was executed on two nodes of a Linux Cluster with InfiniBand network using Open-MPI 1.10.2. Figure 5.8 shows screenshots of Vampir timeline views of selected regions of the measurement, as well as the corresponding Cube report as generated by Scalasca's trace analyzer. In the timeline views, user functions are shown in grey and MPI functions are shown in blue. Figure 5.8a shows the start of the lock contention, where each process initially calls function `foo()` for a rank-dependent duration. At time $0.5620s$ process 0 locks its local window—indicated by the short blue rectangles between the rectangles corresponding to calls to `foo()` and `bar()`. This call to `MPI_Win_lock()` is too short on all participating processes for Vampir to place the name of the call in the respective timeline at this zoom level, thus it is represented only by a short blue rectangle. The same applies to the RMA operations following the locks on processes 1 and higher. Process 0, as the target, obtains an exclusive lock and executes the function `bar()` for 2 seconds. The remaining processes each block in the call to `MPI_Win_unlock()`, waiting for the target to release the lock. Figure 5.8b shows a detailed view of the time interval in which the target releases its lock on the window and passes the lock to process 1. Process 1 obtains the lock and performs its RMA operation, releasing the lock again. Process 2, however, is unable to obtain the lock directly from process 1, as the target (process 0) is busy with the execution of `foo()` after its release of the lock and does not provide target-side progress for remote accesses. Process 2 can obtain the lock only after process 0 provides progress again within the barrier operation (Figure 5.8c). The total waiting time amounts to 2.1 seconds—2 seconds of `bar()` (during the initial lock) and 0.1 seconds of `foo()` after the release of the lock awaiting further target-side progress. As the barrier spans all processes, process 0 has to wait for the last process to join and continues to provide progress for all remaining processes. The call to `foo()` before the barrier is rank independent and lasts for 100 microseconds.

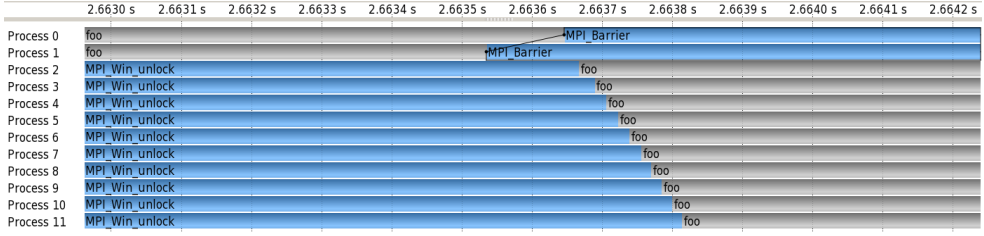
The Cube performance report shown in Figure 5.9a reflects the observed behavior. The time spent in the *Lock Contention* wait state is about 2 seconds for process 1, which requested the lock right after process 0 and had to wait for the end of the 2 second execution of `bar()`. The waiting time on process 2 is not classified as *Lock Contention* but as *Wait for Progress* (not directly shown), as insufficient progress was the last factor extending the overall waiting



(a) Start of lock contention

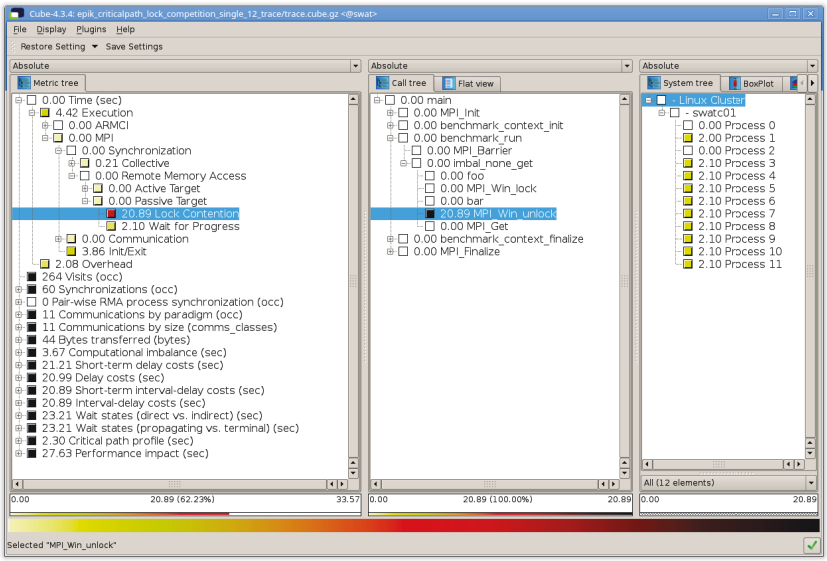


(b) Unlock of initial lock

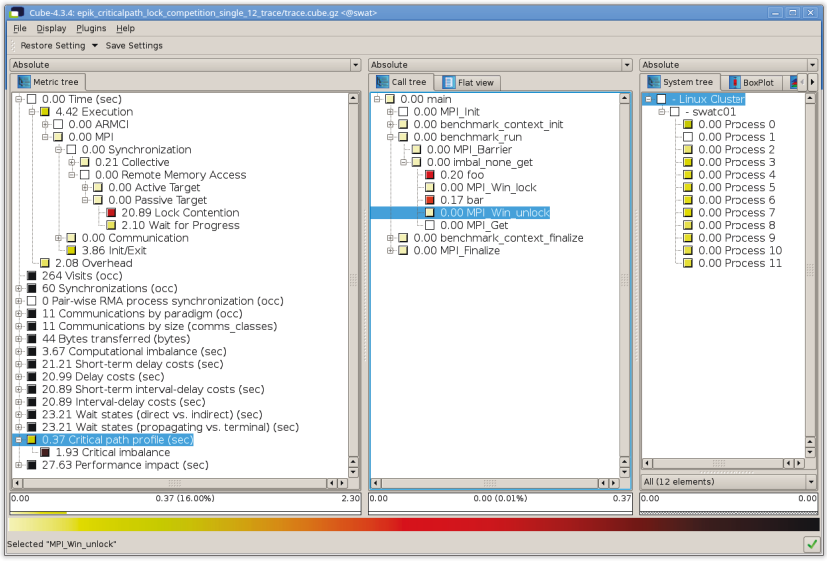


(c) End of lock contention

Figure 5.8.: Timeline view of the start and the end of a lock contention scenario. The execution time of the initial call to function `foo()` is skewed across the processes to ensure that process 0 acquires the lock first and the accesses of the remaining processes are queued. When process 0 releases the lock after the call to function `bar()`, process 1 completes its access and releases the lock again, yet process 0 does not provide further progress for the completion on the other processes. Starting the enclosing barrier synchronization, process 0 provides progress again, enabling completion of the pending lock epochs on the remaining processes.



(a) Cube analysis report highlighting *Lock Contention* time.



(b) Cube analysis report highlighting *Critical Path Profile* time.

Figure 5.9.: Cube analysis report highlighting *Lock Contention* time and the time distribution of the *Critical Path Profile*.

time. However, for the remaining processes, progress was provided and the waiting time is classified as contention-based. The waiting time on processes 2 and higher is increased by about 100 microseconds compared to process 1 as further progress was only provided again after the execution of `foo()` on the target process.

Highlighted in Figure 5.9b is the contribution to the critical path of each process in the function `MPI_Win_unlock`, which illustrates the serialization of processes during the lock contention. Only process 1 is not present on the critical path owing to process 2 not waiting for the process 1 to return the lock but for process 0 to provide target-side progress and grant the lock. Furthermore, the analysis report suggests that the benchmark's overall execution time could be reduced by a better load balancing of the function call `bar()` across all processes. In the measurement, the full 2 seconds of the call to `bar()` on the target process 0 is part of the critical path. If balanced, only 0.17 seconds (2 seconds spread across 12 processes) would remain on the critical path. The remaining 1.83 seconds of the call to `bar()` are part of the 1.93 seconds of *Critical Imbalance* time. It is important to note that the analysis cannot determine whether `bar()` can be balanced across multiple processes, so it is up to the developer to decide on a viable optimization strategy.

In summary, this case study shows that lock contention is correctly identified by the analysis. Furthermore, it showed that depending on the MPI implementation, wait states due to insufficient target-side progress may also influence the waiting time withing a contention chain, as origin processes may wait for both, an origin process to release the lock and the target process to grant the lock again after its release. This case study also highlighted the critical-path metrics in such a scenario and how to interpret them to obtain optimization targets.

