Fachhochschule Aachen
Campus Jülich

Extending a Performance Analysis Tool to Handle MPI Message Probing

Bachelorarbeit

Katharina Haus

Fachbereich 9

Medizintechnik und Technomathematik

Studiengang Angewandte Mathematik und Informatik B.Sc.

Jülich, August 2024

Diese Arbeit ist von mir selbständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Name: Katharina Haus
Jülich, den 12.08.2024

Unterschrift des Studierenden

Diese Arbeit wurde betreut von:

1. Prüfer:	Prof. Dr. Andreas Terstegge
2. Prüfer:	Dr. Markus Geimer

In the realm of high-performance computing (HPC), exascale systems have now reached a point of practical reality and offer unprecedented computing power. As these systems become more powerful, it is crucial to ensure that users can effectively harness this power. It is essential to consider scalability and performance optimization when developing applications running on such systems to be able to fully leverage their capabilities. Performance analysis is a fundamental aspect of optimizing parallel applications. This thesis addresses a specific gap in the Scalasca performance analysis tool: MPI message probing. Message probing is useful to determine the required buffer size for a pending message. This work presents an extended event model that captures probe calls and integrates them into Scalasca's analysis framework. The enhanced tool is evaluated across various test cases, demonstrating its capability to identify inefficiencies related to MPI message probing, especially the *Late Sender* wait state. The results show that the extended analysis provides a more comprehensive insight into application behavior.

Contents

1 Introduction

In the domain of high-performance computing (HPC), efficient performance analysis is crucial for optimizing parallel applications and the assurance of their scalability on large-scale systems. HPC systems often utilize distributed memory architectures, and are typically organized into clusters, with each including several nodes. Each node contains processors and has its own memory space. A core is the smallest processing unit within a processor. Threads, handled by cores, are the smallest units of execution within a process and share resources such as memory. Nodes in such configurations do not have direct access to memory on other nodes. Communication between nodes is achieved through message passing, typically using the Message Passing Interface (MPI) [1]. Many MPI implementations support threading and facilitate hybrid programming approaches. This means MPI handles communication between processes, while Open Multi-Processing (OpenMP) manages parallel threads within a single process.

Performance analysis tools assist users in the identification of bottlenecks, the optimization of resource usage, and the enhancement of the overall application runtime. Among these tools, Scalasca [2, 3] has established itself as a powerful framework for the scalable analysis of large-scale parallel applications. Scalasca's trace analyzer allows for the automated detection of inefficiency patterns. One common inefficiency that Scalasca identifies is the *Late Sender* wait state. This occurs when a process idles in a receive operation waiting for a message from a delayed sender. Such an inefficiency cannot only occur in a receive operation, but also in a probe operation. MPI message probing allows to check for an incoming message without actually receiving it. This operation is particularly useful for determining the size of the probed message, which is necessary for allocating an appropriately sized buffer before the message is received. However, during a probe call, a process can still idle while waiting for the peer process to send the message, and therefore a *Late Sender* wait state can be revealed. Currently, Scalasca completely misses on the probe operations, as they are not present in the underlying trace data.

The primary objective of this thesis is to extend Scalasca to handle MPI message probing. This involves enhancing the event model to capture probe calls during runtime and integrating these events into Scalasca's analysis framework. Scalasca performs different analysis phases to capture different performance metrics. These include the wait state analysis, the delay analysis and the critical path analysis. In the wait state analysis, execution time periods are identified during which a process idles. The delay analysis determines the call paths that are responsible for the identified waiting times. The critical path is the longest execution path without wait states and points out call paths that determine the total runtime of an application.

This work investigates how to support probing of messages in all three analysis phases and describes a prototypical implementation.
The thesis is structured as follows: Chapter 2 offers a comprehensive overview of related work, and the background information necessary for understanding the context of this work. This chapter introduces the Message Passing Interface, with particular focus on point-to-point communication and message probing. Additionally, it outlines the current state of Scalasca, detailing its event model, the various analysis phases, and the limitations it faces in managing probe operations. Chapter 3 presents the extensions made to the event model to capture probe calls and discusses the modifications required in Scalasca's analysis framework to properly handle these events in the various phases of its analysis. In Chapter 4, the extended prototype implementation is evaluated through various test cases and the real-world application ParFlow, showcasing the additional value and insights that can be gained by the enhanced analysis. Finally, Chapter 5 concludes the thesis by summarizing the key contributions and proposing potential directions for future work.

2 Background and Context

This chapter provides the background information necessary to understand this thesis. The first section introduces the Message Passing Interface (MPI) and the concept of MPI point-to-point communication. Special emphasis is placed on message probing and wait states that eventually may occur in this context. The second section is dedicated to performance analysis of MPI applications. The Scalasca performance analysis tool, which utilizes the Score-P measurement tool and the OTF2 Trace Format, is presented here.

2.1 Message Passing Interface (MPI)

The Message Passing Interface (MPI) [1] is an application programming interface (API) consisting of library routines that enable data to be passed between processes. It is widely used in the development of parallel applications and has become the de facto standard for communication in high performance computing (HPC) environments. MPI supports various forms of communication. The main MPI communication categories are point-to-point, one-sided, collective, and I/O operations. Point-to-point refers to the transfer of data between two processes. One-sided communication allows only one process to actively participate in the communication, such as accessing the memory of another process. Collective communication describes a communication operation that involves a group of processes. Additionally, MPI includes I/O operations that allow data exchange between the file system and processes.

An MPI operation can be defined as a series of steps performed by the MPI library with the objective of establishing and enabling data transmission or synchronization. In general, an MPI operation progresses through four stages: initialization, starting completion, and freeing. These stages are implemented as one or more procedure calls. A procedure call denotes a specific MPI function that can be called within the code. During the initialization stage, the argument list is passed to the operation. At this stage, the contents of the data buffers are not yet passed, and the user is still allowed to modify the buffers. Subsequently, the operation starts by transferring control of the data buffers to the MPI library, if any exist. This stage is referred to as the starting stage. From this point on, users are no longer allowed to modify the buffers until control is transferred back. This happens during the third stage, the completion stage, which also indicates whether arguments have been updated. During the following freeing stage, control of additional arguments is returned. The interface allows for blocking, nonblocking and persistent operations. A *blocking operation* goes through all four stages during a single call. A *nonblocking operation* performs the initialization stage and the starting stage during an initiation call. The last two stages are executed in a single completion call. *Persistent operations* involve a separate call for each of the four stages. A pending operation is defined as an operation that has been initiated but not yet completed.

In a blocking call, the invoking process remains in the call until all four stages have been completed. When using nonblocking procedures, the call returns after the operation is initiated, allowing the calling process to perform computation in the meantime and complete and free the operation at a later time. MPI further distinguishes between local and nonlocal procedures based on whether their return requires a corresponding remote procedure call. For example, a blocking receive operation cannot return before the sending process has started the send operation. Therefore, it is a nonlocal procedure. In contrast, a nonblocking receive call is a local procedure because the receive is only initiated, and another call is required to complete the operation.

This work focuses on point-to-point communication, which is discussed in detail in the following section. While collective and one-sided communication, as well as I/O operations, are important aspects of MPI programming, they fall outside the scope of this thesis and will not be discussed further.

2.1.1 Point-to-Point Communication

Point-to-Point communication involves exactly two processes. The core operations for point-to-point communication are sending and receiving messages. One process performs the send operation, while the other one is responsible for the receive operation. The sending process must specify various arguments to identify a message, including the *rank* of the destination process, the *message tag*, and the *communicator*. A *communicator* represents a communication context for a group of processes. Each process belonging to a communicator is assigned a unique *rank* number for identification within this communicator. This integer value may be used to address other processes in the same communicator. The *message tag* is an integer identifier utilized to label a message. These arguments, along with the source rank, which is the rank of the sending process in the communicator, constitute the message envelope. Moreover, the send call takes the address of a memory buffer that contains the data to be transmitted. The receiving process needs to specify the source rank, as well as the message tag and a communicator. In order to receive a message, these arguments need to match the arguments of the message envelope specified by the sending process. It is possible to use wildcard options for the tag and/or the source argument to receive messages with an arbitrary tag and/or from any sender. The receive operation requires the address of a receive buffer that fits the size of the incoming message. Unlike the send operation, the receive operation also takes a status object, which provides information about the actual message envelope and size. This is particularly useful to determine which message was received when wildcard options were used. A receive operation with wildcard options will receive the first pending message that matches the envelope.

MPI differentiates between several communication modes. In *standard mode*, a send is a nonlocal procedure. Its return behavior depends on the message size. If the message is sufficiently small, it can be transferred into a temporary system buffer on the receiver side that can be accessed during the matching receive operation. Hence, the send call may return before the corresponding receive operation was started. Otherwise (i.e., for larger messages), the message is directly transferred into the receive buffer on the peer process. In this scenario the operation cannot complete before the matching receive operation has been initiated. The completion of a *buffered mode* send is independent of the start of a corresponding receive. It differentiates from the standard mode send in that the user is responsible for providing a sufficiently large buffer that allows the message to be held internally on the receiver side until the appropriate receive is called. The return of a *synchronous mode* send guarantees that a matching receive has been initiated, making it a nonlocal procedure. A *ready mode* send can only be invoked if the matching receive has already started, otherwise the operation is erroneous. It thus follows that this send mode can only be used if the sender is aware, through the program logic, that the receive has already been initiated on the peer process. This awareness may allow for a more efficient implementation of the operation than that of the other modes.

Nonlocal procedures require an operation on a corresponding process to be at least initiated. At this point, a synchronization happens because the nonlocal procedure cannot return before the other process reached a specific point in the execution path. A nonblocking call is characterized to be incomplete and local. A procedure call to complete a receive operation that is invoked after the corresponding send has started, might be local. Without the restriction that the corresponding send was started, it would be a nonlocal procedure. This allows nonblocking calls to complete an operation. For instance, an MPI_Test call is nonblocking. It checks if an associated operation can be completed. On confirmation, true is returned, and the operation is completed. To illustrate, a nonblocking send, MPI_Isend, initializes a send operation and returns a *request handle*. The request handle can be utilized to test if the associated operation can be completed and to wait for its completion. To ensure the send operation is finished, the process must call a wait or test procedure with the send request object as an argument. Wait is a blocking call which guarantees that the operation associated with the request handle is completed when the call is left. Test is a nonblocking procedure that returns a flag indicating whether the associated operation is completed. The MPI_Irecv call, which is the nonblocking receive variant, functions similarly. A call to MPI_Wait, or a successful MPI_Test call, guarantee that the entire buffer has been received. The nonblocking completion of an operation includes an implicit synchronization even though it is a local procedure because it secures a happened-before relationship.

2.1.2 Message Probing

In order to receive a message, a receive buffer of sufficient size has to be allocated. There are use cases where the receiving process is not aware of the required buffer size. Therefore, MPI provides a way to determine the size of a pending message: It is possible to probe for a message. Message probing allows to check for an incoming message without actually receiving it. A separate operation is still needed to receive the message. The probe call takes the message envelope as arguments: source, tag, and communicator as well as the address of a status object. Again, it is possible to specify wildcard options for tag and source. The status object allows accessing the actual source and tag in that case, after the call returned. There are several calls that can be used to probe for a message. MPI_Probe is the blocking version that returns when a message matching the arguments is ready to be received. A call to probe checks for the message that would be received by a call to MPI_Recv with the same arguments. The nonblocking MPI_Iprobe functions similarly in case a message is ready to be received, and it returns true. If there is none, false is returned. Another variant are matching probes. In addition to the arguments a regular probe would take, it also takes the address to a message handle. To receive the probed message, a matching receive which takes that message handle has to be invoked. Again, MPI_Mprobe is the blocking call and MPI_Improbe the nonblocking variant, which behaves similarly when successful and otherwise returns false. A message can only be probed once with a matching probe, whereas it can be probed multiple times with the regular probe. It is not necessary to receive a message immediately after it has been probed for. [1]

In a multi-threaded context, the use of MPI_Probe can be problematic. If a process calls MPI_Probe and a matching send operation has been initiated by another process, the MPI_Probe call returns unless the message is received by a concurrent receive call executed by the same process on a different thread. A receive operation following a regular probe will receive the probed message unless the message has already been received on another thread of the same process in the meantime. To avoid these cases, the use of matching probes with matched receives is suggested.

2.1.3 Wait State and Waiting Time

Blocking MPI procedures can exhibit wait states. In general, wait states arise when a process reaches a synchronization point late. This causes one or more processes to idle. The period of idle time is referred to as waiting time. These synchronization points can occur in blocking operations or in the nonblocking completion of an operation. Depending on where the wait state occurs, different patterns of wait states can be characterized. For instance, a receive operation cannot complete before the corresponding send has been started. Figure 1 illustrates this behavior in a timeline diagram. In the diagram, the time is represented on the x-axis, while processes are listed on the y-axis. Each process corresponds to a horizontal line in the diagram. Rectangles aligned with these lines indicate the execution time within specific routines for each process. As illustrated in the diagram, process B starts a blocking receive operation before process A enters the corresponding send operation. In the period before the start of the send operation, it is impossible for process B to receive the message. Therefore, the time which is spent in the receive call before the send call has been entered is waiting time. This inefficiency pattern is referred to as *Late Sender* wait state.

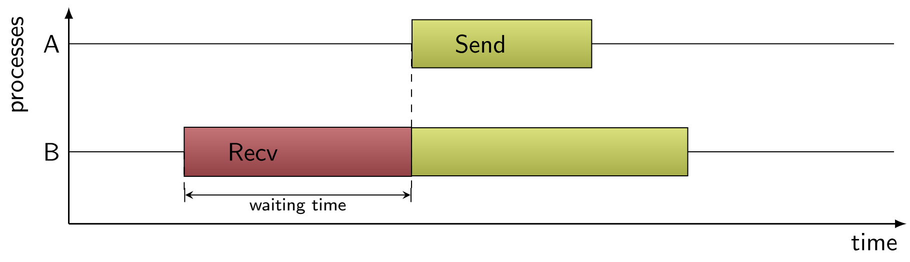


Figure 1: *Illustration of a Late Sender Wait State.*

Figure 2 visualizes that process B enters the receive call after process A enters the send call, but before A leaves the send call. This indicates that a synchronization happened and the send call has been waiting for the start of a related receive operation, as described in Section 2.1.1. In this case, the send call can exhibit a *Late Receiver* wait state. Therefore, the time spent in the send call before the matching receive has been started is waiting time, since the send operation cannot complete before the receive operation started.

Similarly, these wait states can occur in wait calls which complete a communication operation that was initiated with a nonblocking procedure. Likewise, a blocking probe call can exhibit waiting time. In this case, distinguishing whether the send operation synchronizes is sensible. For a non-synchronizing send, the probe call may contain *Late Sender* waiting time, which is identical to the waiting time that would have been experienced by a receive at the same point in time (see Figure 3a).

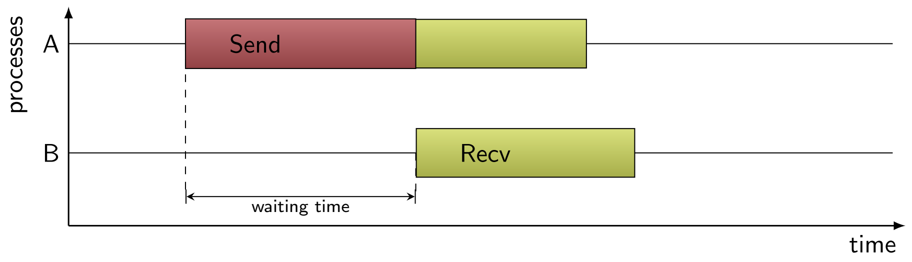
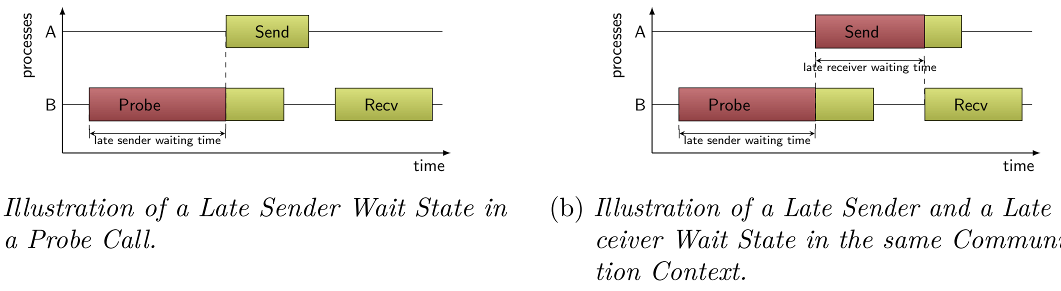


Figure 2: *Illustration of a Late Receiver Wait State.*

A synchronizing send, on the other hand, can exhibit a *Late Receiver* wait state, as it waits for the start of the receive operation rather than the probe operation. Simultaneously, the probe call may also exhibit a *Late Sender* wait state within the same communication context. This results in the send operation having two distinct synchronization points. The first is with the probe, where the probe exhibits its *Late Sender* wait state, and the second is with the receive call, where the send reveals the *Late Receiver* wait state. This case is illustrated in Figure 3b.



(a) *Illustration of a Late Sender Wait State in a Probe Call.*

(b) *Illustration of a Late Sender and a Late Receiver Wait State in the same Communication Context.*

Figure 3: *Visualization of Wait States in Probe Calls.*

2.2 Performance Analysis Tools

Supercomputers offer massive amounts of computational resources that are underutilized by the majority of users. Identifying and addressing performance bottlenecks is therefore essential for optimizing resource usage and enhancing application efficiency. Performance analysis tools assist users in tuning their application. The most common approaches used by such tools are profiling and tracing. In profiling, the performance data is stored in a summarized format. Profile data typically comprise aggregated data, including information about the overall runtime of various code sections, the frequency with which certain methods are called, memory consumption, and other relevant metrics. In contrast, tracing refers to the recording of information about the flow of an application during its execution. The recorded information may be in the form of events, which are defined as specific occurrences during execution, such as entering or leaving code regions. For a more detailed description, please refer to Section 2.2.1. All details about the execution of the application are stored in trace files. Tracing can provide detailed insights into the chronological sequence of events as well as call-specific information, such as sender, receiver, data volume and identification of an MPI message. The detailed recording of data during tracing causes the amount of recorded data to grow in proportion to the execution time and the number of processes and threads. There exist two principal methods to acquire performance data. On the one hand, there is sampling. Here, the application is interrupted at a certain frequency, the sampling rate. The address of the current instruction can be determined by reading the program counter, and the length of the sampling interval can be added to the cumulative execution of the current source code region, for example. Sampling can solely be used to collect statistical information, which is why it is typically employed for profiling purposes. The runtime overhead depends on the sampling rate. Sampling cannot ensure that every source code region is captured. On the other hand, there is instrumentation. Instrumentation refers to the process of adding additional code to a program to collect performance data. In contrast to sampling, this guarantees that each function is recordable. [4]

Well-known tools to collect performance data are Score-P [5, 6], Extrae [7] and HPCToolkit [8]. Score-P and Extrae support both instrumentation and sampling, while HPCToolkit is a sampling-only tool primarily utilized for profile creation. In this work, the focus will be on instrumentation. The analysis tool Paraver [9] is based on Extrae. Score-P is compatible with the analysis tools Vampir [10], Scalasca [11], TAU [12] and formerly Periscope[1] [13]. Scalasca and Vampir are post-mortem tools that process OTF2 [14] trace files. Vampir provides an interactive visualization that allows the user to analyze trace data based on a graphical representation. Scalasca automatically detects performance bottlenecks and categorizes them, which is especially useful for larger traces. TAU is a tool framework that allows for instrumentation, measurement, and analysis. As a part of TAU, the Program Database Toolkit offers automatic instrumentation. Profile data can be visualized using ParaProf [15], and data mining is done via PerfExplorer [16].

2.2.1 Score-P

Score-P [5, 6] offers a joint measurement infrastructure that allows to analyze data with various tools. It provides an instrumentation framework to collect performance-related data like times, visits, communication metrics or hardware counters during the execution of the program. To gain that information during runtime, the application needs to be linked against a provided runtime library that matches the desired parallelism. For instance, Score-P offers separate libraries for serial, MPI, OpenMP

[1]Periscope was an on-line analysis tool that evaluated performance during runtime. Meanwhile, the project has been abandoned.

and hybrid execution. Score-P supports both profiling and tracing techniques. The output depends on the user's specifications. When profiling is desired, a profile in the Cube4 [17] format is produced, whereas for tracing the recorded data is stored in the OTF2 file format, that is presented in the next paragraph. In order to detect wait states, tracing is required because detailed information about the chronological order of the application execution is needed.

Open Trace Format 2 (OTF2)

The Open Trace Format 2 is a trace data format based on the EPILOG [18] trace format and the Open Trace Format (Version 1) [19] that were previously used by the tools Scalasca and Vampir [14]. Both Scalasca and Vampir are now compatible with the OTF2 format. Event trace data constitute a layer that is able to move data from the runtime measurement to the post-mortem analysis. OTF2 is a highly scalable and memory efficient event trace data format. The memory efficiency is reached by separate storage of specific files to avoid redundancies. It uses an anchor file in order to manage the trace data, a global definitions file and separate local trace files and local definition files for each location. A location refers to a specific execution instance, like a process or thread. A local trace file contains events that were recorded on a specific location in temporal order. Each event contains a timestamp and other event-specific attributes. Events can reference the separately stored definitions. Definition objects do not contain a timestamp and refer to objects that are relevant during execution over a longer period of time, for example communicators. The global definition file stores definitions for all locations. In order to avoid additional communication during data collection, each location uses separate local identifiers in its local trace file to reference definition records. The local definition files contain mapping information between the local and global definitions, and as well store definitions that only occur on the specific location. This allows reducing the memory requirements by avoiding redundancies in traces. The OTF2 event model specifies various events, including entering and exiting code regions, sending and receiving messages and exiting collective communication operations.

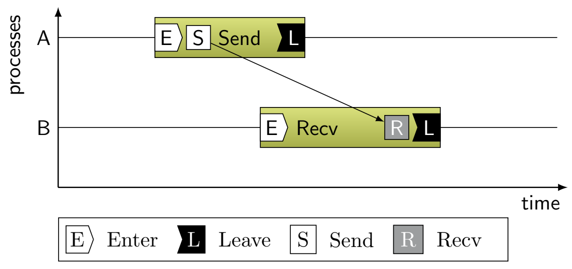


Figure 4: *Illustration of OTF2 Events in Blocking Point-to-Point Communication.*

Figure 4 illustrates the events that are collected during a point-to-point message exchange using blocking communication. For each code region, an enter event (E) marks the beginning, and a leave event (L) marks the end of the region. The send event (S) is generated at the outset of the corresponding region, thus before the actual send happens. A receive event (R) is generated only after the receive has been completed. The arrow represents the transfer of a message.

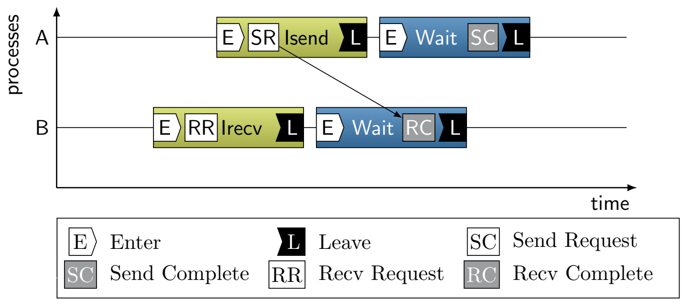


Figure 5: *Illustration of OTF2 Events in Nonblocking Point-to-Point Communication.*

Figure 5 depicts the events that are recorded during nonblocking point-to-point communication. Separate MPI calls are invoked to initiate and complete an operation, resulting in the recording of request and completion events, respectively. The send operation is initiated with an MPI_Isend call, which generates a send request event (SR). This event marks the earliest point in time at which the message could be sent. In the subsequent MPI_Wait call the operation is completed and the send completion event (SC) is recorded at the end of the region. The event guarantees that the sending of the message has been completed. Similarly, the receive request event (RR) and receive completion event (RC) are recorded on the peer process.

2.2.2 Scalasca

Scalasca [2, 3] is a software tool designed to assist users in enhancing the performance of parallel programs, particularly on large-scale systems. It identifies where performance is lagging, especially in terms of communication and synchronization between processes. Scalasca not only highlights these bottlenecks but also helps to understand the underlying causes. This information is gained by the post-mortem analysis of trace data generated with the measurement tool Score-P and present in the OTF2 format. Scalasca, which is mainly programmed in C++, processes the local trace data in parallel using a replay-based analysis approach. The analysis uses the same number of cores as the application itself. Consequently, the memory and processing power available for the analysis increase in proportion to the number of cores.

Before starting the analysis, Scalasca loads the global definitions and local event trace data into the main memory of each process. All occurring object references are unified using the mapping table from the local definition files. Scalasca performs a preprocessing step to ascertain the completeness of the global call tree provided by the trace definition data with respect to the local trace data. Additionally, the local trace is preprocessed to provide the full trace-access functionality. For example, for nonblocking operations an iterator syntax is established, that allows to access the offset to the subsequent event using the same request. Scalasca's parallel analysis workflow is illustrated in Figure 6. All event specific attributes can be accessed via the events in the trace data. Furthermore, Scalasca provides an iterator syntax for navigating through the local trace.

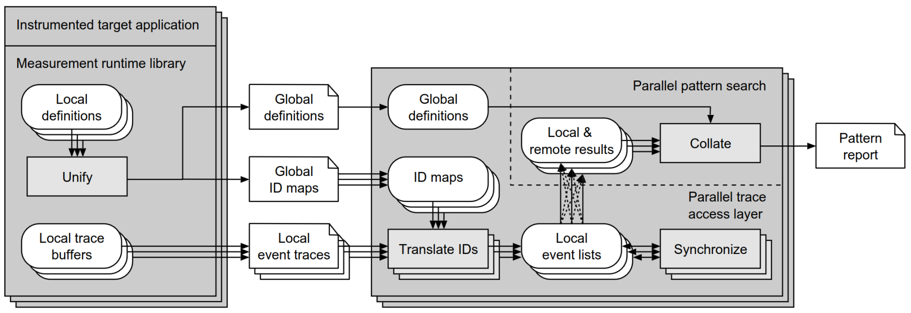


Figure 6: *"Schematic overview of Scalasca's parallel analysis workflow. Gray rectangles denote programs, white rectangles with the upper right corner turned down denote files, and bubbles denote data objects residing in memory. Stacked symbols denote multiple instances of programs, files or data objects running or being processed in parallel."* [3]

Scalasca replays the original communication in order to exchange the process local trace data. Scalasca performs a total of five communication repetitions for various analyses. Prior to Scalasca's analysis of the event trace data, an optional timestamp correction can be performed, which includes two additional replays of the communication. The purpose of the timestamp correction is to guarantee the correctness of the logical order of events. Event records collected on different nodes can have discrepancies due to clock drift or different clock speeds. Throughout the replays, the trace data is traversed in either forward or backward direction. When a communication event is reached, the original communication operation is mimicked by a similar communication operation. In these replays, performance data of the communication regions is transmitted instead of the original buffer content. In case of a backward replay, the communication direction is reversed, resulting in a send event triggering the posting of a receive and vice versa. During the first forward replay, all wait states excluding the *Late Receiver* wait state are detected. The subsequent backward replay identifies *Late Receiver* wait states. Additionally, the synchronization point information of the identified wait states during the first replay is transmitted to the ranks that participated in the corresponding communication operation. In the third replay, the synchronization point information for the *Late Receiver* wait states is transferred. The fourth replay is conducted to calculate the delay costs (i.e., the total amount of waiting time caused by an imbalance, see Section 2.2.2) and to determine the critical path (see Section 2.2.2). The final replay is necessary to classify the detected wait states. The different analyses as well as the timestamp correction are explained in the following paragraphs. To avoid the transmission of redundant messages during the same analysis, specialization relationships between patterns are used to reuse already obtained results. This is realized through an event notification and callback approach, and necessitates that each message be replayed a single time, even if it is part of multiple patterns. Whenever the algorithm detects an inefficiency pattern, it calculates the severity of this pattern instance and the value is then accumulated in a matrix consisting out of the pattern and the call path. This matrix is stored on the analysis process responsible for the rank exhibiting the wait state. At the end of the trace traversal, the local results are combined in a three-dimensional matrix consisting of pattern, call path, and rank. As a result of the analysis, a Cube4 format file is created, which can be evaluated with the visualization tool CubeGUI [20]. As this work concerns only point-to-point communication, all analyses are thus explained in relation to that case.

Timestamp Correction

Distributed computing systems consist of multiple nodes. Each node has its own clock with limited or no synchronization. As a result, event records collected on different nodes can have discrepancies due to clock drift or different clock speeds. When it comes to analyzing the performance data, it strongly depends on the comparability of the recorded timestamps. To enable a meaningful analysis, a global time should be applied. Assuming each process is running with a constant but different clock, it is possible to calculate the global time of each process as a linear function of the local time of each process. In order to achieve this, Score-P is taking offset measurements at MPI initialization and finalization and a linear offset interpolation is performed. Nevertheless, assuming a constant drift is only an approach which can still lead to so-called clock condition violations. Clock conditions ensure the logical order of events. A clock condition violation, therefore, refers to a situation in which a happened-before condition is breached. For example, when the timestamp of a receive event is less than that of the corresponding send event. It should be noted that a send event always indicates the start of a send operation, and a receive event marks the end of a receive operation. In comparison to singular erroneous timestamps, clock condition violations can be detected quite easily. To correct clock condition violations, Becker et al. [21, 22] extended the controlled logical clock algorithm [23] and integrated it into Scalasca. This algorithm is a method for retroactively correcting timestamps that violate clock conditions. This is achieved by shifting events and attempting to preserve local intervals. For example, if a clock condition violation occurs in a send-receive pair, the receive event would be shifted forward in time to happen after the send event. To maintain the length of local intervals between events, the events in proximity would also be shifted. The events are shifted in a manner that the further away they are from the receive event, the smaller the shift. In this way, the shift has an effect only in a certain area. Nevertheless, such a shift can possibly arise in new violations. To prevent this, the algorithm performs two phases. The first phase consists of a forward replay and during this replay, the *forward amortization* takes place. Here, the event responsible for the clock condition violation is shifted, as well as the events immediately following the causing event, while ensuring that the total runtime of the program is not significantly increased. In the second phase, a backward replay, also called *backward amortization*, the previous events are shifted to flatten jump discontinuities introduced in the forward replay using a piecewise process-local linear correction in an amortization interval. The interval size is selected in a manner that precludes the generation of new clock conditions as a result of the shifting of surrounding events.

Wait State Analysis

Scalasca's wait state analysis happens during the first two replay passes of the analysis phase [2, 3]. The event traces are searched for characteristic execution patterns that indicate wait states. The main idea is to calculate waiting times on the processes on which they occur. The algorithm traverses the local traces in parallel and exchanges performance data when an event indicating a communication operation is encountered. During the forward pass, the original communication is replayed. For instance, when a send event is reached in the trace, the replaying process invokes a send with the same envelope as in the original application. Instead of sending the original application data, the event data of the send event, along with the event data of the corresponding enter and leave events, are transferred. When the corresponding rank encounters the receive event in their local trace, this information is received. Figure 7 visualizes this behavior and shows the events that are recorded for a send-receive pair. When process A encounters the send event, it transfers the relevant performance data, including the enter timestamp of that region, to process B. Likewise, when process B encounters the receive event, it posts a receive operation. Once it has received the data, it is able to calculate whether waiting time occurred in the receive region. If the remote enter timestamp of the send region is greater than that of the receive region, a *Late Sender* wait state is detected. The waiting time is calculated by subtracting the local enter timestamp from the received remote enter timestamp. When a wait state is found, a synchronization point is stored. In case of the *Late Sender* wait state, the sender and receiver rank are synchronizing. This synchronization point is stored on the receiver side, as the wait state is detected here. In the subsequent replay, which is a backward replay, the direction of communication is reversed. In this instance, the receiver transmits the synchronization point information to the sender.