Summary

This chapter presents analysis results of a variety of applications using one-sided communication, both of MPI and ARMCI. The scaling study of the SOR benchmark demonstrates the scalability of the presented approaches, ensuring that application developers can inspect application behavior and detect waiting times at large scales. The study of a modified version of the NAS Parallel Benchmark's BT kernel using general active-target synchronization in different scenarios shows how the identified wait states differ on different platforms and how the presented methods can assist developers in fine tuning one-sided synchronization and access patterns. Moreover, the analysis of the CGPOP benchmark demonstrates how wait states in multi-paradigm communication scenarios can interact and underline the necessity of a holistic tool support of communication paradigms available to developers. The studies of SCRUMMA and NWChem demonstrate the effectiveness of the contributed analysis framework based on a high-level active-message scheme. Finally, the lock-contention micro benchmark demonstrates the effective identification of lock contention and lack of target-side progress on an easy to follow example, while also providing insight on the interpretation of the corresponding root-cause and critical-path metrics in the analysis report.

6. Conclusion and outlook

This chapter summarizes the motivation and contributions to the performance analysis of one-sided communication, as described in this thesis. Furthermore, it presents ideas for new research areas to follow up on the presented work.

Message passing is one of the most prominent programming paradigms in high-performance computing. Users can choose from three main communication paradigms: point-to-point, collective, and one-sided. While the first two are often well supported by many tools, the latter often suffers from no or only partial support. However, for developers to use a programming paradigm effectively, comprehensive tool support is essential to the understanding of the behavior of the application. Lack of tool support may therefore drive users away from using one-sided communication, even if it would fit the intended use in an algorithm or application. Focussing on scalable performance analysis techniques, this thesis helps to close the existing gap, and enables effective tuning of parallel applications using one-sided communication.

On the path toward this goal, this thesis presents a generic event model for one-sided communication, capable of describing semantics of multiple one-sided communication interfaces (e.g., MPI, SHMEM, and ARMCI) in use in high-performance computing today. Using this event model, performance analysis tools, such as Scalasca, can define wait-state patterns that help to identify and quantify the time a process is idle, waiting for a remote event to occur. This new generic event model is used to describe well-studied as well as previously unstudied wait-state patterns across multiple one-sided communication interfaces. Based on these definitions, this thesis further presents scalable methods for identifying inefficient communication and synchronization behavior in large-scale event traces. To enable the presented analysis techniques of passive-target synchronization, this work also contributes an active-message communication framework. Extending the existing communication infrastructure of Scalasca, which was limited to the replay of explicitly recorded interaction among processes, this framework permits analysis methods to send information between arbitrary processes while still being embedded in Scalasca's general replay algorithm.

Building on prior work of Böhme et al., this work extends the detection of root causes of wait states and the identification of the critical path in message passing applications to support one-sided communication. A crucial contribution to the support of lock-based synchronization in the root-cause detection is the introduction of contention points. Such points of process synchronization differ significantly from the type of synchronization point in parallel applications that were the basis for Böhme et al.'s analysis. This dissertation extends the formal framework used to identify root causes of wait states and expose the critical path in message passing applications, integrating contention points into Böhme et al.'s cost model. In doing so, this work presents advanced performance analysis techniques for one-sided communication on a par with

similar methods existing for point-to-point and collective communication, enabling developers to choose one-sided communication in suitable scenarios without accepting a lack of tool support.

This dissertation also provides starting points for further research in performance analysis of one-sided communication. The analysis methods presented in this thesis still need to be ported into the production versions of the Score-P measurement system and the Scalasca parallel analyzer and released for public use. This work is already in progress. Exposure to a broader audience will present further chances to refine the implementation.

The root-cause detection of lock contention in one-sided communication can also be extended to cover shared-memory programming paradigms. Lock-based synchronization is a common technique to synchronize execution streams in shared-memory programming interfaces, such as POSIX threads and OpenMP. Both interfaces are often used in high-performance computing to handle intra-node-level parallelism, while MPI is used for inter-node-level parallelism. Recently, Böhme et al. already extended their techniques to OpenMP parallelism [25]; the support of root-cause detection in lock-based synchronization in OpenMP and similar interfaces would be a natural next step.

Furthermore, contention also occurs in other situations common to parallel programming where mutual exclusion needs to be guaranteed: network links, file systems, or the CPU memory system. Similar to the lock-based synchronization in one-sided communication, the synchronization in these scenarios is implicit. Access order and synchronization partners need to be inferred through secondary information. In some cases, a medium may also multiplex parts of several concurrent accesses, creating an additional layer of complexity. To support these scenarios, measurement systems would have to be extended to record the necessary information, data exchange algorithms have to be developed, and in the case of multiplexed concurrency, cost attribution may need to be adapted.

Overall scalability in terms of memory consumption may also benefit from exploring different load balancing techniques for gathering lock-epoch information. Instead of collecting the full epoch information at the target, temporally distinct parts of the contention chain could be gathered across multiple processes, lowering the memory footprint of the individual processes, especially in scenarios where a huge number of processes access a single target. Processes with connecting parts of the contention chain would then exchange data of the first and last epochs with the corresponding neighbors.

As one-sided communication is often used as the communication layer for higher-level programming paradigms, particularly partitioned global address space languages, the methods shown here may also function as the basis for higher-level metrics helping users to decide which instructions lead to wait states in the communication infrastructure. Finally, new research opportunities may also arise with the advent of new performance analysis interfaces such as MPI_T and a callback interface currently in design by the MPI For:w um. As soon as a broader range of MPI implementations support these emerging interfaces, the information gained through them may benefit tools greatly. For example, an implementation may trigger a callback on the target when a process acquires a lock and enable a tool to record the provided information. Then, heuristics to determine lock order can easily be replaced with explicitly recorded information.

A. Measurement Data

A.1. SOR

Table A.1.: Execution times in seconds of the parallel analysis of the SOR benchmark using point-to-point communication on scales from 512 to 65,535 processes on the IBM Blue Gene/Q system JUQUEEN at Forschungszentrum Jülich.

Processes	Total [s]	Phase [s]				
		1	2	3	4	5
512	4.80	1.53	0.58	0.35	1.99	0.35
1,024	4.93	1.55	0.58	0.36	2.09	0.35
2,048	5.25	1.64	0.57	0.35	2.34	0.34
4,096	5.67	1.74	0.57	0.38	2.64	0.34
8,192	6.30	1.84	0.58	0.35	3.18	0.35
16,384	7.34	2.03	0.58	0.35	4.05	0.34
32,768	8.92	2.41	0.58	0.35	5.25	0.34
65,536	14.67	3.55	0.57	0.34	9.87	0.34

Table A.2.: Execution times in seconds of the parallel analysis of the SOR benchmark using one-sided communication with general active-target synchronization on scales from 512 to 65,535 processes on the IBM Blue Gene/Q system JUQUEEN at Forschungszentrum Jülich.

Processes	Total [s]	Phase [s]				
		1	2	3	4	5
512	3.92	1.81	0.52	0.08	1.30	0.19
1,024	3.86	1.85	0.52	0.08	1.21	0.20
2,048	4.19	1.90	0.52	0.08	1.46	0.22
4,096	4.39	1.99	0.53	0.08	1.57	0.22
8,192	4.90	2.10	0.52	0.08	1.99	0.21
16,384	5.36	2.41	0.51	0.08	2.16	0.20
32,768	7.36	2.95	0.51	0.08	3.62	0.20
65,536	12.08	4.48	0.52	0.08	6.81	0.19

Table A.3.: Execution times in seconds of the parallel analysis of the SOR benchmark using one-sided communication with fence synchronization on scales from 512 to 65,535 processes on the IBM Blue Gene/Q system JUQUEEN at Forschungszentrum Jülich.

Processes	Total [s]	Phase [s]				
		1	2	3	4	5
512	2.88	1.11	0.02	0.07	1.60	0.07
1,024	3.05	1.09	0.02	0.07	1.80	0.07
2,048	3.44	1.17	0.02	0.07	2.10	0.07
4,096	4.04	1.27	0.02	0.07	2.61	0.07
8,192	4.76	1.35	0.02	0.07	3.26	0.07
16,384	5.99	1.58	0.02	0.06	4.26	0.07
32,768	8.22	2.06	0.02	0.07	6.01	0.07
65,536	12.88	3.23	0.02	0.06	9.49	0.07

Table A.4.: Execution times of the parallel analysis of the SOR benchmark using one-sided communication with passive-target synchronization on scales from 512 to 65,535 processes on the IBM Blue Gene/Q system JUQUEEN at Forschungszentrum Jülich.