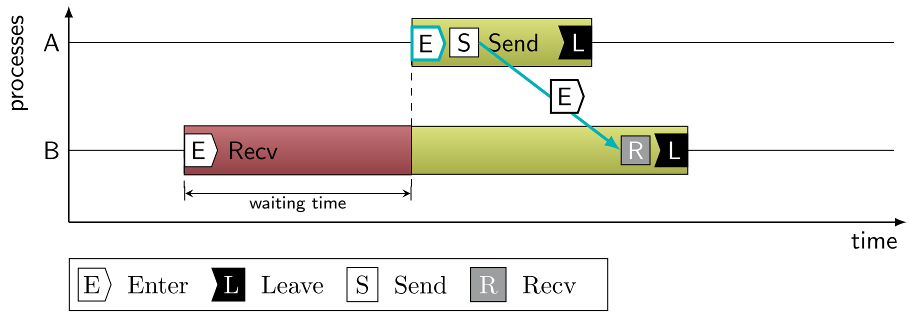


Figure 7: *Illustration of the Communication Replay for the Late Sender Wait State.*

Another form of the *Late Sender* wait state is the *Late Sender Wrong Order* wait state. Here, the recipient waits to receive a message while another message is ready to be received. To determine a wrong order situation, each process owns a ring buffer which stores the last occurrences of *Late Sender* wait states. Whenever a receive event is reached, the timestamps are compared to find out if a *Late Sender* instance in the ring buffer correlates with a wrong order situation. Figure 8 illustrates the aforementioned situation.

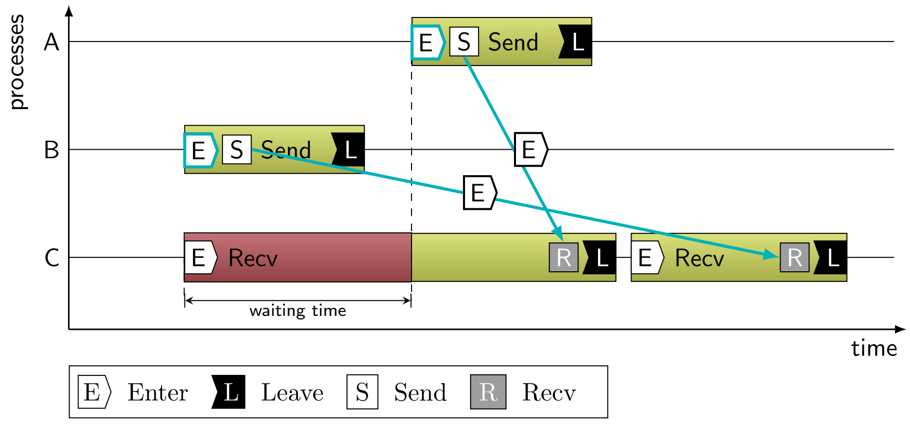


Figure 8: *Illustration of the Communication Replay for the Late Sender Wrong Order Wait State.*

In the initial receive operation of process C a *Late Sender* wait state is detected. Here, the enter timestamp of the send operation from process A is received. Therefore, the send enter timestamp as part of the *Late Sender* instance is stored in the ring buffer. The following receive operation of rank C receives the enter timestamp of the

send operation from rank B. Subsequently, the received timestamp is compared with the timestamp stored within the ring buffer. Because the send enter timestamp of rank B is less than the timestamp of the send enter event that corresponds to the *Late Sender* wait state detected before, the wait state is classified as a wrong order situation.
During the backward pass, *Late Receiver* wait states are identified. The communication direction is reversed. That means a send event in the trace leads the replaying process to invoke a receive operation with that envelope. A receive event leads the corresponding rank to post a send operation, sending the timestamps of the receive region. Figure 9 shows the communication replay. Process A compares the received enter timestamp with its own enter and leave timestamps of the send region. If the enter timestamp of the receive operation falls between the enter and exit timestamps of the send region, a *Late Receiver* wait state is identified. The waiting time is calculated as the difference between the two enter timestamps. The synchronization point is stored on the sending process and during the next forward replay it is transferred to the receiver's process.

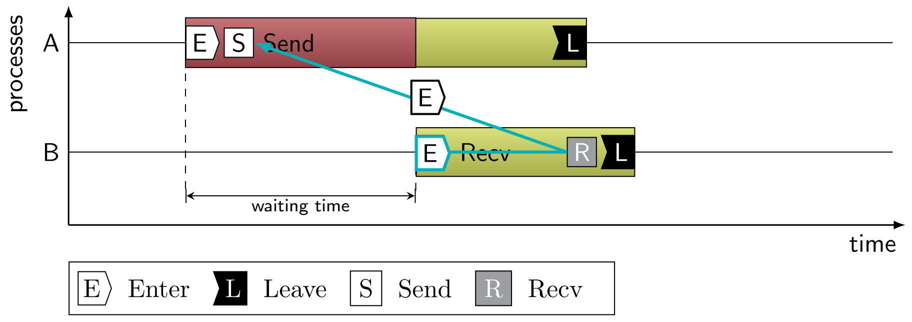


Figure 9: *Illustration of the Communication Replay for the Late Receiver Wait State.*

Delay Analysis

Böhme et al. [24, 25] extended Scalasca's wait-state detection by a delay analysis pass to identify the root causes of wait states. A delay is the counterpart of a wait state. It is defined as an interval that causes a process to arrive late at a synchronization point and thus induces waiting time on the corresponding peer process. The costs of a delay are the amount of wait states it causes. Delay costs may far exceed the delay itself. The cost model allows for a ranking of delays according to their associated resource waste. The delay costs are further distinguished into short-term and long-term delay costs. Short-term delay costs describe the amount of direct waiting time that is caused by the delay whereas long-term delay costs refer to the amount of indirect waiting time a delay causes. This is illustrated by Figure 10 using two *Late Sender* wait states. All processes execute a routine *Foo*. Process A spends more time in that routine than the other two processes and therefore delays the following send operation. Right after leaving the routine *Foo*, process B enters a receive operation to receive a message from process A. Here, process B reveals a *Late Sender* wait state, since the corresponding send has not been entered yet. The wait state is a direct result of the delay in routine *Foo* on process A. As such, the waiting time are short-term costs of this delay. Process C as well enters a receive operation right after leaving routine *Foo* in order to receive a message from B. Here, another *Late Sender* wait state is encountered. The wait state in the receive operation of B subsequently causes the wait state on process C. Therefore, the amount of wait state that is caused by the imbalance in routine *Foo* are the long-term costs of the delay, and it is an indirect wait state. However, process B also has a delay in the receive operation, which directly causes the second part of the wait state on process C. Consequently, the amount is short-term delay costs, as this is a direct wait state. The wait state on process B is designated a propagating wait state, as it precipitates a wait state on process C. The wait state on process C is classified a terminal wait state, as it represents the end of the causal chain and is not a source for further wait states.

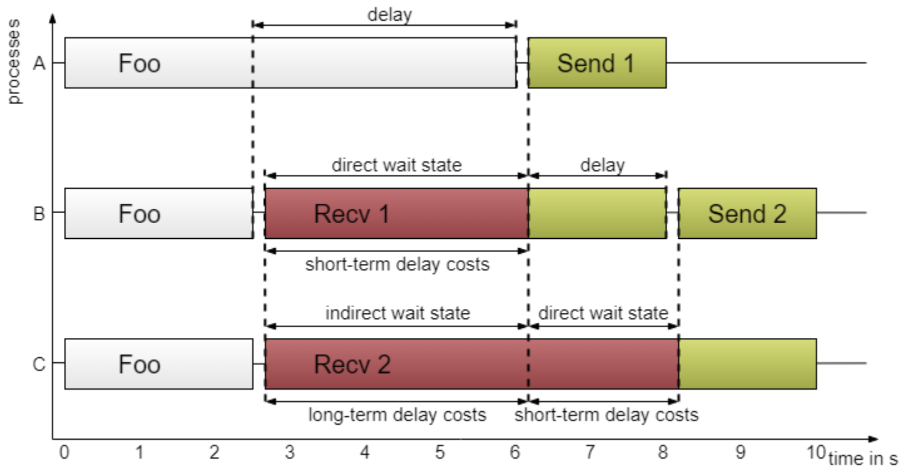


Figure 10: *Illustration of Short- and Long-term Delay Costs.*

This section provides a conceptual description of the process for determining delay costs. It does not aim to provide a comprehensive account of the details, as this would exceed the scope of the present work. The delay costs are calculated during a backward replay. They are calculated on the causing rank, which is the rank where the delay occurs. For the example visualized in Figure 10 these are the sender ranks. For all wait states detected in the former replays, the root cause analysis is triggered. First, the execution differences since the previous synchronization point between the delayed and the waiting process are characterized. Taking a look at Figure 10 two synchronization intervals can be determined, assuming that a synchronization between all processes happened in front of routine *Foo*. Process A and B share a synchronization interval until the end of *Recv1*. The synchronization interval from process B and C reaches from *Foo* to the end of *Recv2*. Each process calculates a time vector d which contains the execution times for each region within the specified interval, excluding waiting times and the concluding communication operations. Additionally, a vector containing the waiting time, denoted as ω, is calculated for each process.

$$d(\text{A,Foo,Send1}) = \begin{bmatrix}\text{Foo: 6}\end{bmatrix} \tag{2.1}$$

$$d(\text{B,Foo,Recv1}) = \begin{bmatrix}\text{Foo: 2.5}\end{bmatrix}, \qquad \omega_B = \begin{bmatrix}\text{Recv1: 3.5}\end{bmatrix} \tag{2.2}$$

$$d(\text{B,Foo,Send2}) = \begin{bmatrix}\text{Foo: 2.5}\\ \text{Recv1: 2}\end{bmatrix}, \qquad \omega_B = \begin{bmatrix}\text{Recv1: 3.5}\end{bmatrix} \tag{2.3}$$

$$d(\text{C,Foo,Recv2}) = \begin{bmatrix}\text{Foo: 2.5}\end{bmatrix}, \qquad \omega_D = \begin{bmatrix}\text{Recv2: 5.5}\end{bmatrix} \tag{2.4}$$

Upon reaching a receive event during the backward replay, the calculated time vector d, and the waiting time w are transferred to the sender. The sending process is then able to calculate a delay vector δ, containing the execution differences excluding waiting times. During the backward replay, routine *Recv2* on process C and routine *Send2* on process B are encountered first. Here, process B receives the values from Equation 2.4 and is able to calculate δ:

$$\delta(\text{B}) = d(\text{B,Foo,Send2}) - d(\text{C,Foo,Recv2}) = \begin{bmatrix}\text{Recv1: 2}\end{bmatrix} \tag{2.5}$$

The delay vector δ contains the amount of delay that is located on the rank. In the next step, the short- and long-term delay costs are calculated. Taking a look at Figure 10, the short-term costs of the delay on rank B are the amount of direct waiting time in *Recv2* on rank C. B divides the total amount of waiting time in *Recv2* obtained from vector ω into direct and indirect waiting time by comparing it to the amount of delay $\delta(B)$. The total amount of delay on B equals two seconds and therefore process B causes two seconds of the wait state on process C directly, which are the short-term costs of the delay. The remaining 3.5 seconds waiting time result in the propagation of the wait state on process B and therefore are long-term costs of the delay that causes that wait state. This information is stored in a propagation factor φ, that indicates whether a wait state propagates further. When rank B encounters the receive event in routine *Recv1* the time vector $d(\text{B,Foo,Recv1})$, the waiting time ω_B, as well as the propagation factor $\varphi = 3.5$ are transferred to rank A which is then able to calculate its delay vector:

$$\delta(\text{A}) = d(\text{A,Foo,Send1}) - d(\text{B,Foo,Recv1}) = \begin{bmatrix} \text{Foo: } 3.5 \end{bmatrix} \tag{2.6}$$

Process A as well compares the amount of its delay with the waiting time on process B. The amount of the delay matches with the amount of waiting time and therefore the short-term delay costs are 3.5 seconds. Since the wait state on process B propagates, the long-term delay costs of the delay are the propagation factor of that wait state and equal 3.5 seconds.

Critical Path Analysis

The critical path is defined as the longest execution path without wait states. It points out activities that determine the total runtime of an application. An activity is defined as a single execution of a call path by a specific process. Improving the runtime of activities that are on the critical path allow to potentially improve the total runtime of the application, whereas performance optimization of activities that are not on the critical path only will extend wait states but not improve the overall runtime. To analyze the critical path, Böhme et al. [26] introduce different metrics: the critical path profile and the critical path imbalance indicator. The imbalance indicator hints how much time is wasted in a call due to load imbalance, and the critical path profile shows how much time an activity spent on the critical path. The critical path analysis takes place in a backward replay. First, it is identified which MPI rank entered the finalize call last. This determines the endpoint of the critical path. The critical path can only be on one process at a time, therefore an ownership flag is set. The critical path stays on a process until the next communication event which exhibited waiting time is encountered. Here, the synchronization point information stored during the former replays is utilized. The critical path will then transition to the process which is responsible for the waiting time.

Challenges in Analyzing Probe Calls

If the probe call is to be taken into account in the various analyses, one encounters the problem that in the case of probed receives, there are two synchronization points for the corresponding send event. One synchronization pair consists of the send and the probe events and the other of the send and the receive events. The exchange during the analyses takes place at the corresponding synchronization points and is designed to use a similar communication operation for data exchange. For example, to be able to recognize a *Late Sender* wait state in probe, the enter timestamp of the send operation would be required in the probe region. This raises the question of how probe should be handled in the forward and backward replays. The following chapter will examine various strategies that can be employed to include probe into the different analyses.

3 Analysis of Message Probing

At present, Scalasca is not capable to analyze probes. This is due to the fact that Score-P does not gather performance data for probes during runtime, as there exists currently no OTF2 event to store call-specific probe information. This chapter therefore covers the introduction of appropriate OTF2 events and discusses how these events can be handled in the various analyses performed by Scalasca.

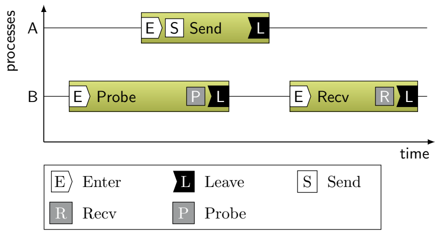
3.1 Extended Event Model

This section discusses which events are required in relation to probe calls. In order to incorporate probe calls into the various analyses of Scalasca, it is essential to collect data pertaining to these calls during runtime, with the objective of generating events associated with them. The initial consideration is limited to defining the requirements for which information should be obtained from the event record for a regular probe, prior to an examination of the specific attributes that are required for a matching probe.

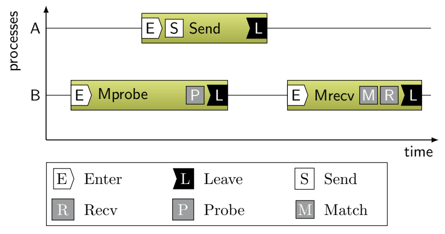
In the wait state analysis, for instance, a probe requires the event data from the send with which it synchronizes in order to calculate a potential *Late Sender* wait state. Consequently, the probe event must be capable of providing the necessary attributes to execute a receive operation that receives the event data of the corresponding send. As previously outlined in Section 2.1.1, the message envelope is required for this. The relevant information about the source rank and the message tag can be retrieved from the status object returned by the probe call. Since the communicator argument in the probe call is unable to take a wildcard option, the information can directly be extracted from the call parameters. This proposal would therefore result in a probe event containing the parameters of the source rank, the message tag and the communicator. Glancing at a receive event record, it becomes evident that this event comprises the proposed parameters for a probe, in addition to other parameters. This leads to the idea that it is sufficient to reference the corresponding receive event in a probe event, rather than the individual parameters, given that these can be accessed in Scalasca via the receive event. In order to implement this approach, Score-P would have to perform such a matching between the probe event and the probed receive event at runtime. This would require Score-P to track all probe events until the corresponding receive occurs to be able to correlate them. In a multi-threaded context, a shared data structure with an appropriate locking mechanisms must be used for this purpose. If multiple messages with the same envelope are pending, a probed message may be received by a concurrent receive operation executed by the same process on a different thread, even if there is a receive operation that directly follows the probe call (see Section 2.1.2). In such a situation, it may vary from execution to execution which of the messages is retrieved at which receive operation, and determining the correct matching is not possible. This matching problem is inherent and occurs regardless of whether the matching is performed by Score-P or Scalasca. Therefore, it must be assumed that the probe and the receive occur on the same location. Given that the MPI standard recommends the use of matching probes in a multi-threaded context, this assumption is reasonable. If Score-P is responsible for matching probe and receive events, this would result in additional runtime overhead during the measurement due to the required event tracking. Consequently, this approach has been rejected, and the former one is noted for now.

Next, the requirements that are placed on an event for matching probes are examined. In this context, a call to MPI_Mrecv would receive a probed message using a message handle that was returned by an MPI_Mprobe call. Currently, Score-P records a regular receive or receive complete event after the reception of a message by a matched receive, depending on whether MPI_Mrecv or the non-blocking variant MPI_Imrecv was utilized. Given that both the probe and the matched receive utilize the same message handle, it is feasible to store an ID correlating with said message handle as the event data for a matching probe and to substitute the receive event by a matched receive event comprising that message handle ID, in addition to the arguments of a regular receive event. A matching between these events can be achieved in the preprocessing step of Scalasca, similar to the offset iterator syntax established for events for calls using requests, as detailed in Section 2.2.2. This would allow for direct access to the matching receive event in a probe event and vice versa. Upon encountering a matching probe event in the trace, the associated matched receive event can be accessed, and its parameters can be used to receive the event data of the corresponding send. It is possible that further analyses may be triggered by a receive event that requires the event data from the send region. Therefore, it should be possible to determine during the analysis which receive corresponds to which probe. Based on the approach discussed so far, this would be possible, as in the case of a probe event, the related receive event would be the next one where the parameters of the message envelope match those of the probe event. In the case of a matching probe event, the iterator syntax would be sufficient for that purpose.

During the development of this thesis, the OTF2 and Score-P developers undertook concurrent efforts to standardize the events for regular and matching probes. Having the same event representation for the probe and the matching probe would be advantageous, as it would allow for analysis in the same manner. Since there should be a possibility that a receive may be correlated to its probe, it is not possible to apply the event suggestion for a regular probe for both, the probe and the matching probe. Given that the message handle is strictly necessary to receive a message that was probed with a matching probe, the message handle ID should be part of the event data. Furthermore, the introduction of a distinct matched receive event is not advisable, as it would necessitate modifications to the existing analysis, since the regular receive event is a well-established trigger for specific pattern recognition algorithms. Accordingly, an additional event containing the message handle ID is stored in a matched receive region in front of the receive event. This event is called a match event. The regular probe still requires the message envelope to be able to correlate it to its receive event. Therefore, a probe event is written for both matched and regular probes, comprising the call-specific information pertaining to the message envelope and an additional message handle ID. To distinguish between matching and regular probes, the message handle ID equals zero in case of regular probes. Score-P is responsible for storing the message handle IDs in the event data. The measurement system assigns consecutive message handle IDs, starting with the value of 1, which leaves the value of 0 for regular probes.



(a) *Illustration of the Events in Regular Probes.*



(b) *Illustration of the Events in Matching Probes and Matched Receives.*

Figure 11: *Illustration of the Events in Probe Calls.*

Figure 11 illustrates the sequence of events recorded for the regular and matching probes. In the case of the nonblocking variants, the event sequence would be similar when the probe is successful, and no specific event would be recorded for an unsuccessful probe. As explained above, it is not possible to correlate the regular probe and the receive in a multi-threaded context with certainty. In the case of matching probes, it is possible for threads to share access to the message handle, with one thread executing the matching probe operation and another thread executing the matched receive operation. In this case, a correlation can still be made via the message handle. Scalasca is currently only able to handle pure MPI applications or multi-threaded applications using the option `MPI_THREAD_FUNNELED`, i.e., only the main thread invokes MPI calls. Consequently, in applications that can be analyzed by Scalasca, MPI calls can only occur on one location per process. Therefore, the description of the implementation of the individual analyses assumes that a probe event record occurs on the same location where the corresponding receive takes place.

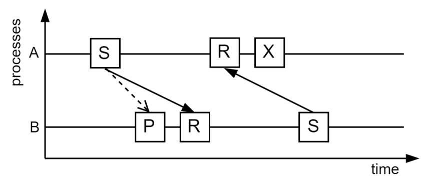
3.2 Timestamp Correction

The process calling the probe synchronizes with the peer process that is calling the send. Accordingly, a clock condition must be met: The send has to start before the probe returns, namely the timestamp of the send event has to be prior to that of the probe event. However, due to unsynchronized clocks it may happen that the recorded timestamps do not reflect the actual order of events, resulting in a scenario where the probe event occurs before the send event. This is referred to as a clock condition violation. Such a clock condition violation has to be addressed through the timestamp correction. This is achieved through two phases, the so-called forward and backward amortization. Initially, the steps required to incorporate probe events during the forward amortization are examined. The forward amortization takes place in a forward replay. Here, the event data is exchanged when communication events are encountered in the traces using a similar communication operation. In order for the replaying process to perform any action upon appearance of a probe event in the trace, a callback must first be registered that ensures the invocation of a specific method when a probe event is encountered. To identify a clock condition violation in the probe, it is necessary to compare the timestamps of the send and the probe events. Consequently, the callback method for a probe event must post a receive operation with the parameters that are stored in the event data to receive the event data from the send. Now, the algorithm described in Section 2.2.2 can be applied in the same manner as for a send-receive pair, with the exception that the timestamp of the probe is taken into account, rather than that of a receive.

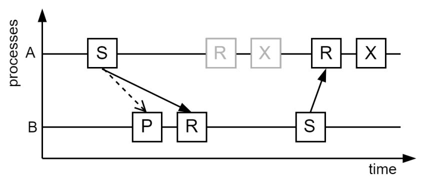
As previously stated, the analysis assumes that the probe and the receive event occur on the same location. In the absence of a clock condition violation in the send-probe pair, it follows that the subsequent receive, which received the probed message during the original execution of the application, cannot have such a violation. This is because a probe must always precede the corresponding receive. In the event of a clock condition violation in the send-receive pair, a corresponding violation must also be present in the send-probe pair. This is due to the fact that the events on a given location must align with the logically correct sequence. During the forward replay, the probe event would be encountered first, and the forward amortization would correct the violation by shifting the event responsible for the violation, namely the probe event, as well as any subsequent events, including the receive event. Therefore, at the point where the replay reaches a receive event that represents a probed receive, it is no longer possible for a clock condition violation to be present.

Two remaining weaknesses are identified in this method. Firstly, the transmitted event data is received when the probe event is encountered. However, the analysis process responsible for that location still invokes a receive call when the receive event is reached in the trace. In the case of a probed receive, the execution of this operation must be prevented, otherwise a deadlock will occur. A similar problem arises if the same message has been probed several times. Here, the analysis process may only initiate a receive operation when encountering the first of that probes, since this represents the synchronization and the processes would deadlock if the receive operation is posted multiple times, as there is only a single related send operation. To address these issues, it must be possible to ascertain whether a probe event with identical parameters has already occurred before the probed message was received. In the case of a receive event, it is crucial to be able to determine whether it is a probed receive. As a probed receive can be disregarded in the timestamp analysis, there is no requirement to pass the data received when the probe event was encountered to the callback function of the corresponding receive event. Consequently, the data structure of an unordered set would be sufficient to retain the information about previously encountered probe events during the replay, whose related receive events have not yet occurred. A struct can be defined that contains variables for communicator, source rank, message tag, and message handle ID. Moreover, it offers an equality operator. Upon detection of a probe event in the trace, an instance of the aforementioned struct is created using its arguments. If an entry matching the struct data is already present in the unordered set, the callback function for the probe will return without invoking a receive operation. Otherwise, the receive operation and the forward amortization are performed, and the struct instance is then added to the set. When a match event is encountered, its message handle ID is stored in a process-locally accessible variable. The default value for the aforementioned variable is zero, which corresponds with the message handle ID of a regular probe. The match event occurs in the same matched receive region as the receive event. Therefore, no other match or receive event can be in between. Finally, when a receive event is reached, it is checked if a struct instance, containing the communicator, source rank, and message tag obtained from the receive event as well as the message handle ID that is stored in the process-local variable, exists in the unordered set. In this case, the receive event represents a probed receive and the callback method returns after deleting the instance from the set. From this point on, another probe with the same arguments would probe for a new message. In addition, the message handle ID variable is set to zero again. Adopting this approach would resolve the clock condition violations in send-probe pairs. However, as backward amortization has not yet been considered, there is a risk that the correction of other violations could result in the creation of a new violation in a probe.

Figure 12 illustrates the forward amortization for a clock condition violation in a send-receive pair. The send (S), receive (R) and probe (P) events are denoted with their first letter. For the sake of clarity, the enter and leave events are not shown.



(a) *Illustration of a Clock Condition Violation in a Send-Receive Pair.*



(b) *The causing event (R) and the subsequent event (X) are shifted forward in time.*

Figure 12: *Forward Amortization to correct Clock Condition Violation.*

An arbitrary event is denoted as X. The solid arrows illustrate the message exchange at runtime, while the dashed arrows show the synchronization between the probe and the send. Here, a clock condition violation is indicated by an arrow pointing backwards in time. Figure 12a shows such a violation between the receive event on process A and the send event on process B. During the forward replay, the related send event timestamp is received by process A, and the clock condition violation is rectified by shifting the receive event to occur after its send event. Afterward, also the following events are shifted. In this case, this concerns event X, and the result is shown in Figure 12b.

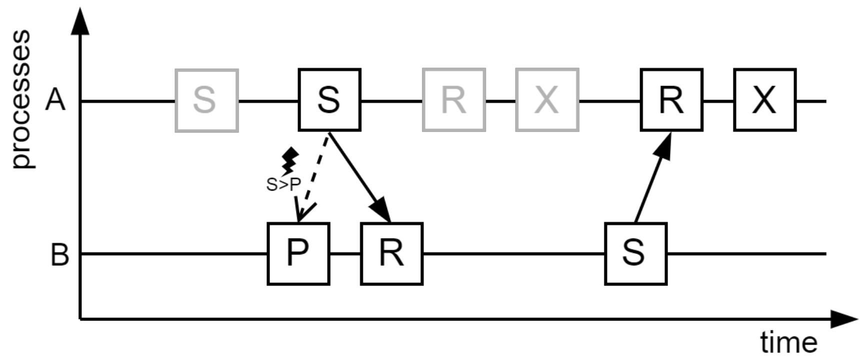


Figure 13: *Illustration of a Clock Condition Violation that is caused by Backward Amortization.*

The backward amortization is performed to restore the length of local intervals between events. In this example, the send event on rank A would be shifted. As this takes place during a backward replay, the timestamp of the receive event on rank B is received when the send event on rank A is encountered. To avoid new clock condition violations, the shift of the send event is limited to the received timestamp. The current approach does not handle probes in the backward amortization, which could result in a new clock condition violation. As illustrated in Figure 13, the limiting factor for the send shift is the receive timestamp, not the probe timestamp. Consequently, in case of a probed receive, the timestamp of the probe must be transferred instead of that of the receive event. In order to implement this, it is necessary to determine whether a receive is a probed receive at the time the receive event is encountered. The application of the same strategy as in the forward replay will not yield the desired result, as a probe must be followed by a receive, but the converse holds not true. Furthermore, a message can be probed multiple times, which could potentially lead to complications when traversing the trace backward, as the last probe, which is the first in time, would be relevant. However, the information regarding the existence of an additional probe for that message is not known in advance. It thus follows that the information pertaining to whether a receive is probed and whether a probe is synchronizing must be known prior to the backward replay. Two options are considered. In the forward replay, the correlation of probes and receives can be stored and subsequently reused in backward replay. Scalasca provides a unique identifier for each event, which could be used to create a map where the receive event ID is the key and the relevant probe ID is the value. This would necessitate the aforementioned struct being stored not as a set but as a map, where the struct is the key and the probes event ID is the value. Storing the set beyond the runtime of the forward replay and preventing the deletion of elements does not serve the desired purpose, as it does not address the issue of probe events occurring with the same arguments and being associated with different receives over the entire runtime. Another possibility is to move this matching to Scalasca's preprocessing step. This would require to store the event ID of the associated event within the event representation itself, thus enabling access during the replays. In the case of a probe event, the addition of a further member to the event representation would not be an issue, given that almost every probe is associated with a receive. Only those that are repeated probes for the same message should be ignored, and the variable can be assigned a specific value indicating this. In the case of receive events, the additional parameter can be considered unfavorable. There are numerous applications that do not utilize message probing. The presence of a default value for the parameter would indicate that the receive has not been probed. Providing that parameter would necessitate storage, independent of its usage. To address this issue, it can be considered to introduce a specific event representation for probed receives. While reading the trace data into the main memory, a receive event can be classified whether it is probed and Scalasca would represent the event internally as a probed receive containing the additional parameter. Given that the required storage space is that of an integer and that the preprocessing approach would allow a single matching process, rather than one occurring during each replay, this approach has been implemented in this thesis. Since the proposal with internally different event representations would require additional implementation effort, it has not yet been realized and is left for future work.

Because of the related event IDs present in the event representations, handling probes in the timestamp correction can be simplified. During forward amortization, no additional information has to be stored. When a probe callback is invoked, it is checked whether the receive ID accessible in the event representation of the probe is valid (e.g., not the special value for repeated probes). In case of a valid ID, a receive is invoked, and a correction takes place in case of a violation. Otherwise, the callback returns immediately. The same process is followed when a receive callback is invoked. In case of a valid probe ID, the callback returns immediately and otherwise the callback code is executed. A similar method is used during backward amortization. Since the receive is encountered first, there the probe ID is checked. In case of a probed receive, no message is sent. When a probe with a valid receive ID is encountered, its timestamp is sent.

Implementation of Event Matching

In the preprocessing, the matching of probes and receives can be performed during a forward pass of the local trace. Whenever a probe event is encountered, it must be stored in the case that no other pending probe exists with identical parameters. Accordingly, the aforementioned structure, comprising the message envelope and the message handle ID, is adequate for reuse as a key in a map, with the event ID of the probe serving as the value. On the occurrence of a matching instance of this struct already present in that map, the probe is not stored, as it is a repeated probe for the same message. Upon encountering a match event, its message handle ID is stored in a process-local variable that is initialized with a default value. When a receive event occurs, it is checked whether an instance of a struct containing the message envelope along with the message handle ID obtained from the process-local variable, serves as a key in the map. In this case, a match is identified, and the probe event ID is stored in the event representation of the receive. Likewise, the ID of the current receive event is stored in the probe event representation, which can be accessed via its unique identifier. The previously used entry is removed from the map and the process-local variable is set to its default value.

3.3 Wait State Analysis

The wait state analysis takes place during the first two replays of the analysis phase, where the first replay, a forward pass, is used to determine *Late Sender* wait states. The subsequent backward replay is necessary to detect *Late Receiver* wait states, and for *Late Sender* instances the synchronization point information is transferred back to the peer process. A probe call can only reveal *Late Sender* waiting time. However, as shown in Figure 3b it is possible to have a *Late Sender* and a *Late*

Receiver wait state in the same communication context. The current analysis already detects such a *Late Receiver* wait state in the backward pass, as the synchronization only affects the send and the receive. The analysis must therefore be extended to include *Late Sender* wait states in probe regions and to classify these instances, determining whether they correspond to a *Late Sender Wrong Order* situation. It is first evaluated how probe events can be integrated into the analysis to identify *Late Sender* wait states before considering how to detect wrong order situations.
Late Sender wait states are detected during a forward replay. First, a callback function must be registered that is invoked whenever a probe event is encountered during the first forward pass. In essence, the information required by the probe is identical to that which the receiver utilizes to identify a *Late Sender*: The event data of the corresponding send. Figure 14 illustrates the communication replay to recognize a *Late Sender*. Rank A encounters a send event in the trace and sends its event data, along with the event data from the enter and leave events, to B. On rank B a probe event occurs and the corresponding callback function is invoked. Here, it must be checked whether the given probe has a valid receive event ID. This is to ensure that it is not a repeated probe for the same message, and therefore it possibly includes waiting time. In the example depicted below, the ID is valid. Consequently, a receive operation is performed inside the callback function to obtain the message data. The algorithm to detect a *Late Sender* instance in a receive region can be reused to detect them in a probe region. The waiting time is calculated as the time difference between the timestamp of the enter event of the probe region and that of the remote send enter event. In case the waiting time is greater than zero, a *Late Sender* wait state is detected.