Processes	Total [s]	Phase [s]				
		1	2	3	4	5
512	44.59	40.80	0.33	0.43	2.56	0.47
1,024	44.95	40.91	0.33	0.43	2.82	0.46
2,048	45.22	40.97	0.33	0.43	3.01	0.48
4,096	45.83	41.10	0.32	0.42	3.51	0.48
8,192	46.59	41.29	0.34	0.43	4.06	0.47
16,384	47.73	41.60	0.34	0.43	4.87	0.49
32,768	50.52	42.17	0.34	0.43	7.07	0.50
65,536	55.92	43.50	0.32	0.43	11.17	0.50

A.2. BT-RMA

Table A.5.: Performance metrics for different variants of BT-RMA running on 256 cores of the IBM Power6 575 system JUMP2. All values are aggregated across all processes and inclusive, that is, they include the time for sub-patterns (indicated through indentation).

Metric [s]	fence-only	gats-fence	gats-single	gats-multi
Total time	109,361.68	61,197.15	61,958.75	60,503.99
Execution time	58,109.23	54,420.63	54,280.99	54,219.74
MPI time	51,252.45	6,776.52	7,677.76	6,284.25
Collective	0.77	1.37	5.89	1.53
Wait at Barrier	0.75	1.34	5.87	1.51
RMA Synchronization	48,703.78	3,701.09	2,294.03	3,476.01
Wait at Fence	6,080.03	878.34	–	–
Early Wait	–	1,536.29	912.73	1,923.75
Late Complete	–	1.84	40.26	1.99
Late Post	–	0.47	2.17	0.92
RMA Communication	1,324.92	1,881.20	4,034.39	1,603.23
Late Post	–	797.09	2,893.84	848.56

Metric [occ.]	fence-only	gats-fence	gats-single	gats-multi
Pair-wise sync.	$5.988 \cdot 10^9$	$7.763 \cdot 10^7$	$1.234 \cdot 10^7$	$1.234 \cdot 10^7$
Unneeded pair-wise sync.	$5.976 \cdot 10^9$	$6.529 \cdot 10^7$	–	–

Table A.6.: Performance metrics for different variants of BT-RMA running on 1,024 cores of the IBM Blue Gene/P system JUGENE. All values are aggregated across all processes and inclusive, that is, they include the time for sub-patterns (indicated through indentation).

Metric [s]	fence-only	gats-fence	gats-single	gats-multi
Total time	862,852.50	235,560.79	233,595.01	234,197.46
Execution time	220,392.94	214,204.01	213,868.62	213,798.25
MPI time	642,459.56	21,356.78	19,726.39	20,399.21
Collective	246.88	1,686.25	2,088.60	2,938.56
Wait at Barrier	238.62	1,674.79	2,076.28	2,926.07
RMA Synchronization	639,439.77	16,901.96	13,675.91	14,115.94
Wait at Fence	16,302.48	187.76	–	–
Early Wait	–	2,656.19	2,661.14	2,824.42
Late Complete	–	43.45	40.92	41.88
Late Post	–	–	–	–
RMA Communication	1,920.77	1,980.18	1,972.15	1,981.52
Late Post	–	–	–	–

Metric [occ.]	fence-only	gats-fence	gats-single	gats-multi
Pair-wise sync.	$9.844{\cdot}10^{10}$	$6.241{\cdot}10^{8}$	$9.871{\cdot}10^{7}$	$9.871{\cdot}10^{7}$
Unneeded pair-wise sync.	$9.832{\cdot}10^{10}$	$5.253{\cdot}10^{8}$	–	–

Table A.7.: Performance metrics for different variants of BT-RMA running on 1,024 cores of the IBM Blue Gene/Q system JUQUEEN. All values are aggregated across all processes and inclusive, that is, they include the time for sub-patterns (indicated through indentation).

Metric [s]	fence-only	gats-fence	gats-single	gats-multi
Total time	230,257.89	176,749.02	174,062.99	174,520.64
Execution time	201,582.36	161,964.02	160,466.29	160,584.66
MPI time	28,675.53	14,784.99	13,596.69	13,935.98
Collective	0.96	1.92	2.35	2.07
Wait at Barrier	0.93	1.88	2.32	2.03
RMA Synchronization	27,884.91	14,137.09	12,954.35	13,216.67
Wait at Fence	14,462.19	473.45	–	–
Early Wait	–	2,795.67	2,777.35	2,927.43
Late Complete	–	16.48	17.16	16.53
Late Post	–	404.98	400.50	441.50
RMA Communication	726.62	572.44	567.98	605.03
Late Post	–	–	–	–

Metric [occ.]	fence-only	gats-fence	gats-single	gats-multi
Pair-wise sync.	$9.844{\cdot}10^{10}$	$6.241{\cdot}10^{8}$	$9.871{\cdot}10^{7}$	$9.871{\cdot}10^{7}$
Unneeded pair-wise sync.	$9.832{\cdot}10^{10}$	$5.253{\cdot}10^{8}$	–	–

A.3. CGPOP

Table A.8.: Performance metrics of the isolated call-tree of `solver.esolver` in the CGPOP benchmark with the 180×120 input tiles on 60 cores of the RWTH cluster.

Metric [s]	original	modified
Total time	125.21	108.26
MPI time	59.78	34.28
Synchronization	56.20	34.28
Collective	52.98	–
Wait at Barrier	51.95	–
Completion	0.87	–
RMA Synchronization	3.23	34.28
Late Post	1.56	26.21

A.4. SRUMMA

Table A.9.: Execution times of the parallel analysis of the SRUMMA benchmark on scales from 128 to 32,768 processes on the IBM Blue Gene/Q system JUQUEEN at Forschungszentrum Jülich.

Processes	Total [s]	Phase [s]					Gets/Proc.
		1	2	3	4	5	
128	0.36	0.14	0.02	0.02	0.16	0.02	112
256	0.68	0.23	0.03	0.02	0.38	0.03	128
512	1.18	0.36	0.05	0.03	0.70	0.04	224
1,024	1.58	0.46	0.06	0.03	0.98	0.05	256
2,048	3.31	0.80	0.16	0.04	2.25	0.07	448
4,096	4.24	1.01	0.17	0.04	2.93	0.09	512
8,192	9.54	2.05	0.38	0.05	6.91	0.14	896
16,384	12.57	2.49	0.46	0.06	9.40	0.16	1,024
32,768	27.82	5.28	1.10	0.08	21.11	0.25	1,792

Table A.10.: Selected performance metric from measurement runs of the ARMCI SRUMMA matrix-multiply benchmark across different scales, ranging from 128 to 32,768 processes.

Processes	Total [s]	User [s]	MPI [s]	MPI Sync. [s]	Wait at Barrier [s]
128	$4.583 \cdot 10^{4}$	$1.517 \cdot 10^{4}$	$1.152 \cdot 10^{3}$	$1.148 \cdot 10^{3}$	$1.148 \cdot 10^{3}$
256	$2.750 \cdot 10^{4}$	$1.530 \cdot 10^{4}$	$2.796 \cdot 10^{2}$	$2.514 \cdot 10^{2}$	$2.514 \cdot 10^{2}$
512	$3.193 \cdot 10^{4}$	$1.551 \cdot 10^{4}$	$7.736 \cdot 10^{2}$	$6.898 \cdot 10^{2}$	$6.897 \cdot 10^{2}$
1,024	$3.659 \cdot 10^{4}$	$1.595 \cdot 10^{4}$	$1.532 \cdot 10^{2}$	$8.686 \cdot 10^{1}$	$8.669 \cdot 10^{1}$
2,048	$4.656 \cdot 10^{4}$	$1.694 \cdot 10^{4}$	$2.495 \cdot 10^{3}$	$6.159 \cdot 10^{2}$	$6.156 \cdot 10^{2}$
4,096	$6.235 \cdot 10^{4}$	$1.870 \cdot 10^{4}$	$8.816 \cdot 10^{3}$	$3.288 \cdot 10^{3}$	$3.287 \cdot 10^{3}$
8,192	$1.072 \cdot 10^{5}$	$2.209 \cdot 10^{4}$	$2.129 \cdot 10^{4}$	$1.516 \cdot 10^{4}$	$1.516 \cdot 10^{4}$
16,384	$2.474 \cdot 10^{5}$	$2.874 \cdot 10^{4}$	$8.088 \cdot 10^{4}$	$6.244 \cdot 10^{4}$	$6.244 \cdot 10^{4}$
32,768	$9.474 \cdot 10^{5}$	$4.458 \cdot 10^{4}$	$4.195 \cdot 10^{5}$	$2.702 \cdot 10^{5}$	$2.701 \cdot 10^{5}$