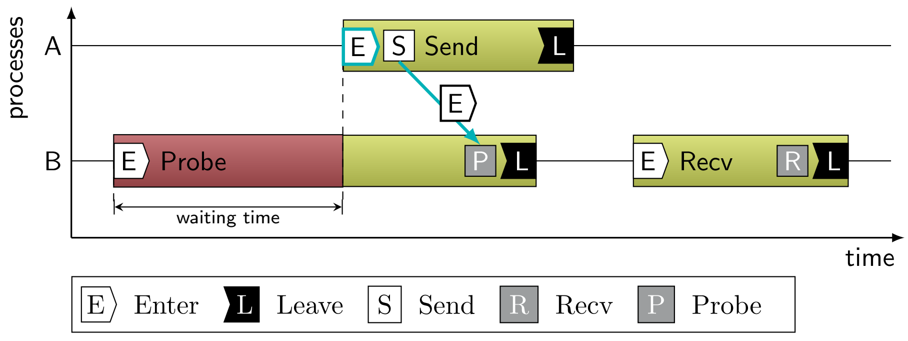


Figure 14: *Illustration of the Communication Replay for the Late Sender Wait State in Probe.*

It is not possible for a probed receive to have a *Late Sender*, as the return of the probe ensures that the send operation on the peer process was initiated. In the context of the wait state analysis, which identifies instances of *Late Sender* wait states, such a receive may be skipped. However, to avoid any potential interference with other analyses of inefficiency patterns that may be triggered when a receive event is encountered, like the classification of *Late Sender Wrong Order situations*, it is ensured that the probed receive event has access to the event data received in the probe. As a consequence of the callback infrastructure, the data is only accessible within the context of the same callback. One way to ensure the accessibility would be to resend the message to the same rank, as MPI allows the source and destination rank of a message to be identical. However, when encountering the receive event, this would result in a mismatch as the receive operation posted there expects a message from the original source of that message. The use of a wildcard parameter for the source rank in this context could potentially lead to unexpected behavior in the replay if other messages are sent that randomly match such a receive. Using the probe event ID available in a receive event would allow distinguishing which rank has to be specified in the receive operation, depending on whether the ID is valid. However, re-sending messages within the same process does not appear to be a favorable solution, which is why this approach has not been pursued further. The approach implemented in this work employs a data structure to store the aforementioned information at a higher level until the receive event occurs. A struct is defined that holds the event information from the send region, including the enter, send, and leave events. When a message is received in the callback function of a probe, such a struct instance is created and put into a map with the receive event ID as a key. Upon encountering a receive event, its event ID is searched for in the aforementioned map. Based on the result of the lookup, the event data from the related send is either extracted from that entry or a receive operation is invoked, as the data were not yet obtained. Upon reaching a receive event, the timestamp of that event is compared to that of the *Late Sender* instances in the buffer to ascertain whether an instance is associated with a wrong order situation.

Although the aforementioned example illustrates the detection of a *Late Sender* in probe calls, the detection in matching probe calls functions identically, as the same event sequence is considered. While there is also a match event in the receive region, this is solely utilized in the preprocessing step and is disregarded in the described replay.

It is now considered which additional steps are necessary to identify a *Late Sender Wrong Order* wait state in a probe. The current analysis determines a wrong order situation in a receive region by storing the *Late Sender* occurrences in a ring buffer. When a receive event is encountered, the received timestamp—corresponding to the send event—is compared with the send timestamps accessible via the *Late Sender* instances stored within the ring buffer. The *Late Sender* instances where the send enter timestamp is greater than the timestamp of the send enter event corresponding to the current receive event are classified as a wrong order situation. A probe exhibiting a *Late Sender* wait state can correlate with a wrong order situation irrespective of whether the message that may be received earlier is probed or not. Figure 15 illustrates the aforementioned situation. Process C probes for the message from rank A and reveals a *Late Sender* wait state. This wait state is calculated during the replay where the event data from the send event are received when the probe event is encountered. When the subsequent receive event in the region Recv1 is reached, no message is received, but the event data from the corresponding send can be accessed through the data structure described above. When the analysis rank responsible for rank C encounters the following receive event, it obtains the event data related to the send operation on rank B. In the case the previously detected *Late Sender* is buffered, the enter timestamp of *Send1* can be compared with the enter timestamp of *Send2*. The latter one is accessible via the stored *Late Sender* instance, and the wait state could be classified a wrong order situation. Scalasca's callback infrastructure already invokes a callback whenever a *Late Sender* instance is detected, and that instance is added to the aforementioned ring buffer. It thus follows that the *Late Sender* instances revealed in probe calls are put into the same ring buffer as those identified in a receive, which enables the classification of wrong order situations. While it would be possible to classify a previously detected *Late Sender* as a wrong order situation not only when a receive event occurs, but already when a probe event is encountered, this would require another distinction between probed and regular receives. Instead, it is sufficient to only check for wrong order instances when receive events are encountered, which is favorable for reasons of consistency, and therefore implemented in this thesis.

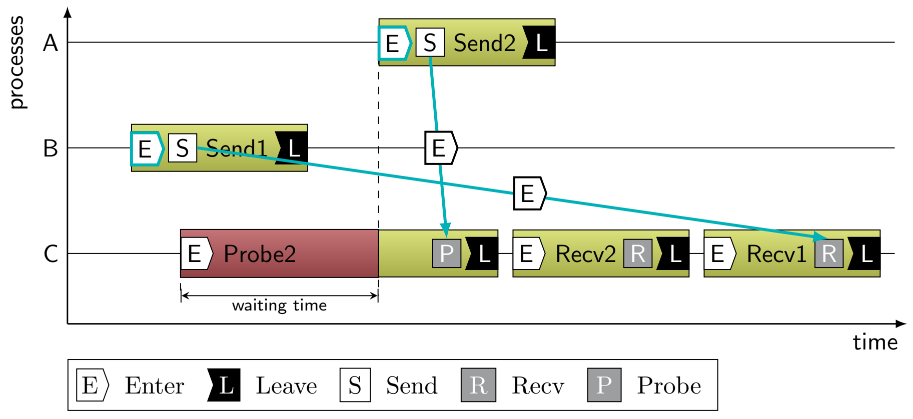


Figure 15: *Illustration of the Communication Replay for the Late Sender Wrong Order Wait State in Probe.*

3.4 Delay Analysis

In examining the delay analysis for probes, it is essential to differentiate between two scenarios: The first, a simple *Late Sender* case in which the send and the probe are synchronizing, and the second, a case involving a simultaneous *Late Receiver*, in which the send has two synchronization points, one with the probe and one with the receive. Since the implementation of the first scenario is relatively straightforward, it is considered first.

As the delay analysis takes place in a backward replay, the trace is traversed from back to front. When a receive event is encountered, it is checked whether it reveals a *Late Sender* wait state and associated delay costs for the wait state must be calculated. In this case, the synchronization interval is determined, using the synchronization point information that was stored during the former replays. Subsequently, the time vector for the specified interval is calculated and stored into the message data, as well as the waiting time. The initiation of a send operation to transfer the entire message from the receive event occurs regardless of whether or not a *Late Sender* wait state is present. When the analysis process responsible for the peer rank encounters the send event, the aforementioned event data is received. Subsequently, the synchronization point information is used to determine, whether a synchronization point but no waiting time is stored, thus indicating a *Late Sender* wait state. In case such a synchronization point is stored, the time vector and the waiting time is extracted from the remote event data. Only in case there was a *Late Sender*, the previous synchronization point (in time) is determined. A delay vector containing the execution time differences except waiting time as explained in Section 2.2.2 is calculated, and the waiting time can be mapped to short- and long-term delay costs. This method can be adopted for the probes. Here, once the local time vector has been calculated, the message data from the probe must be transferred to the process where the corresponding send event occurs. As the critical path analysis (see Section 2.2.2) also takes place during this replay phase, the data from the receive event are relevant from the sender's perspective. Therefore, both the probe and the receive event data should be transmitted. It is not possible for the sender to ascertain whether a related receive was probed. In the original algorithm, a single receive call is posted upon the occurrence of the send event. In the backward traversal the receive event is encountered before the probe event, and the information whether it is a probed receive is available. Two strategies are conceivable. In case of a probed receive, the message data from the receive event is stored in an internal data structure until the related probe is encountered. At this point, a single message is sent containing both the event data from the receive and the probe. An alternative approach would be to send individual messages when the events are encountered, but to include the information as to whether the receive is probed in the transferred message data from the receive, as this message is sent first. Once more, it is essential to verify the ID in the receive event to prevent the handling of repeated probes. On the rank where the send event occurs, the message data from the receive event is retrieved first. This allows to ascertain whether it is a probed receive. If so, another receive operation is initiated to obtain the message data sent by the probe. Subsequently, the algorithm can be employed to calculate the delays and related costs. In the case of matching probes, a problem arises when multiple matching probes are followed by the same number of matched receives in an arbitrary order, all of which have the same envelope. Due to the message handle ID, the probe events are correlated to the correct receive event. However, the messages sent during the replay will not match as desired. The MPI standard ensures that pending messages with the same envelope are received in the order they were sent. Consequently, when the last of the probe events (in time) is encountered during the backward replay, a message is sent which indicates that it was a probed receive. Accordingly, when this message is received upon encountering the corresponding send event on the peer process, a second receive operation is invoked. However, instead of receiving the message data from the receive event that matches the probe, the message data from the second to the last probe event would be received. In the replay, all messages for the matching probes are sent before those for the receives, reflecting the order in which the events are encountered. To address this issue, different communicators for the messages sent for probe events and those sent for receive events can be utilized. Scalasca already provides the infrastructure for this by storing duplicated communicators for those communicators used in point-to point-communication. These duplicated communicators can be used for this purpose. In this thesis, the approach of sending individual messages when the probe and receive events are encountered was implemented.

The situation becomes somewhat more complex when two synchronization points are present in the send. Figure 16 shows a *Late Sender* and *Late Receiver* wait state in the same communication context. For the sake of simplicity, only the communication-specific events are shown in this timeline diagram and the enter and exit events are not depicted. The communication during the execution of the application is illustrated with solid arrows, whereas the communication during the current backward pass is illustrated with dashed arrows. During the previous communication replays, the depicted wait states were identified and related synchronization point information was stored. Upon storing the related synchronization point information, the first problem is encountered. As explained before, the synchronization point is stored when encountering a wait state, and it is transferred to the peer process in the subsequent replay. Two maps are utilized for the purpose of storing the synchronization point information. The *synchpoint map* contains the event where the synchronization point occurs as a key with the value being a struct of data about the wait state. This includes for the waiting time. The second map, the *synchranks map*, as well contains the event as the key and a set of numbers of the synchronizing ranks as the value. The rank numbers refer to the rank of the corresponding process when using a global communicator. In the case of point-to-point communication, the set comprises only a single rank number, that of the peer process.

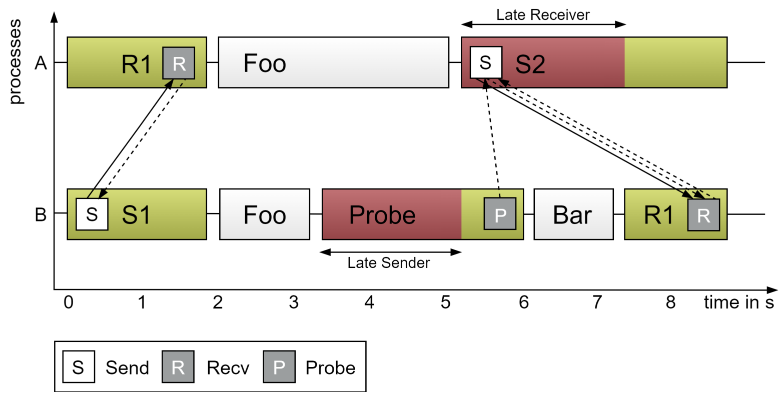


Figure 16: *Illustration of the Communication Replay for the Delay Analysis.*

Since the *Late Sender* wait state is detected in the first replay, a synchronization point on rank B is stored by inserting a pair consisting of the probe event and the wait state information into the *synchpoint map*, as well as another pair comprising the probe event and the rank number of process A into the *synchranks map*. In the subsequent replay, the trace is traversed backwards and the *Late Receiver* wait state is identified when process A reaches the send event and obtains the event data from the receive event sent by rank B. As the received event data indicates that the receive was probed, a subsequent receive operation is initiated to obtain the event data from the probe. At the time the *Late Receiver* is detected, a synchronization point is stored on process A. This entails the incorporation of the send event and wait state data into the *synchpoint map*, along with the send event and the rank number of process B into the *synchranks map*. Next, the message data from the probe is received, which transfers the synchronization point information of the *Late Sender* wait state. This would lead process A to store a pair consisting of the send event and the wait state information into the *synchpoint map*. The waiting time stored would be zero, as the waiting time occurred on the peer process. Another pair comprising the send event and the rank number of process B would be inserted into the *synchranks map*. Since both maps already contain the send event as a key due to the *Late Receiver* wait state, the insertion would fail, and the maps would remain unchanged[1]. Simply storing one synchronization point in the send event,

[1]This behavior is due to the use of a C++ STL map. However, overwriting the existing data would not solve the underlying problem either.

would lead to erroneous behavior. The delay analysis functions in a manner whereby the time vector between the current and the previous synchronization point (in time) is calculated and transferred to the process that causes a wait state. In case of *Late Sender* wait states, this entails the transfer of the time vector from the receiver's rank, where the wait state originates, to the sender's process, which is responsible for that particular wait state. This allows for the calculation of the delay vector. Figure 16 illustrates this, wherein the dashed arrow indicates that the probe event triggers a message to be sent to process A, where the send event should trigger a corresponding receive operation.

In case of *Late Receiver* instances, the wait state occurs on the side of the sender. This indicates that the receiver's process is responsible for calculating the delay. Consequently, a message is sent when a send event indicates a *Late Receiver* wait state. This message must be received when the corresponding receive event occurs on the peer process. Here, the synchronization point information is employed to ascertain whether a synchronization point is the counterpart of a *Late Receiver*. This wait state is indicated by an entry of the send event in the *synchpoint map*, where the waiting time is zero. Only in case of a *Late Receiver* this message exchange takes place. With regard to the backward replay, the message is transmitted in reversed direction. As illustrated in Figure 16, this is depicted by the dashed arrows originating from *S2* and terminating at *R1*, as well as from *R1* extending to *S2*. It is imperative to exercise caution when transmitting messages between two processes in opposite directions to prevent the occurrence of deadlocks. Returning to the actual problem, the analysis rank responsible for process B would encounter the receive event in region *R1*. It would then search for the previous synchronization point on its rank, which occurs in the probe event. The time vector would be calculated, containing the execution time of routine Bar, and subsequently the vector would be transferred to rank A. Rank A also determines its previous synchronization point. This occurs in routine *R1* and the time vector would contain the execution time of routine Foo. The local time vector would be transferred back to rank A, due to the *Late Receiver* wait state. The interval between *R1* and *S2* would be incorrectly mapped to the interval between *Probe* and *R1*. The interval that should actually be mapped to the interval between *Probe* and *R1* would be empty, as both are synchronized with *S2*. This demonstrates that the existing analysis is unable to accommodate such scenarios, as it would lead to a discrepancy in the synchronization intervals on the sender's side, resulting in an erroneous mapping of the delays. For the current analysis to be effective, it is essential that synchronization intervals containing the same processes do not overlap. The analysis relies on the assumption that an event can only participate in one synchronization point. However, in the case of probes, this is not always the case. The development of an enhanced analysis that can handle this case is beyond the scope of this thesis.