Processes	MPI Comm. [s]	Late Sender [s]	ARMCI [s]	ARMCI Sync. [s]
128	$3.324 \cdot 10^{-1}$	$2.836 \cdot 10^{-1}$	$2.951 \cdot 10^{4}$	$1.239 \cdot 10^{4}$
256	$8.333 \cdot 10^{-1}$	$7.291 \cdot 10^{-1}$	$1.192 \cdot 10^{4}$	$3.660 \cdot 10^{3}$
512	$2.608 \cdot 10^{0}$	$2.262 \cdot 10^{0}$	$1.564 \cdot 10^{4}$	$4.630 \cdot 10^{3}$
1,024	$8.117 \cdot 10^{0}$	$7.192 \cdot 10^{0}$	$2.048 \cdot 10^{4}$	$5.339 \cdot 10^{3}$
2,048	$2.543 \cdot 10^{1}$	$2.306 \cdot 10^{1}$	$2.712 \cdot 10^{4}$	$7.422 \cdot 10^{3}$
4,096	$1.082 \cdot 10^{2}$	$9.911 \cdot 10^{1}$	$3.484 \cdot 10^{4}$	$1.103 \cdot 10^{4}$
8,192	$5.662 \cdot 10^{2}$	$5.179 \cdot 10^{2}$	$6.387 \cdot 10^{4}$	$2.796 \cdot 10^{4}$
16,384	$2.425 \cdot 10^{3}$	$2.251 \cdot 10^{3}$	$1.378 \cdot 10^{5}$	$9.081 \cdot 10^{4}$
32,768	$9.905 \cdot 10^{3}$	$8.814 \cdot 10^{3}$	$4.833 \cdot 10^{5}$	$3.917 \cdot 10^{5}$

Processes	ARMCI Comm. [s]	Wait for Progress (Comm.) [s]
128	$1.712 \cdot 10^{4}$	$1.696 \cdot 10^{4}$
256	$8.256 \cdot 10^{3}$	$7.937 \cdot 10^{3}$
512	$1.100 \cdot 10^{4}$	$1.042 \cdot 10^{4}$
1,024	$1.511 \cdot 10^{4}$	$1.439 \cdot 10^{4}$
2,048	$1.961 \cdot 10^{4}$	$1.843 \cdot 10^{4}$
4,096	$2.342 \cdot 10^{4}$	$2.196 \cdot 10^{4}$
8,192	$3.398 \cdot 10^{4}$	$2.999 \cdot 10^{4}$
16,384	$3.854 \cdot 10^{4}$	$3.237 \cdot 10^{4}$
32,768	$5.678 \cdot 10^{4}$	$4.360 \cdot 10^{4}$

Definitions

General Terms and Definitions

Inefficiency Patterns in One-sided Communication

Statement of publications

Parts of this thesis were previously published in the following works. My personal contribution to these publications is noted below the corresponding publication.

[1] Marc-André Hermanns, Markus Geimer, Bernd Mohr, and Felix Wolf. Scalable detection of MPI-2 remote memory access inefficiency patterns. In *Proc. of the 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Espoo, Finland*, volume 5759 of *Lecture Notes in Computer Science*, pages 31–41. Springer, September-October 2009. Outstanding Paper Award.

> Contribution: In this paper, I took the lead in developing the wait-state analysis for active-target synchronization. Markus Geimer, as the lead developer of the replay infrastructure, consulted on its use. He also transformed the existing BT benchmark of the NAS Parallel Benchmark Suite 2.4 to use one-sided communication with different active-target synchronization schemes. I modified an existing SOR kernel benchmark to use one-sided communication. Together we performed the measurements for the publication and discussed possible optimizations for our implementations. Bernd Mohr and Felix Wolf accompanied the work with suggestions for improvements. Markus Geimer and I wrote a first draft of the paper that was subsequently revised by all authors.

[2] Marc-André Hermanns, Markus Geimer, Bernd Mohr, and Felix Wolf. Scalable detection of MPI-2 remote memory access inefficiency patterns. *Intl. Journal of High Performance Computing Applications (IJHPCA)*, 26(3):227–236, August 2012.

> Contribution: Publication [1] was elected as one of the best for the EuroPVM/MPI 2009 conference and this publication is an invited extension of it. Based on discussions with other experts on one-sided communication at the conference, I developed the new performance metrics for *Pair-wise synchronizations for RMA* and *Unneeded pair-wise synchronization for RMA* presented in this paper. Markus Geimer and I updated and extended the measurements. All authors contributed to the extension and revision of the text.

[3] Marc-André Hermanns, Markus Geimer, Bernd Mohr, and Felix Wolf. *Trace-Based Detection of Lock Contention in MPI One-Sided Communication*, pages 97–114. Springer International Publishing, Cham, Oct 2017.

> Contribution: For this publication, I took the lead in designing and implementing the detection algorithm for lock contention based on the active-message infrastructure. All authors contributed to the writing of the publication.

[4] Marc-André Hermanns, Sriram Krishnamoorthy, and Felix Wolf. A scalable infrastructure for the performance analysis of passive target synchronization. *Parallel Computing*, 39(3):132 – 145, March 2013.

> Contribution: In this publication, I took the lead in developing the detection algorithm for wait states due to insufficient remote progress in passive-target synchronization. The wait-state definition was the direct result of discussions between Sriram Krishnamoorty and me about the implementation of ARMCI as a backend for Global Arrays. Sriram Krishnamoorthy and myself performed and evaluated the measurements. Sriram Krishnamoorthy and I wrote a first draft of the paper. Felix Wolf supported subsequent revisions.

[5] Marc-André Hermanns, Manfred Miklosch, David Böhme, and Felix Wolf. Understanding the formation of wait states in applications with one-sided communication. In *EuroMPI '13: Proc. of the 20th European MPI Users' Group Meeting, Madrid, Spain, September 15–18, 2013*, pages 73–78, New York, NY, USA, September 2013. ACM.

> Contribution: For this publication, I took the lead in implementing the root-cause analysis for synchronization-based wait states. Manfred Miklosch implemented the necessary infrastructure during his Master's thesis, prior to this publication. David Böhme, the original designer of the replay-based approach, consulted the implementation. All authors but Manfred Miklosch, who at the time already finished his thesis and left the institute, contributed to the writing of the paper.

[6] Andreas Knüpfer, Robert Dietrich, Jens Doleschal, Markus Geimer, Marc-André Hermanns, Christian Rössel, Ronny Tschüter, Bert Wesarg, and Felix Wolf. Generic support for remote memory access operations in Score-P and OTF2. In *Proc. of the 6th Parallel Tools Workshop, Stuttgart, September 2012*, pages 57–74. Springer Berlin Heidelberg, May 2013.

> Contribution: Andreas Knüpfer took the lead for this invited publication to the Parallel Tools Workshop conference 2012. Jens Doleschal and myself took the lead in the definition of generic event records in OTF2 to describe the communication of various one-sided communication interfaces, such as MPI, ARMCI, and SHMEM. I incorporated my experiences gained with the implementation of a similar event model in Scalasca's internal measurement system for MPI, ARMCI, and SHMEM. The discussions between me and Jens Doleschal lead to the initial definition of the OTF2 event model. This model was subsequently revised by additional input from the OTF2 core developers, ensuring that the event model also fits other one-sided-like APIs such as memory copies in NVIDIA's Cuda. All authors contributed to the writing of the publication.

Bibliography

[7] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, John M. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[8] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: incorporating long messages into the LogP model–one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[9] George Almási, Paul Hargrove, Ilie Gabriel Tănase, and Yili Zheng. UPC collectives library 2.0. In Chapman et al. [38].

[10] Wolfgang Arden, Brillouët, Patrick Cogez, Mart Greaf, Bert Huizing, and Reinhard Mahnkopf. More than Moore. White paper, ITRS, 2010.

[11] David Bailey, E. Barszcz, J. Barton, D. Browning, Carter R., L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, Horst Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Advanced Supercomputing Devision, March 1994.

[12] Rajkishore Barik, Zoran Budimlic, Vincent Cavé, Sanjay Chatterjee, Yi Guo, David M. Peixotto, Raghavan Raman, Jun Shirako, Sagnak Tasirlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The Habanero multicore software research project. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 735–736. ACM, 2009.