In order to enable the delay analysis for at least one of the two wait states, this work excludes *Late Receivers* in that case, as *Late Senders* are much more common in practice, and thus have a higher impact. As a side effect of this, the implementation also becomes more straightforward. In the second replay the *Late Receiver* wait states are detected, and the synchronization point information of the *Late Sender* instances is transferred to the sender side. During the backward replay it must be ensured that the receive operation, triggered by the occurrence of a send event, obtains the data from the receive event. This allows to be aware whether the receive was probed and therefore a second receive operation must be initiated to receive the message data from the probe. when the message data from the receive event are retrieved, the *Late Receiver* wait state is identified, and its synchronization information stored prior to the reception of the *Late Sender* synchronization point information transmitted during the probe. Before storing the *Late Sender* synchronization point information, it is possible to ascertain whether the send event is already part of the *synchpoint map*. In such a case, and if the value for the related waiting time is greater zero, a *Late Receiver* would be indicated. The *Late Receiver* synchronization point would be deleted, and the *Late Sender* synchronization point stored instead. This approach permits the analysis of delays for the *Late Sender* and, at the same time, the detection of *Late Receiver* instances. However, due to the absence of synchronization point information in concurrent *Late Receiver* wait states, these are disregarded in the delay analysis. Considering the *Late Receiver* instead of the *Late Sender* in the delay analysis would be far more complicated, as not only the received *Late Sender* synchronization point must not be stored. Moreover, on the peer process where that wait state occurred, the synchronization point must also be deleted to ensure the accurate mapping of the synchronization intervals. This implies that during the subsequent replay, the synchronization point data for *Late Receivers* must be transferred to the receiver's side, along with the information to delete a synchronization point in the probe event that was stored for a *Late Sender* instance. As previously stated, this thesis implemented the first approach.

3.5 Critical Path Analysis

The critical path analysis is conducted concurrently with the delay analysis during the same replay. As discussed in the previous Section, a *Late Receiver* instance occurring in the same communication context as a *Late Sender* is not considered in the delay analysis. Consequently, they cannot be taken into account in the critical path analysis. Section 2.2.2 explained that the critical path switches to the rank responsible for a wait state whenever a synchronization point is encountered. This signifies the rank which exhibits the delay. However, in the absence of synchronization point data for the aforementioned instances of the *Late Receiver*, the critical path would remain on the respective rank, potentially resulting in the inclusion of waiting time, which is inconsistent with the definition of the critical path. It is therefore necessary to take this knowledge into account when considering this metric. Apart from that, the algorithm can easily be extended to handle probe events. The backward replay already exchanges messages correctly for the delay analysis, and the critical path analysis takes place in the same replay. Therefore, it is sufficient to add a flag indicating whether the ownership of the critical path should be transitioned to the peer process to the message data that is transferred to the corresponding process.

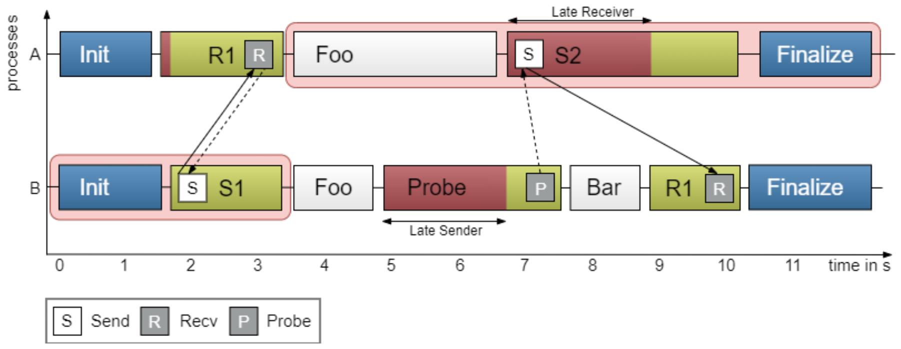


Figure 17: *Illustration of Critical Path Detection.*

Figure 17 illustrates the detection of the critical path in case of a *Late Sender* and a *Late Receiver* wait state in the same communication context. The red background denotes the detected path course, the solid arrows indicate the direction of communication during the execution of the program, and the dashed arrows denote the relevant point-to-point communication for the critical path analysis. To enhance the clarity of the illustration, only point-to-point specific events are depicted; those pertaining to collective operations or enter and leave events are not included. Such a collective event would be encountered in the *Finalize* region, where a synchronization takes place to ensure that all MPI processes exit the application simultaneously. This is replayed by a collective operation that determines the rank which enters the finalize call last, and therefore marks the endpoint of the critical path. In Figure 17 this results in the analysis process responsible for rank A setting the ownership flag for the critical path. When encountering the probe event on rank B, a flag is sent indicating whether the peer process should assume the possession of the critical path. As the current owner of the critical path is rank A, the aforementioned flag is false, and the critical path remains on rank A. Upon encountering the receive event on process A, the flag is set to true, as routine *R1* exhibits a wait state due to a delayed sender. This information is received by process B at the time the send event in region *S1* occurs. Consequently, process B sets the ownership flag and the critical path continues on that rank. Due to the missing *Late Receiver* synchronization point, there is no message exchange from the send event in *S2* to the receive event in *R1*. Therefore, the necessity of transferring the critical path ownership cannot be accomplished. At this juncture, the actual critical path would transition to process B and revert to rank A upon encountering the probe event, given the presence of waiting time. If process B had been the last to enter the finalize call, the critical path would have been determined correctly. In this scenario, the endpoint of the critical path is at rank B, and thus the ownership flag would be set on rank B at the beginning of the analysis. The next communication takes place when the probe event is encountered, and here the critical path would be transferred to rank A. The remainder of the critical path analysis would follow the aforementioned description. As shown in the example, the detected critical path is not entirely correct. But for the most common use case of probes to determine the required message buffer size of the receive usually following shortly after, the error in the critical-path profile is rather small. In addition, having both a *Late Sender* and a *Late Receiver* in a communication context is a corner case, and thus the presented approach seems to be a reasonable compromise to at least approximate the critical path until the infrastructure is fixed to handle two synchronization points for one event. This is left for future work.

4 Evaluation

The analyses for probing messages presented in the previous chapter were prototypically implemented in the Scalasca analyzer. This chapter presents an evaluation of the implementation. First, a brief overview of the experimental setup is provided, followed by the presentation of representative microbenchmarks that illustrate the functionality of the probe handling. Subsequently, the real-world application ParFlow is analyzed to demonstrate that the inefficiency patterns observed in probes are not merely theoretical, but rather manifest in actual applications.

4.1 Setup

All measurements presented in this chapter were conducted on the JURECA supercomputer [27], which is operated at the Jülich Supercomputing Center (JSC) using the standard compute nodes. The cluster provides 480 standard compute nodes. Each of these nodes is equipped with two AMD EPYC 7742 processors, each having 64 cores and running at 2.25 GHz, providing a total of 128 cores per node. These nodes contain 512 GB of DDR4 (Double Data Rate 4) memory, configured in 16 modules of 32 GB. Networking is provided by InfiniBand HDR100 (NVIDIA Mellanox Connect-X6). The software environment included the Intel compiler, version 2022.1.0, and ParaStationMPI, version 5.7.0, which offers an implementation of the MPI standard. The SLURM job scheduler (Simple Linux Utility for Resource Management) was utilized to submit jobs to the system, and therefore execute the measurements.

4.2 Functional Evaluation

In order to demonstrate the functionality of the prototype implementation, different test cases are considered. These include tests demonstrating the functionality of the preprocessing matching, the timestamp correction and the analysis of *Late Sender* wait states in probes. The *Late Sender* test cases also consider a wrong order situation, as well as the situation where a *Late Receiver* in a send and a *Late Sender* in a probe occur at the same time. The test codes are executed a single time to generate the OTF2 trace data with a development version of Score-P[1] that includes the probe and match events. For specific tests, the same trace data is analyzed with the Scalasca master branch available at the time of writing[2] and with a Scalasca

[1] https://gitlab.jsc.fz-juelich.de/perftools/scorep/-/commit/a518a74d859cdb324cb130e9decb61966e1dc525 [only accessible for authorized accounts]

[2] https://gitlab.jsc.fz-juelich.de/perftools/scalasca/-/commits/f42a9aa632594f1fdf0d283e1399a86aa2db11bc [only accessible for authorized accounts]

version including the prototype implementation[3] of the probe handling in order to compare the results and to see whether they match the expectations based on the trace data. In the following, the test cases are visualized in the form of timeline diagrams. The code can be found in the Appendix.

4.2.1 Test Case: Preprocessing Matching

To demonstrate the correct matching of probes and receive events during the preprocessing, this test case includes matching probe calls and repeated probe calls. Figure 18 illustrates process A sending two messages to process B. B probes for both of the messages, but for the first one with a matched probe. Subsequently, the second message is received prior to the first. B then sends a message to rank A, and A probes for this message two times before receiving it. The blue connections indicate the expected matching.

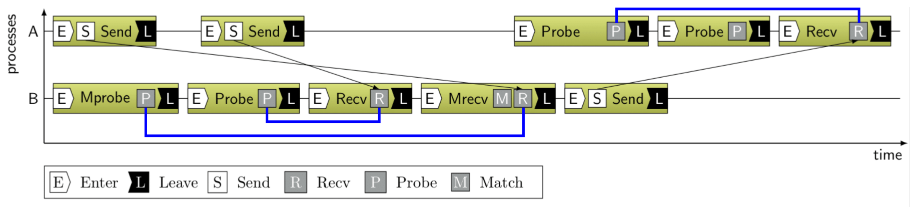


Figure 18: *Test Case: Preprocessing Matching.*

For verification, the test case was executed with debug output containing the matching event IDs and the expected matching was confirmed. The repeated probe on rank A was assigned a special value, indicating that it can be ignored in the analyses.

4.2.2 Test Case: Clock Condition Violation in Send-Probe Pair

The functionality of the timestamp correction cannot be illustrated by simply running a test code, as a clock condition violation must be present. These violations arise during the measurement due to the use of process local clocks. For the purpose of demonstrating the implemented functionality, the timestamps in a measured trace were manipulated to enforce such a violation. The timeline diagram in Figure 19 depicts the time course of the events after manipulation.
The timestamp of the probe event on rank A is prior to that of the send event on process B. This represents a violation of the clock condition. As the Scalasca master branch does not take probes into account, this violation remains undetected. In the example illustrated in Figure 19 the algorithm only checks whether the clock condition that the send timestamp must be prior to that of the receive holds true.

[3]https://gitlab.jsc.fz-juelich.de/perftools/scalasca/-/tree/79-properly-handle-probing-of-messages?ref_type=heads [only accessible for authorized accounts]

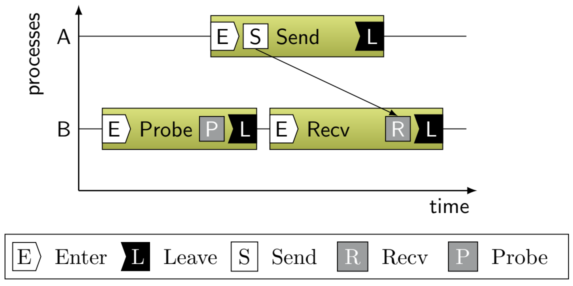


Figure 19: *Test Case: Clock Condition Violation in Send-Probe Pair.*

When applying the timestamp correction from the prototype implementation, the depicted clock condition violation was corrected. This shows the correctness of the implemented handling of probes in the forward amortization, since the events on process B were shifted forward in time to guarantee the probe event occurs after the send event.

4.2.3 Test Case: Clock Condition Violation in Send-Receive Pair

To demonstrate that the handling of probe in the backward amortization works as well, the test case illustrated in Figure 12a was utilized. Similar to the previous test case, measured trace data was manipulated to exhibit the desired clock condition violation. As explained in Section 3.2 the violation occurs in a send-receive pair. When using the Scalasca master branch, this violation is fixed, but during the backward amortization, a new violation in the preceding send-probe pair is introduced. The execution of the timestamp correction, which includes the probe handling implementation, corrected the violation in the send-receive pair as well. In addition, no violation in the send-probe pair was created during the backward amortization due to the fact that the algorithm transferred the probe timestamp, which marks the earliest point in time until which the send event may be moved, to the peer process.

4.2.4 Test Case: Late Sender

The timeline diagram depicted in Figure 20 illustrates a sequence of events obtained from the trace data generated during the execution of the code in Listing 2 on three processes. It should be noted that the timeline diagram does not reflect the actual execution times, as the send and receive operations consumed only a few milliseconds. However, to be able to display them, the operations appear longer than they actually lasted. The *Late Sender* wait states are enforced by invoking a sleep function. Process A remains two seconds in the *Sleep* region, and process B one second. Consequently, rank B spends two seconds in the *Probe* region, and process C spends three seconds in that region. Therefore, the expected waiting time in the

Probe of rank B corresponds to two seconds. They are expected to be short-term costs of the delay on rank A, that is revealed in the *Sleep* region. On rank C three seconds of waiting time are expected in the *Probe* region. Two second of the waiting time are anticipated to be long-term costs of the delay on rank A. The remaining second is expected to be short-term costs of the delay caused by the *Sleep* region on process B. Since process C is the last to enter the finalize call it is therefore expected to be the endpoint of the critical path. Process A remains one second in *Finalize* whereas process B and C spend almost no time in that call. Due to the synchronization happening in a finalize call, a second of waiting time on rank A can be expected here, which are long term costs of the delays on rank A and B. The critical path is anticipated to mainly hold the times spent in the *Sleep* regions. It is expected to start on process A and be transferred to process B when the send event is encountered. It should transition to process C when the next send event is encountered, and remains there for the remainder of the execution.

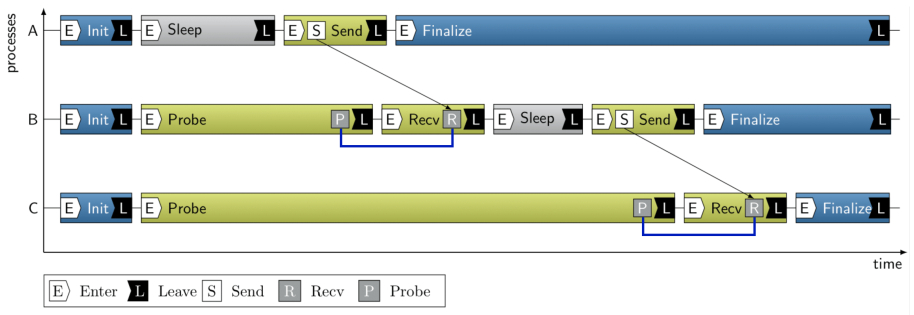


Figure 20: *Test Case: Late Sender.*

When analyzing the trace data with the Scalasca master branch, no wait states except the one in the *Finalize* region on rank A were identified. The critical path remained on process C throughout the whole execution time, covering the time spent in the *Probe* region. The waiting time in the *Finalize* region was analyzed to be short-term costs of a delay detected in the *Probe* region. Using the prototype implementation, the expected *Late Sender* wait states were identified. Moreover, the critical path was detected correctly. It starts on rank A, transitions to rank B upon encountering the send event, and then progresses from rank B to rank C when the next send event is reached. Furthermore, the detected delay costs correspond to the expectations. The analysis identifies 0.33 seconds of the wait state in *Finalize* to be long-term costs of the delay in *Sleep* on rank B and the other 0.67 seconds being long-term costs of the delay on rank A.
When it comes to the identification of performance bottlenecks in this test case, relying on the analysis results obtained with the Scalasca master branch would be inadvisable, as it would lead to the assumption that the *Probe* region is the function

that contributed the most time to the critical path and is responsible for the wait state in *Finalize*. The analysis including the probe handling correctly identified the *Sleep* regions to be the reason for the wait states.