[13] Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John Linford. Replay-based synchronization of timestamps in event traces of massively parallel applications. *Scalable Computing: Practice and Experience*, 10(1):49–60, March 2009. Special Issue: *Simulation in Emergent Computational Systems*.

[14] M Behr, D Arora, OM Coronado, and M Pasquali. Models and finite element techniques for blood flow simulation. *International Journal of Computational Fluid Dynamics*, 20(3-4):175–181, 2006.

[15] Robert Bell, AllenD. Malony, and Sameer Shende. ParaProf: A portable, extensible, and scalable tool for parallel performance profile analysis. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 17–26. Springer Berlin Heidelberg, 2003.

[16] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[17] James C. Beyer, Eric J. Stotzer, Alistair Hart, and Bronis R.R. de Supinski. OpenMP for accelerators. In Barbara M. Chapman, William D. Gropp, Kalyan Kumaran, and Matthias S. Müller, editors, *OpenMP in the Petascale Era*, volume 6665 of *Lecture Notes in Computer Science*, pages 108–121. Springer Berlin Heidelberg, 2011.

[18] Scott Biersdorff, Chee Wai Lee, Allen D. Malony, and Laxmikant V. Kalé. Integrated performance views in Charm++: Projections meets TAU. *Parallel Processing, International Conference on*, 0:140–147, 2009.

[19] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987.

[20] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[21] Robert Bocchino, David Callahan, Bradford Lee Chamberlain, Sung-Eun Choi, Steven J. Deitz, R. E. Diaconescu, James Dinan, Samuel Figueroa, Shannon Hoffswell, Mary Beth Hribar, David Iten, Mark James, Mackale Joyner, Jacob Nelson, John Plevjak, Lee Prokowich, Albert Sidelnik, Greg Titus, Wayne Wong, and Hans P. Zima. Chapel language specification 0.91. Technical report, Cray Inc., April 2012.

[22] Barry W. Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24, August 1986.

[23] David Boehme. *Characterizing Load and Communication Imbalance in Large-Scale Parallel Applications*. PhD thesis, RWTH Aachen University, 2013.

[24] David Böhme, Bronis R. de Supinski, Markus Geimer, Martin Schulz, and Felix Wolf. Scalable critical-path based performance analysis. In *Proc. of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China*. IEEE Computer Society, May 2012.

[25] David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. Identifying the Root Causes of Wait States in Large-Scale Parallel Applications. *ACM Transactions on Parallel Computing*, 3(2):1–24, jul 2016.

[26] David Böhme, Markus Geimer, Felix Wolf, and Lukas Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *Proc. of the 39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA*, pages 90–100. IEEE Computer Society, September 2010. Best Paper Award.

[27] Dan Bonachea. GASNet Specification, v1.1. Technical report, University of California, Berkeley, October 2002.

[28] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1-3):91–99, 2004.

[29] Zoran Budimlic, Vincent Cavé, Raghavan Raman, Jun Shirako, Sagnak Tasirlar, Jisheng Zhao, and Vivek Sarkar. The design and implementation of the Habanero-Java parallel programming language. In Cristina Videira Lopes and Kathleen Fisher, editors, *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 185–186. ACM, 2011.

[30] Alexandru Calotoiu, Christian Siebert, and Felix Wolf. Pattern-independent detection of manual collectives in MPI programs. In *Proc. of the 18th Euro-Par Conference, Rhodes Island, Greece*, volume 7484 of *Lecture Notes in Computer Science*, pages 28–39. Springer, August 2012.

[31] CAPS. HMPP directives: HMPP workbench 3.2. User manual, CAPS, 2012.

[32] William Carlson, Jesse Draper, David Culler, Katherine Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, 17100 Science Dr., Bowie, MD 20715, May 1999.

[33] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[34] Bradford Lee Chamberlain, David Callahan, and Hans P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[35] Anthony Chan, William Gropp, and Ewing Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2-3):155–165, 2008.

[36] Barbara Chapman, Anthony Curtis, Ricardo Mauricio, Swaroop Pophale, Ram Nanjegowda, Amrita Banerjee, Karl Feind, Jeffery A. Kuehn, Stephen Poole, and Lauren Smith. *OpenSHMEM Application Programming Interface*, 1.1 edition, June 2014.

[37] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[38] Barbara Chapman, Vivek Sarkar, and John Mellor-Crummey, editors. *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models*. ACM, oct 2011.

[39] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 519–538. ACM, 2005.

[40] Dong Chen, Noel Eisley, Philip Heidelberger, Sameer Kumar, Amith Mamidala, Fabrizio Petrini, Robert Senger, Yutaka Sugawara, Robert Walkup, Burkhard Steinmacher-Burow, Anamitra Choudhury, Yogish Sabharwal, Swati Singhal, and Jeffrey J. Parker. Looking under the hood of the IBM Blue Gene/Q network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 69:1–69:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[41] Guogjing Cong, I-Hsin Chung, Huifang Wen, David Klepacki, Hiroki Murata, Yasushi Negishi, and Takao Moriyama. A holistic approach towards automated performance analysis and tuning. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 33–44. Springer Berlin Heidelberg, 2009.

[42] Guojing Cong, Hui-fang Wen, Yasushi Negishi, and Hiroki Murata. Tool-assisted performance measurement and tuning of UPC applications. In Chapman et al. [38].

[43] Cray. Man page collection: Shared memory access (SHMEM). Technical Report S-2383-23, Cray Inc., December 2003.

[44] Cray Inc. Using Cray performance measurement and analysis tools. User Manual S–2376–610, Cray Inc., March 2013.

[45] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. Technical report, University of California Berkeley, Berkeley, CA, USA, 1992.

[46] P.J. Denning. Working sets past and present. *Software Engineering, IEEE Transactions on*, SE-6(1):64–84, 1980.

[47] Luiz DeRose, Bill Homer, and Dean Johnson. Detecting application load imbalance on high end massively parallel systems. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 150–159. Springer Berlin / Heidelberg, 2007.

[48] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[49] James Dinan, Pavan Balaji, Jeff R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. Supporting the Global Arrays PGAS model using MPI one-sided communication. In *IPDPS*, pages 739–750. IEEE Computer Society, 2012.

[50] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM.

[51] Isaac Dooley, Chee Wai Lee, and Laxmikant Kale. Continuous performance monitoring for large-scale parallel applications. In *16th annual IEEE International Conference on High Performance Computing (HiPC 2009)*, December 2009.

[52] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In *Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium, August 30 – September 2 2011*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, 2012.

[53] R. J. Harrison et al. Toward high-performance computational chemistry: II. a scalable self-consistent field program. *Journal of Computational Chemistry*, 17(1):124–132, 1996.

[54] Karl-Filip Faxén and John Ardelius. Manycore work stealing. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 10:1–10:2, New York, NY, USA, 2011. ACM.

[55] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[56] Markus Geimer, Felix Wolf, Andreas Knüpfer, Bernd Mohr, and Brian J. N. Wylie. A parallel trace-data interface for scalable performance analysis. In *Proc. of the 8th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), Umeå, Sweden*, volume 4699 of *Lecture Notes in Computer Science*, pages 398–408. Springer, June 2006.

[57] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[58] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. volume 4192 of *Lecture Notes in Computer Science*, pages 303–312, Heidelberg, Germany, 2006. Springer.

[59] Michael Gerndt and Michael Ott. Automatic performance analysis with Periscope. *Concurrency and Computation: Practice and Experience*, 22(6):736–748, 2010.

[60] Richard L. Graham, Steve Poole, Pavel Shamis, Gil Bloch, Noam Bloch, Hillel Chapman, Michael Kagan, Ariel Shahar, Ishai Rabinovitz, and Gilad Shainer. ConnectX-2 infiniband management queues: First investigation of the new support for network offloaded collective operations. In *CCGRID*, pages 53–62. IEEE, 2010.

[61] Jeff R. Hammond, James Dinan, Pavan Balaji, Ivo Kabadshow, Sreeram Potluri, and Vinod Tipparaju. OSPRI: An optimized one-sided communication runtime for leadership-class machines. In *Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models*, 2012.

[62] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012.

[63] Marc-André Hermanns. Ereignisbasierte Leistungsanalyse von Remote-Memory-Access-Operationen. Technical Report ZAM-IB-2004-15, Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik, Jülich, 2005.

[64] Marc-André Hermanns, Markus Geimer, Felix Wolf, and Brian J. N. Wylie. Verifying causality between distant performance phenomena in large-scale MPI applications. In *Proc. of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Weimar, Germany*, pages 78–84. IEEE Computer Society, February 2009.

[65] Marc-André Hermanns, Bernd Mohr, and Felix Wolf. Event-based measurement and analysis of one-sided communication. In *Proc. of the 11th Euro-Par Conference, Lisboa, Portugal*, volume 3648 of *Lecture Notes in Computer Science*, pages 156–165. Springer, August-September 2005.

[66] G. Hermannsson and L. Wittie. Fast locks in distributed shared memory systems. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 1, pages 574 –583, jan. 1994.

[67] Kevin A. Huck and Allen D. Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 41–, Washington, DC, USA, 2005. IEEE Computer Society.

[68] Kevin A. Huck, Allen D. Malony, Sameer Shende, and Alan Morris. Knowledge support and automation for performance analysis with PerfExplorer 2.0. *Sci. Program.*, 16(2–3):123–134, April 2008.

[69] HyperTransport Consortium. HyperTransport I/O technology overview. White Paper, June 2004.

[70] IBM. *IBM High Productivity Computing Systems Toolkit*. IBM T.J. Watson Research Center, September 2009.

[71] Intel Corp. An introduction to the Intel QuickPath interconnect. White Paper, Jan 2009.

[72] Intel Corp. Intel trace analyzer and collector, 2012.

[73] Intel Corp. Intel VTune Amplifier XE, 2012.

[74] Intel Corp. The Intel Xeon Phi product family. White Paper, 2013.

[75] Intel Corporation. Structured Tracefile Format. Technical Report March, Intel Corp., 2015.

[76] Laxmikant V. Kalé and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. In Timlynn Babitsky and Jim Salmons, editors, *OOPSLA*, pages 91–108. ACM, 1993.

[77] James E. Kelley, Jr and Morgan R. Walker. Critical-path planning and scheduling. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 160–173, New York, NY, USA, 1959. ACM.

[78] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. OpenCL as a unified programming model for heterogeneous CPU/GPU clusters. *SIGPLAN Not.*, 47(8):299–300, February 2012.

[79] Wooyoung Kim and M. Voss. Multicore desktop programming with intel threading building blocks. *Software, IEEE*, 28(1):23–31, 2011.

[80] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the open trace format (otf). In *Proceedings of the 6th international conference on Computational Science - Volume Part II*, ICCS'06, pages 526–533, Berlin, Heidelberg, 2006. Springer-Verlag.

[81] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of 5th Parallel Tools Workshop, Dresden, Germany*, September 2011.

[82] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Physical Review*, 140(4A):A1133–A1138, 1965.

[83] Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, Johannes Kepler Universität Linz, September 2000.

[84] Manojkumar Krishnan and Jarek Nieplocha. SRUMMA: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In *IPDPS*, page 70, 2004.

[85] Rick Kufrin. PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux. In *Proc. of the 6th International Conference on Linux Clusters: The HPC Revolution*, April 2005.

[86] Andrej Kühnal, Marc-André Hermanns, Bernd Mohr, and Felix Wolf. Specification of inefficiency patterns for MPI-2 one-sided communication. In *Proc. of the 12th Euro-Par Conference, Dresden, Germany*, volume 4128 of *Lecture Notes in Computer Science*, pages 47–62. Springer, August - September 2006.

[87] Sameer Kumar, Gabor Dozsa, George Almási, Philip Heidelberger, Dong Chen, Mark E. Giampapa, Blockso Michael, Ahmad Faraj, Jeff Parker, Joseph Ratterman, Brian Smith, and Charles J. Archer. The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 94–103, New York, NY, USA, 2008. ACM.

[88] Jesus Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. Dip: A parallel program development environment. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 665–674. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0024763.

[89] Robert Latham, Robert Ross, and Rajeev Thakur. Implementing MPI-IO atomic mode and shared file pointers using mpi one-sided communication. *Int. J. High Perform. Comput. Appl.*, 21(2):132–143, 2007.

[90] Jinpil Lee and Mitsuhisa Sato. Implementation and performance evaluation of XcalableMP: A parallel programming language for distributed memory systems. In Wang-Chien Lee and Xin Yuan, editors, *39th International Conference on Parallel Processing, ICPP Workshops 2010, San Diego, California, USA, 13-16 September 2010*, pages 413–420. IEEE Computer Society, 2010.

[91] Charles E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*, pages 522–527. ACM, 2009.

[92] Adam Leko, Dan Bonachea, Hung-Hsun Su, and Alan D. George. GASP: A performance analysis tool interface for global address space programming models. Technical Report LBNL-61606, Lawrence Berkeley National Lab, September 2006. Version 1.5 (09/14/2006).

[93] Guoyong Mao, David Böhme, Marc-André Hermanns, Markus Geimer, Daniel Lorenz, and Felix Wolf. Catching idlers with ease: A lightweight wait-state profiler for MPI programs. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 103:103–103:108, New York, NY, USA, 2014. ACM.

[94] Wagner Meira, Jr., Thomas J. LeBlanc, and Virgílio A. F. Almeida. Using cause-effect analysis to understand the performance of distributed programs. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '98, pages 101–111, New York, NY, USA, 1998. ACM.

[95] Mitesh R. Meswani, Laura Carrington, Allan Snavely, and Stephen W. Poole. Tools for benchmarking, tracing, and simulating SHMEM applications. In *Geengineering the Future, Proceedings of the 54th Cray User Group Meeting, Stuttgart, Germany*. Cray Inc., 2012.

[96] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. The Top500 list. Electronically published at http://www.top500.org/lists/2015/06, June 2015.

[97] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28:37–46, November 1995.

[98] Seung-jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In Chapman et al. [38].

[99] Bernd Mohr, Andrej Kühnal, Marc-André Hermanns, and Felix Wolf. Performance analysis of one-sided communication mechanisms. In *Proc. of the Conference on Parallel Computing (ParCo), Malaga, Spain*, September 2005. *Minisymposium Performance Analysis.*

[100] Bernd Mohr, Allen D. Malony, Sameer S. Shende, and Felix Wolf. Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In *3rd European Workshop on OpenMP (EWOMP), Barcelona, Spain*, September 2001.

[101] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[102] Csaba Andras Moritz and Matthew I. Frank. LoGPC: modeling network contention in message-passing programs. *SIGMETRICS Perform. Eval. Rev.*, 26(1):254–263, June 1998.

[103] MPI Forum, editor. *MPI-2: Extensions to the Message-Passing Interface*, chapter 6: One-sided communication, pages 109–144. MPI Forum, July 17th 1997.

[104] MPI Forum, editor. *MPI: A Message-Passing Interface Standard. Version 3.0.* MPI Forum, September 2012.

[105] MPI Forum, editor. *MPI: A Message-Passing Interface Standard. Version 3.0*, chapter 5: Collective Communication, pages 141–222. In MPI Forum [104], September 2012.

[106] MPI Forum, editor. *MPI: A Message-Passing Interface Standard. Version 3.0*, chapter 11: One-Sided Communication, pages 403–472. In MPI Forum [104], September 2012. Chapter edited by W. Gropp and T. Hoefler and R. Thakur.

[107] MPI Forum, editor. *MPI: A Message-Passing Interface Standard. Version 3.0*, chapter 14: Tool Support, pages 555–590. In MPI Forum [104], September 2012.

[108] *MPI Peruse 2.0 - A Performance Revealing Extensions Interface to MPI*, March 2006.

[109] Frank Mueller, Xing Wu, Martin Schulz, Bronis R. de Supinski, and Todd Gamblin. ScalaTrace: Tracing, analysis and modeling of HPC codes at scale. In Kristján Jónasson, editor, *PARA (2)*, volume 7134 of *Lecture Notes in Computer Science*, pages 410–418. Springer, 2010.

[110] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing scalable applications with Vampir, VampirServer and VampirTrace. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 637–644, Jülich/Aachen, Germany, September 4–7, 2007 2007. Forschungszentrum Jülich and RWTH Aachen University, IOS Press.

[111] Jan Mußler, Daniel Lorenz, and Felix Wolf. Reducing the overhead of direct application instrumentation using prior static analysis. In *Proc. of the 17th Euro-Par Conference, Bordeaux, France*, volume 6852 of *Lecture Notes in Computer Science*, pages 65–76. Springer, September 2011.

[112] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12:69–80, 1996.

[113] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Bokut, and Mitsuhisa Sato. XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 27–36, Piscataway, NJ, USA, 2014. IEEE Press.