4.2.5 Test Case: Late Sender Wrong Order

In order to demonstrate a wrong order situation, this test case requires execution on at least three processes. The process with rank number zero within the global communicator is responsible for probing and receiving messages from the other processes. In Figure 21 this process corresponds to process A. The messages are probed and received in ascending order according to their source rank, starting with rank C. The message of rank B is received last and thus establishes the wrong order situation. All processes, with the exception of process A, spend the number of seconds corresponding to their rank in a sleep call before sending a message to rank A. For process B this time corresponds to two seconds, and three seconds for process C. Figure 21 illustrates the chronological event sequence for the execution on four processes.

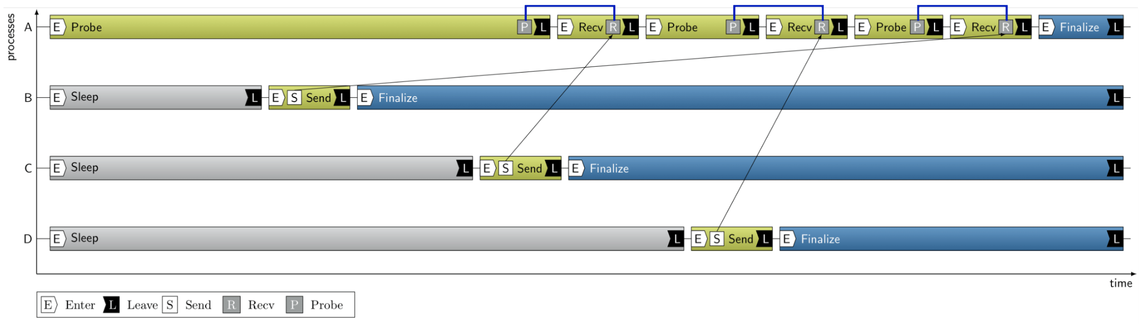


Figure 21: *Test Case: Late Sender Wrong Order.*

Rank A spends two seconds in the first *Probe* region and another second in the subsequent *Probe* region. Again, the time spent in the send and receive operations is insignificant. B remains in the *Sleep* region for one second, C for two, and D for three. Consequently, B spends two seconds in the *Finalize* region and C remains there for one second. A and D almost spent no time in there. The Scalasca master version identified the time spent in the *Finalize* region on rank B and C as waiting time. These are assigned to be short-term costs of a delay exhibited in the *Probe* region on process A. The critical path is detected on rank A for the entire execution. The *Probe* region represents the most significant contributor to the critical path and therefore lead to the assumption that improving the performance of the application is not possible, as probe is an MPI call. The analysis including the probe handling proves this assumption wrong. It identifies wait states within the first two *Probe* regions on rank A and classifies them as wrong order situations. The first one includes two seconds of waiting time and the following one second. The time spent in the *Sleep* region on rank C reflects the direct cause for the wait state in the first *Probe* region. One second of the time spent in the *Sleep* region on rank D is identified to be the delay causing the wait state in the second *Probe* region. The analysis as well detects three seconds of waiting time in the *Finalize* region. These are classified as long-term costs of the delay in the *Sleep* regions on rank C and D, where rank C is responsible for 1.33 seconds and rank C for 1.67 seconds. The critical path is identified to remain the whole time on process D. Accordingly, the *Sleep* region contributes the greatest amount of time to the critical path and is identified as a delay, which in turn indicates that it is the performance bottleneck.

4.2.6 Test Case: Late Sender and Late Receiver

The timeline diagram depicted in Figure 22 shows a simultaneous *Late Sender* and *Late Receiver* wait state. The trace data reveal that process A spends about 0.1 seconds in the *FillArray* routine, another second in the *Sleep1* region, and slightly more than two seconds in the *Send* routine. Process B spends about one second in the *Probe* region, two seconds in the *Sleep2* routine, and almost no time in the *Recv* region.

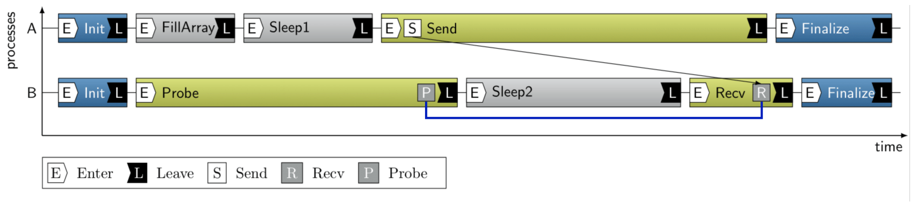


Figure 22: *Test Case: Late Sender and Late Receiver.*

First, the results obtained with the Scalasca master branch are examined. There, two seconds of *Late Receiver* waiting time are identified in the *Send* region. The delay analysis reveals 1.29 seconds of short-time delay costs in the *Sleep2* region and another 0.71 seconds in the *Probe* routine. The critical path is identified as remaining on rank B throughout the entire execution. When using the prototype implementation of the probe handling, the analysis results differ. The two seconds of *Late Receiver* waiting time are still identified, but moreover, an additional 1.1 seconds of *Late Sender* waiting time is identified within the *Probe* region. The detected *Late Sender* wait state has an impact on the critical path and the results of the delay analysis. Since the simultaneous *Late Receiver* wait states are not handled in the delay analysis, only the delay costs related to *Late Sender* wait state are calculated. These correspond to a second of short-term delay costs in routine *Sleep1*. The critical path concludes at rank B, where the *Finalize* routine is entered last. It contains the *Recv* and *Sleep2* regions on that rank and then transitions to rank A, because in the

Probe region waiting time occurred. On rank A the critical path covers the *Sleep1*, *FillArray* and *Init* regions. Despite the delay and critical path analysis ignoring the *Late Receiver* wait state, the critical path was correctly identified due to the fact that process B entered the finalize call last. The delay costs related to the *Late Sender* wait state are determined as one second in *Sleep1* and 0.1 seconds in the *FillArray* routine.

When taking a look at the delay analysis and critical path analysis results, the Scalasca master version identifies routine *Sleep2* as the primary candidate for improvement in runtime, but also it identifies the *Probe* routine to be the responsible for waiting time. The Scalasca analysis including the probe handling identifies *Sleep1* and *FillArray* to be responsible for waiting time. Furthermore, when taking a look at the critical path it becomes clear that *Sleep2* is another candidate for optimization. Accordingly, the performance bottlenecks could be more effectively identified with the probe handling, as the current Scalasca master version completely misses the delay in *Sleep1* and therefore detects an erroneous critical path.

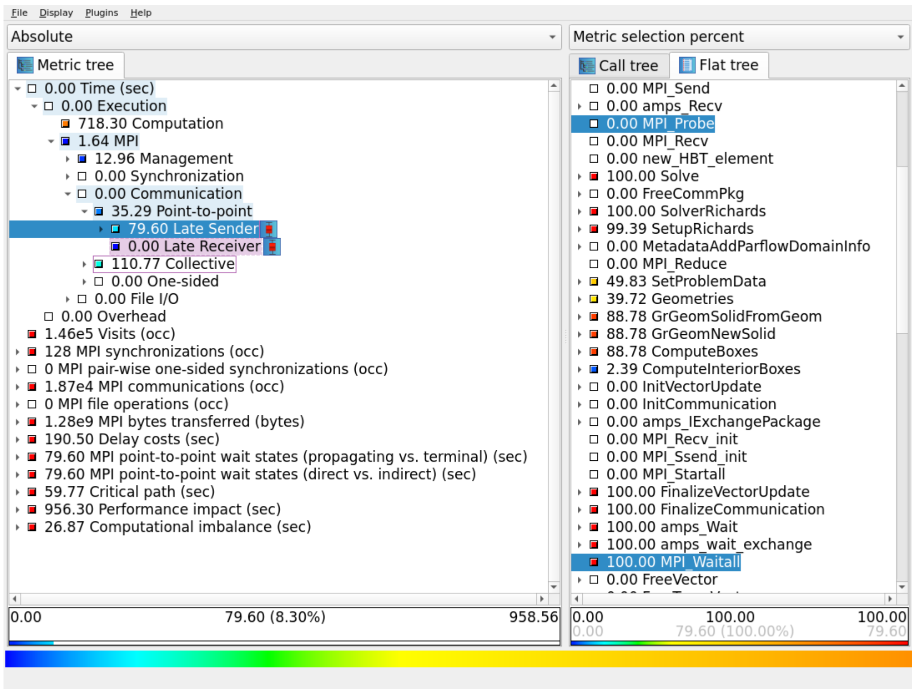
4.3 Real World Example

In order to demonstrate the additional value that probe handling offers in the analysis of production codes, this section will evaluate the analysis of the ParFlow application using Scalasca. ParFlow [28] is an open-source application that models the hydrologic cycle and simulates surface and subsurface flows. It uses numerical methods to solve the Richardson equation. ParFlow can be run in parallel using MPI. In the context of this work, ParFlow was executed with example input data on 16 processes distributed across four nodes. A filter was applied to ensure that Score-P only collected event data for call paths in which MPI communication occurred. This reduced both the measurement overhead and the amount of trace data, and facilitated the evaluation of the analysis results. The parallel execution time was 59.77 seconds, with each process spending an equivalent amount of time, resulting in a total execution time of 956.32 seconds.

The analysis results conducted with the Scalasca master branch will be presented first, and then compared to those obtained with the Scalasca version that includes the implemented probe handling. This comparison will allow for an evaluation of the new insights gained. The timestamp correction is performed prior to the various analyses and will therefore be considered first.

The Scalasca master version identified and corrected ten clock condition violations, whereas the version with probe handling detected and fixed 42 violations. Since both analyses utilized the same trace data, it can be assumed that the 32 additional violations occurred in send-probe pairs and were missed by the latest Scalasca version.

Figures 23 and 24 show screenshots of the analysis result visualizations in the Cube report browser. Cube has a three-dimensional structure comprising a metric dimension, a program dimension and a system dimension. The metric dimension includes metrics like time, delay costs and the critical path. Detected wait states are displayed as inner nodes in the time metric. The program dimension contains the call tree of the measured application. The tree is either displayed nested in the same way as the regions are nested in the trace data, or it is displayed in a flat hierarchy to be able to view the metrics data for a specific region in aggregated form. The metric values can be mapped to specific parts of the call tree to identify where it occurred. The colors next to the nodes indicate the degree of influence of the node. Blue represents a low level of influence, while red denotes a high level of influence. The course of the color scale is illustrated at the bottom, though it is truncated since the tree view for the system dimension (right of the call tree pane) is not shown. The program dimension contains the locations specified by the trace data. Except the case where the call tree is displayed flat, the dimensions are present in a tree structure. Collapsed nodes show inclusive values (i.e., the value of the node itself as well as the aggregated values of their child nodes), while expanded nodes show exclusive values and the distribution across child nodes. This hierarchy is depicted in Figure 23 on the example of the time metric.



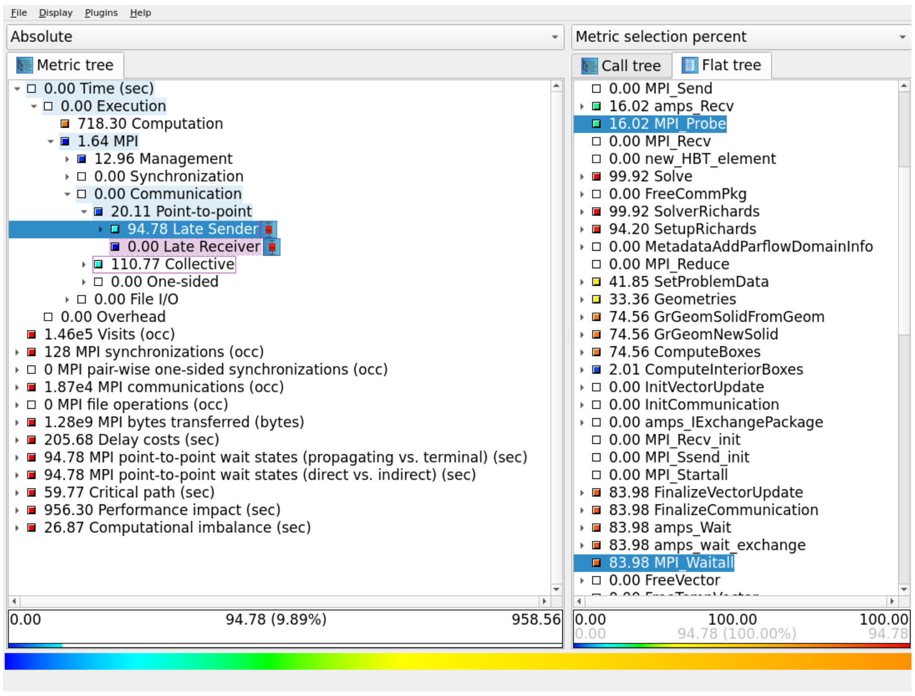
```
0.00 Time (sec)
  0.00 Execution
    718.30 Computation
    1.64 MPI
      12.96 Management
      0.00 Synchronization
      0.00 Communication
        35.29 Point-to-point
          79.60 Late Sender
          0.00 Late Receiver
        110.77 Collective
        0.00 One-sided
      0.00 File I/O
  0.00 Overhead
1.46e5 Visits (occ)
128 MPI synchronizations (occ)
0 MPI pair-wise one-sided synchronizations (occ)
1.87e4 MPI communications (occ)
0 MPI file operations (occ)
1.28e9 MPI bytes transferred (bytes)
190.50 Delay costs (sec)
79.60 MPI point-to-point wait states (propagating vs. terminal) (sec)
79.60 MPI point-to-point wait states (direct vs. indirect) (sec)
59.77 Critical path (sec)
956.30 Performance impact (sec)
26.87 Computational imbalance (sec)

0.00                79.60 (8.30%)                958.56
```

```
0.00 MPI_Send
0.00 amps_Recv
0.00 MPI_Probe
0.00 MPI_Recv
0.00 new_HBT_element
100.00 Solve
0.00 FreeCommPkg
100.00 SolverRichards
99.39 SetupRichards
0.00 MetadataAddParflowDomainInfo
0.00 MPI_Reduce
49.83 SetProblemData
39.72 Geometries
88.78 GrGeomSolidFromGeom
88.78 GrGeomNewSolid
88.78 ComputeBoxes
2.39 ComputeInteriorBoxes
0.00 InitVectorUpdate
0.00 InitCommunication
0.00 amps_IExchangePackage
0.00 MPI_Recv_init
0.00 MPI_Ssend_init
0.00 MPI_Startall
100.00 FinalizeVectorUpdate
100.00 FinalizeCommunication
100.00 amps_Wait
100.00 amps_wait_exchange
100.00 MPI_Waitall
0.00 FreeVector

0.00           100.00           100.00
0.00      79.60 (100.00%)        79.60
```

Figure 23: *Cube Visualization of Wait States in ParFlow with Scalasca Master.*

The Figure shows the metric and program dimensions of the results obtained with the Scalasca master version. The execution time is divided into computation and MPI time, which is split further into several sub-metrics. In the time metric, the *Late*