[114] Girija J. Narlikar and Guy E. Blelloch. Pthreads for dynamic and irregular parallelism. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–16, Washington, DC, USA, 1998. IEEE Computer Society.

[115] Sarah Neuwirth, Dirk Frey, Mondrian Nuessle, and Ulrich Bruening. Scalable communication architecture for network-attached accelerators. In *2015 IEEE 21st Int. Symp. High Perform. Comput. Archit. HPCA 2015*, pages 627–638, 2015.

[116] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 16:1–16:14, New York, NY, USA, 2008. ACM.

[117] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The ARMCI approach. *Int. J. High Perform. Comput. Appl.*, 20:233–253, May 2006.

[118] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy libray for ditributed array libraries and compiler run-time systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 533–546, London, UK, UK, 1999. Springer-Verlag.

[119] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: a nonuniform memory access programming model for high-performance computers. *J. Supercomput.*, 10:169–189, June 1996.

[120] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *J. Parallel Distrib. Comput.*, 69(8):696–710, 2009.

[121] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998.

[122] NVIDIA Corp. NVIDIA's next generation CUDA compute architecture: Fermi. White paper, 2009.

[123] NVIDIA Corp. CUDA parallel computing platform. `http://www.nvidia.com/object/cuda_home_new.html`, 2012.

[124] OpenACC Consortium. *The OpenACC Application Programming Interface*, 2013.

[125] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 4.0 edition, July 2013.

[126] Robert Palumbo, Hossein Saiedian, and Mansour Zand. The operational semantics of an active message system. In *CSC '92: Proceedings of the 1992 ACM annual conference on Communications*, pages 367–375, New York, NY, USA, 1992. ACM.

[127] R. G. Parr and W. Yang. *Density-Functional Theory of Atoms and Molecules*. Oxford University Press, Inc., New York, 1989.

[128] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[129] J. P. Perdew and K. Schmidt. Jacob's ladder of density functional approximations for the exchange-correlation energy. *AIP Conference Proceedings*, 577(1):1–20, 2001.

[130] Vincent Pillet, Jesús Labarta, Toni Cortés, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. *Transputer and occam Developments*, pages 17–32, April 1995. `http://www.bsc.es/paraver`.

[131] Stephen W. Poole, Oscar Hernandez, Jeffery A. Kuehn, Galen M. Shipman, Anthony Curtis, and Karl Feind. OpenSHMEM - toward a unified RMA model. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1379–1391. Springer, 2011.

[132] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D.K. Panda. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 80–89, October 2013.

[133] Keith H. Randall. *CILK: Efficient Multithreaded Computing*. PhD thesis, Massachusetts Institute of Technology, 1998.

[134] John Reid. Coarrays in the next fortran standard. *SIGPLAN Fortran Forum*, 29:10–27, July 2010.

[135] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. Mrnet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 21–, New York, NY, USA, 2003. ACM.

[136] Einar Rustad. A high level technical overview of the NumaConnect technology and products. Technical report, numaScale, 2007.

[137] Vijay Saraswat, Bard Bloom, gor Peshansky, Olivier Tardieu, and David Grove. X10 2.5. Language specification, IBM, 2014.

[138] William B. Sawyer and Arthur A. Mirin. The implementation of the finite-volume dynamical core in the community atmosphere model. *Journal of Computational and Applied Mathematics*, 203(2):387–396, 2007.

[139] S. Sbaraglia, H. Wen, S. Seelam, I. Chung, G. Cong, K. Ekanadham, and D. Klepacki. A productivity centered application performance tuning framework. In *Proceedings of the 2Nd International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '07, pages 49:1–49:10, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[140] Dirk Schmidl, Tim Cramer, Christian Terboven, Dieter an Mey, and Matthias S. Müller. An OpenMP extension library for memory affinity. In Luiz DeRose, Bronis R. de Supinski, Stephen L. Olivier, Barbara M. Chapman, and Matthias S. Müller, editors, *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings*, volume 8766 of *Lecture Notes in Computer Science*, pages 103–114. Springer, 2014.

[141] Lars Schneidenbach, David Böhme, and Bettina Schnor. Performance issues of synchronisation in the mpi-2 one-sided communication api. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 177–184. Springer Berlin Heidelberg, 2008.

[142] Martin Schulz, Greg Bronevetsky, and Bronis R. Supinski. On the performance of transparent MPI piggyback messages. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 194–201, Berlin, Heidelberg, 2008. Springer-Verlag.

[143] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. Open|SpeedShop: An open source infrastructure for parallel performance analysis. *Sci. Program.*, 16(2-3):105–121, April 2008.

[144] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, Christian Trott, Greg Scantlen, and Paul S. Crozier. The development of Mellanox/NVIDIA GPUDirect over InfiniBand - a new model for GPU to GPU communications over InfiniBand - a new model for GPU to GPU communications. *Computer Science - R&D*, 26(3-4):267–273, 2011.

[145] Sameer S. Shende and Allen D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[146] Christian Siebert and Jesper Larsson Träff. Efficient MPI implementation of a parallel, stable merge algorithm. In *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 204–213. Springer Berlin / Heidelberg, September 2012.

[147] Fengguang Song, Felix Wolf, Nikhil Bhatia, Jack Dongarra, and Shirley Moore. An algebra for cross-experiment performance analysis. In *Proc. of the International Conference on Parallel Processing (ICPP), Montreal, Canada*, pages 63–72. IEEE Society, August 2004.

[148] R. Speck, L. Arnold, and P. Gibbon. Towards a petascale tree code: Scaling and efficiency of the {PEPC} library. *Journal of Computational Science*, 2(2):138–143, 2011. Simulation Software for Supercomputers.

[149] Andrew Stone, John Dennis, and Michelle Mills Strout. The CGPOP miniapp, version 1.0. Technical Report CS-11-103, Colorado State University, July 2011.

[150] Hung-Hsun Su. *Parallel Performance Wizard: Framework and Techniques for Parallel Application Optimization.* PhD thesis, University of Florida, 2010.

[151] Hung-Hsun Su, Max Billingsley, and Alan D. George. Parallel Performance Wizard: A performance system for the analysis of partitioned global-address-space applications. *Int. J. High Perform. Comput. Appl.*, 24:485–510, November 2010.

[152] Hung-Hsun Su, Dan Bonachea, Adam Leko, Hans Sherburne, Max Billingsley, III., and Alan D. George. Gasp! a standardized performance analysis tool interface for global address space programming models. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors, *PARA'06: Proceedings of the 8th international conference on Applied parallel computing*, pages 450–459, Berlin, Heidelberg, 2007. Springer-Verlag.

[153] G. Sutmann, L. Westphal, and M. Bolten. Particle based simulations of complex systems with MP2C: Hydrodynamics and electrostatics. In *ICNAAM 2010: International Conference of Numerical Analysis and Applied Mathematics 2010, AIP Conference Proceedings, Vol. 1281, Issue 1, 2010. - 978-0-7354-0834-0. - S. 1768 - 1772*, 2010. Record converted from VDB: 12.11.2012.

[154] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[155] Zoltán Szebenyi, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Felix Wolf, and Brian J. N. Wylie. Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs. In *Proc. of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Anchorage, AK, USA*, pages 640–648. IEEE Computer Society, May 2011.

[156] Nathan R. Tallent, Laksono Adhianto, and John Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *Supercomputing 2010*, New Orleans, LA, USA, November 2010.

[157] Nathan R. Tallent, John M. Mellor-Crummey, Laksono Adhianto, Michael W. Fagan, and Mark Krentel. Diagnosing performance bottlenecks in emerging petascale applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 51:1–51:11, New York, NY, USA, 2009. ACM.

[158] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. *SIGPLAN Not.*, 45(5):269–280, January 2010.

[159] N.R. Tallent and J.M. Mellor-Crummey. Identifying performance bottlenecks in work-stealing computations. *Computer*, 42(12):44–50, 2009.

[160] Monika ten Bruggencate and Duncan Roweth. DMAPP - an API for one-sided program models on Baker systems. In *Cray User Group Meeting (CUG 2010), Edinburgh*, 2010.

[161] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *IJHPCA*, 19(1):49–66, 2005.

[162] C. Thiffault, M. Voss, S.T. Healey, and Seon Wook Kim. Dynamic instrumentation of large-scale mpi and openmp applications. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9 pp.–, April 2003.

[163] Mustafa M. Tikir, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. PSINS: An open source event tracer and execution simulator for MPI applications. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, August 2009.

[164] University Corporation for Atmospheric Research (UCAR). The Community Earth System Model, Februar 2012.

[165] UPC Consortium. *UPC Language Specifications*, 1.3 edition, November 2013.

[166] M. Valiev et al. NWChem: A comprehensive and scalable open-source solution for large-scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[167] Michael L. Van De Vanter, Douglass E. Post, and Mary E. Zosel. Hpc needs a tool strategy. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 55–59, New York, NY, USA, 2005. ACM.

[168] Jefferey Vetter and Chris Chambreau. mpiP: Lightweight, scalable MPI profiling.

[169] Jeffrey Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *Proceedings of the 14th international conference on Supercomputing*, ICS '00, pages 245–254, New York, NY, USA, 2000. ACM.

[170] A. Vishnu, M. ten Bruggencate, and R. Olson. Evaluating the potential of Cray Gemini interconnect for PGAS communication runtime systems. In *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on*, pages 70 –77, aug. 2011.

[171] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. *SIGARCH Comput. Archit. News*, 20(2):256–266, April 1992.

[172] Rob Van der Wijngaart and Haoqiang Jin. Nas parallel benchmarks, multi-zone versions. Technical Report NAS-03-010, NASA Advanced Supercomputing Devision, July 2003.

[173] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Active pebbles: parallel programming for data-driven applications. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 235–244, New York, NY, USA, 2011. ACM.

[174] Felix Wolf and Bernd Mohr. Automatic performance analysis of MPI applications based on event traces. In *Proc. of the 6th Euro-Par Conference, Munich, Germany*, number 1900 in Lecture Notes in Computer Science, pages 123–132. Springer, August-September 2000.

[175] Felix Wolf and Bernd Mohr. KOJAK - A tool set for automatic performance analysis of parallel applications. In *Proc. of the 9th Euro-Par Conference, Klagenfurt, Austria*, volume 2790 of *Lecture Notes in Computer Science*, pages 1301–1304. Springer, August 2003. Demonstrations of Parallel and Distributed Computing.

[176] Felix Wolf, Bernd Mohr, Nikhil Bhatia, Marc-André Hermanns, Markus Geimer, Brian J. N. Wylie, David Böhme, and Alexandre Strube. EPILOG binary trace-data format. Specification 1.8X, Forschungszentrum Jülich, May 2013.

[177] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.

[178] Changil Yoon, Vikas Aggarwal, Vrishali Hajare, Alan D. George, and Max Billingsley III. Proceedings of the fifth conference on partitioned global address space programming models. In Chapman et al. [38].

[179] J Young and S Yalamanchili. Commodity Converged Fabrics for Global Address Spaces in Accelerator Clouds. In *2012 IEEE 14th Int. Conf. High Perform. Comput. Commun. 2012 IEEE 9th Int. Conf. Embed. Softw. Syst.*, pages 303–310, 2012.

[180] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *International Journal of High Performance Computing Applications*, 13(3):277–288, 1999.

[181] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine A. Yelick. UPC++: A PGAS extension for C++. In *IPDPS*, 2014.

Websites

[182] Cube.
http://scalasca.org/software/cube-4.x/.

[183] Dyninst.
http://www.dyninst.org/.

[184] GASNet.
http://gasnet.lbl.gov/.

[185] HPCTOOLKIT: tools for performance analysis of optimized parallel programs.
http://hpctoolkit.org/.

[186] IBM High Performance Computing Toolkit.
http://researcher.watson.ibm.com/researcher/view_group.php?id=2754.

[187] IBM High Productivity Computing Systems Toolkit.
http://researcher.watson.ibm.com/researcher/view_group.php?id=2753.

[188] Intel Trace Analyzer and Collector.
https://software.intel.com/en-us/intel-trace-analyzer.

[189] Intel VTune Amplifier XE.
https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[190] The MPI Parallel Environment (mpe).
http://www.mcs.anl.gov/research/projects/perfvis/software/MPE/.

[191] mpiP: Lightweight, scalable MPI profiling.
http://mpip.sourceforge.net/.

[192] Open|speedshop.
http://openspeedshop.org/.

[193] Paradyns tools project.
http://www.paradyn.org.

[194] Parallel Performance Wizard.
http://ppw.hcs.ufl.edu.

[195] Paraver.
http://www.bsc.es/computer-sciences/performance-tools/paraver/.

[196] PerfSuite: An accessible, open source performance analysis environment for Linux.
http://perfsuite.ncsa.illinois.edu/.

[197] Periscope Tuning Framework.
`http://periscope.in.tum.de/`.

[198] The Scalasca performance analysis toolset.
`http://www.scalasca.org/`.

[199] Score-P.
`http://www.vi-hps.org/projects/score-p/`.

[200] TAU – Tuning and Analysis Utilities.
`http://www.cs.uoregon.edu/research/tau/`.

[201] VAMPIR: Visualization and analysis of mpi resources.
`http://www.vampir.eu/`.

Band / Volume 22
Three-dimensional Solute Transport Modeling in Coupled Soil and Plant Root Systems
N. Schröder (2013), xii, 126 pp
ISBN: 978-3-89336-923-2
URN: urn:nbn:de:0001-2013112209

Band / Volume 23
Characterizing Load and Communication Imbalance in Parallel Applications
D. Böhme (2014), xv, 111 pp
ISBN: 978-3-89336-940-9
URN: urn:nbn:de:0001-2014012708

Band / Volume 24
Automated Optimization Methods for Scientific Workflows in e-Science Infrastructures
S. Holl (2014), xvi, 182 pp
ISBN: 978-3-89336-949-2
URN: urn:nbn:de:0001-2014022000

Band / Volume 25
Numerical simulation of gas-induced orbital decay of binary systems in young clusters
A. C. Korntreff (2014), 98 pp
ISBN: 978-3-89336-979-9
URN: urn:nbn:de:0001-2014072202

Band / Volume 26
UNICORE Summit 2014
Proceedings, 24th June 2014 | Leipzig, Germany
edited by V. Huber, R. Müller-Pfefferkorn, M. Romberg (2014), iii, 60 pp
ISBN: 978-3-95806-004-3
URN: urn:nbn:de:0001-2014111408

Band / Volume 27
Automatische Erfassung präziser Trajektorien in Personenströmen hoher Dichte
M. Boltes (2015), xii, 308 pp
ISBN: 978-3-95806-025-8
URN: urn:nbn:de:0001-2015011609

Band / Volume 28
Computational Trends in Solvation and Transport in Liquids
edited by G. Sutmann, J. Grotendorst, G. Gompper, D. Marx (2015)
ISBN: 978-3-95806-030-2
URN: urn:nbn:de:0001-2015020300

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 29
Computer simulation of pedestrian dynamics at high densities
C. Eilhardt (2015), viii, 142 pp
ISBN: 978-3-95806-032-6
URN: urn:nbn:de:0001-2015020502

Band / Volume 30
Efficient Task-Local I/O Operations of Massively Parallel Applications
W. Frings (2016), xiv, 140 pp
ISBN: 978-3-95806-152-1
URN: urn:nbn:de:0001-2016062000

Band / Volume 31
A study on buoyancy-driven flows: Using particle image velocimetry for validating the Fire Dynamics Simulator
by A. Meunders (2016), xxi, 150 pp
ISBN: 978-3-95806-173-6
URN: urn:nbn:de:0001-2016091517

Band / Volume 32
Methoden für die Bemessung der Leistungsfähigkeit multidirektional genutzter Fußverkehrsanlagen
S. Holl (2016), xii, 170 pp
ISBN: 978-3-95806-191-0
URN: urn:nbn:de:0001-2016120103

Band / Volume 33
JSC Guest Student Programme Proceedings 2016
edited by I. Kabadshow (2017), iii, 191 pp
ISBN: 978-3-95806-225-2
URN: urn:nbn:de:0001-2017032106

Band / Volume 34
Multivariate Methods for Life Safety Analysis in Case of Fire
B. Schröder (2017), x, 222 pp
ISBN: 978-3-95806-254-2
URN: urn:nbn:de:0001-2017081810

Band / Volume 35
Understanding the formation of wait states in one-sided communication
M.-A. Hermanns (2018), xiv, 144 pp
ISBN: 978-3-95806-297-9
URN: urn:nbn:de:0001-2018012504

Weitere ***Schriften des Verlags im Forschungszentrum Jülich*** unter
http://wwwzb1.fz-juelich.de/verlagextern1/index.asp

IAS Series
Band / Volume 35
ISBN 978-3-95806-297-9

Mitglied der Helmholtz-Gemeinschaft