Sender and *Late Receiver* wait states are listed as child nodes of the time spent in point to point communication. It can be seen, Scalasca master detected 79.6 seconds of *Late Sender* waiting time, which is classified as 66.53 seconds of standard *Late Sender* instances and 13.07 seconds of wrong order situations (not shown). Only a minimal amount of *Late Receiver* wait states were detected. The call tree is shown in a flat hierarchy to display the waiting time accumulated for a specific region. The values displayed in the flat tree indicate the percentage of time dedicated to the selected metric. In Figure 23 it is depicted that 100% of the *Late Sender* waiting time was spent in MPI_Waitall calls. MPI_Waitall is a blocking call which is passed a list of request objects that identify operations initiated with a non-blocking call. During the MPI_Waitall call, the invoking process remains in the call until all related operations are completed. The identification of *Late Sender* wait states in these calls indicates that in the calls in question the operation that was completed last, and therefore is responsible for the observed waiting time, was a receive.



```
0.00 Time (sec)
  0.00 Execution
    718.30 Computation
    1.64 MPI
      12.96 Management
      0.00 Synchronization
      0.00 Communication
        20.11 Point-to-point
          94.78 Late Sender
          0.00 Late Receiver
        110.77 Collective
        0.00 One-sided
      0.00 File I/O
  0.00 Overhead
1.46e5 Visits (occ)
128 MPI synchronizations (occ)
0 MPI pair-wise one-sided synchronizations (occ)
1.87e4 MPI communications (occ)
0 MPI file operations (occ)
1.28e9 MPI bytes transferred (bytes)
205.68 Delay costs (sec)
94.78 MPI point-to-point wait states (propagating vs. terminal) (sec)
94.78 MPI point-to-point wait states (direct vs. indirect) (sec)
59.77 Critical path (sec)
956.30 Performance impact (sec)
26.87 Computational imbalance (sec)

0.00            94.78 (9.89%)            958.56
```

```
0.00 MPI_Send
16.02 amps_Recv
16.02 MPI_Probe
0.00 MPI_Recv
0.00 new_HBT_element
99.92 Solve
0.00 FreeCommPkg
99.92 SolverRichards
94.20 SetupRichards
0.00 MetadataAddParflowDomainInfo
0.00 MPI_Reduce
41.85 SetProblemData
33.36 Geometries
74.56 GrGeomSolidFromGeom
74.56 GrGeomNewSolid
74.56 ComputeBoxes
2.01 ComputeInteriorBoxes
0.00 InitVectorUpdate
0.00 InitCommunication
0.00 amps_IExchangePackage
0.00 MPI_Recv_init
0.00 MPI_Ssend_init
0.00 MPI_Startall
83.98 FinalizeVectorUpdate
83.98 FinalizeCommunication
83.98 amps_Wait
83.98 amps_wait_exchange
83.98 MPI_Waitall
0.00 FreeVector

0.00            100.00            100.00
0.00        94.78 (100.00%)        94.78
```

Figure 24: *Cube Visualization of Wait States in ParFlow with Probe Handling.*

The Scalasca analysis with probe handling reveals a greater amount of *Late Sender* waiting time than was evident in the preceding results. Figure 24 depicts the analysis results obtained by the prototype implementation, visualized in Cube. Here, 94.78 seconds of *Late Sender* waiting time was detected, thus 13.3 seconds additional waiting time. These 13.3 seconds can be expected to correspond to waiting time spent in probe calls, given that both analysis runs analyzed the same trace data and the only difference is that the prototype implementation handles the probes. The

flat tree values indicate that 16.02% of the *Late Sender* waiting time was spent in MPI_Probe calls and 83.98% in MPI_Waitall calls.
The detected waiting time in the probes can have an effect on the delay costs and the critical path. Since almost no *Late Receiver* waiting times were identified in the application, it can be assumed that the critical path and the delay analysis conducted with the probe handling Scalasca version are correct. Figure 25 visualizes the critical path imbalance obtained with the Scalasca master branch and Figure 26 that one obtained with the prototype implementation. Figure 26 also illustrate that the additional waiting time is assigned as *Late Sender* delay costs. The critical path imbalance shows the positive difference between the time in a call path on the critical path and the average time spent in this call path. In other words, it highlights the call paths where the execution time on the critical path exceeded the average execution time. The total amount of imbalance only differentiates slightly, but its distribution to the different ranks shows differences. On rank seven 3.30 seconds of imbalance were detected in the original analysis whereas the probe handling analysis detected 4.58 seconds. In order to determine where the imbalance differences occur, the call tree is taken into account. Here, it becomes evident that the routine *WritePFBinary* actually has a greater impact on the critical path than assumed by the analysis that did not take probes into account.
The analysis of the ParFlow application demonstrates that the methodologies presented in this work for handling probes offer enhanced value for the Scalasca analysis. Although no fundamentally new ideas for improving the performance were identified in this application—since the performance bottleneck lies in the computation time of numerical methods for solving the Richardson equation—the extended analysis successfully identified an additional 16.02% of waiting time. Consequently, the extended prototype implementation detected waiting times that were previously missed by the original analysis, and the probe handling prevented that the previously uncovered waiting times falsify the critical path. This shows that the extended analysis provides valuable improvements.

5 Conclusion and Future Work

This thesis presented an in-depth exploration of methodologies for handling probe calls in the performance analysis tool Scalasca. The existing OTF2 event model was extended with two additional events: a probe and a match event. The probe event is recorded in the MPI_Probe, MPI_Mprobe and successful MPI_Iprobe, and MPI_Improbe calls, while the match event is written in MPI_Mrecv and MPI_Imrecv regions prior to the receive event. Based on these events, methodologies to extend the timestamp correction, wait state detection, delay analysis and critical path analysis were discussed and prototypically implemented in the Scalasca analyzer.

First, the preprocessing step was extended to perform a matching between the probe events and the receive events that retrieve the probed messages. This matching allows to distinguish whether a receive was probed. Using this information, the timestamp correction was enhanced to fix clock condition violations in send-probe pairs. Subsequently, the events were handled in the wait state analysis, enabling the detection of *Late Sender* wait states in probe calls. Here, a distinction was made between pure *Late Sender* wait states, wrong order situations, and simultaneous *Late Sender* and *Late Receiver* situations. The latter introduced the problem that two synchronization points are present in the send event, which could not be handled by the delay analysis infrastructure. Since *Late Sender* wait states are more common than *Late Receivers*, the delay costs for the *Late Senders* were calculated in this corner case. Furthermore, the critical path analysis was extended to handle the probe events. To demonstrate the additional value offered by the probe handling and the functionality of the implemented prototype, various test cases and the real-world application ParFlow were evaluated. These use cases showed that the extended prototype implementation detected waiting times that were previously missed by the original analysis, and that the probe handling prevented the previously uncovered waiting times to falsify the critical path, thereby demonstrating the enhancement of the extended analysis.

Future work could address improvements to the implemented prototype analysis that could not be implemented in this thesis, with the aim of integrating probe handling into a release version of Scalasca. Currently, the preprocessing matching between probe and receive events requires the internal representation of the receive event to include an additional parameter for holding the event ID of the related probe, or a special value indicating that the receive was not probed. However, many applications do not use message probing, and in these cases, the additional storage is needed but not utilized. To address this issue, it may be beneficial to introduce an internal event representation for probed receive events within Scalasca. When the trace data is read into the main memory of each process, Scalasca creates objects representing each event. During this process, at the time a receive event is encountered, it could be determined whether the receive matches a prior probe event. In the case it does, Scalasca would not store an instance of the regular receive event representation, but an instance of an internal probed receive event representation that offers an additional parameter for the probe event ID. Additionally, the infrastructure of the delay analysis should be adjusted to handle two synchronization points for a single event. Initially, this would involve changing the data structure of the *synchpoint* and *synchrank* maps. As the event where the synchronization point occurs is the key of these maps, the data structure for the value would need to be a list, making it possible to store synchronization information for multiple points. These should be ordered by the timestamps of the events on the peer processes. Considerations will need to be made on how to manage this in the analysis. Another area for future work involves extending the implementation to handle multi-threaded applications, where MPI communication can occur on each thread. This may result in the probe and receive events occurring at different locations, requiring a revision of the preprocessing matching. In the case of regular probes, the matching can still be erroneous. However, for matching probes, the related receive can be identified through the message handle ID. This would require the variable storing this ID to be shared across the threads, along with a corresponding locking mechanism to prevent simultaneous writes by different threads. A further challenge arises because even when the event ID from the corresponding receive is stored with the probe event, this receive event cannot be accessed if it occurs on another location, as event IDs are only accessible within the same location.

References

[1] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*, November 2023. URL https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[2] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 303–312, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

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

[4] Michael Gerndt. *Performance Analysis Tools*, pages 1515–1522. Springer US, Boston, MA, 2011.

[5] Score-P. Website, accessed 2024-07-18. URL https://www.score-p.org/.

[6] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer 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 Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[7] Adrian Munera, Sara Royuela, Germán Llort, Estanislao Mercadal, Franck Wartel, and Eduardo Quiñones. Experiences on the characterization of parallel applications in embedded systems with extrae/paraver. pages 1–11, 2020.

[8] Laksono Ad hianto, S Banerjee, M. Fagan, M Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22, 2009.

[9] Departament Computadors, Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. *WoTUG-18*, 44, 1995.

[10] Wolfgang Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12, 1996.

[11] Scalasca. Website, accessed 2024-07-18. URL https://www.scalasca.org/.

[12] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[13] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. Periscope: An online-based distributed performance analysis tool. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 1–16, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[14] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang Nagel, and Felix Wolf. Open trace format 2: The next generation of scalable trace formats and support libraries. volume 22, pages 481 – 490, 2012.

[15] Robert Bell, Allen D. 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*, pages 17–26, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[16] K.A. Huck and A.D. Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 41–41, 2005.

[17] Pavel Saviankou, Michael Knobloch, Anke Visser, and Bernd Mohr. Cube v4: From performance report explorer to performance analysis tool. *Procedia Computer Science*, 51:1343–1352, 2015.

[18] F. Wolf and B. Mohr. EPILOG Binary Trace-Data Format (Version 1.1). Technical Report ZAM-IB-2004-06, Jülich, 2004.

[19] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the open trace format (otf). In Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, pages 526–533, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[20] Cube. Website, accessed 2024-07-18. URL https://scalasca.org/software/cube-4.x/.

[21] Daniel Becker, Rolf Rabenseifner, and Felix Wolf. Timestamp synchronization for event traces of large-scale message-passing applications. In *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'07, page 315–325, Berlin, Heidelberg, 2007. Springer-Verlag.

[22] Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John C. Linford. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Comput.*, 35(12):595–607, 2009.

[23] Rolf Rabenseifner. The controlled logical clock-a global time for trace based software monitoring of parallel applications in workstation clusters. In *PDP*, pages 477–484. Citeseer, 1997.

[24] David Böhme, Markus Geimer, Felix Wolf, and Lukas Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *2010 39th International Conference on Parallel Processing*, pages 90–100, 2010.

[25] David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. Identifying the root causes of wait states in large-scale parallel applications. 3(2), 2016.

[26] David Böhme, Felix Wolf, Bronis R. de Supinski, Martin Schulz, and Markus Geimer. Scalable critical-path based performance analysis. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1330–1340, 2012.

[27] JURECA administrators. Jureca user documentation, accessed 2024-08-06. URL https://apps.fz-juelich.de/jsc/hps/jureca/.

[28] ParFlow. Software. URL https://doi.org/10.5281/zenodo.10989198.

List of Figures

Test Codes

```c
#include <mpi.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        int data1 = 42;
        MPI_Send(&data1, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);

        int data2 = 24;
        MPI_Send(&data2, 1, MPI_INT, 1, 2, MPI_COMM_WORLD);

        MPI_Probe(1, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Probe(1, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        int recv_data;
        MPI_Recv(&recv_data, 1, MPI_INT, 1, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    } else if (rank == 1) {
        MPI_Message message;

        MPI_Mprobe(0, 1, MPI_COMM_WORLD, &message, MPI_STATUS_IGNORE);
        MPI_Probe(0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        int recv_data1;
        int recv_data2;

        MPI_Recv(&recv_data1, 1, MPI_INT, 0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Mrecv(&recv_data2, 1, MPI_INT, &message, MPI_STATUS_IGNORE);

        int send_data = recv_data1 + recv_data2;
        MPI_Send(&send_data, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

Listing 1: *Test Case: Preprocessing Matching*

```

#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

namespace custom {
    void sleep(int seconds) {
        ::sleep(seconds);
    }
};

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    if (world_rank == 0) {
        int number = 42;
        custom::sleep(2);
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (world_rank == 1) {
        int recv_number;
        MPI_Status status;
        MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(&recv_number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
         ↪  MPI_STATUS_IGNORE);
        custom::sleep(1);
        MPI_Send(&recv_number, 1, MPI_INT, 2, 1, MPI_COMM_WORLD);
    } else if (world_rank == 2) {
        int recv_number;
        MPI_Status status;
        MPI_Probe(1, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&recv_number, 1, MPI_INT, 1, 1, MPI_COMM_WORLD,
         ↪  MPI_STATUS_IGNORE);
    }

    MPI_Finalize();
    return 0;
}

```

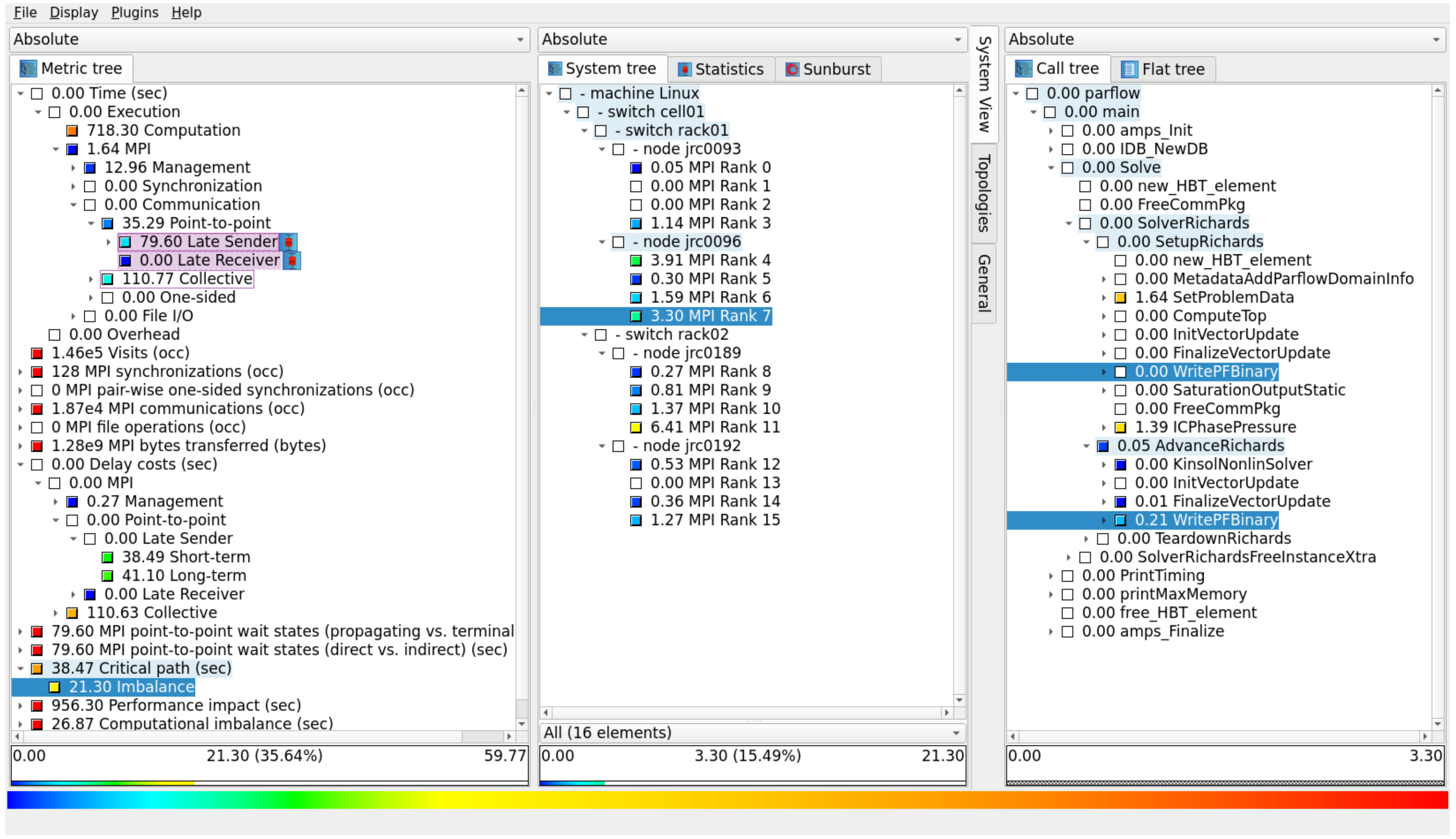
Listing 2: *Test Case: Late Sender*

```

#include <iostream>
#include <vector>
#include <mpi.h>
#include <unistd.h>

#define MSG_SIZE 1000

namespace custom {
    void sleep(int seconds) {
        ::sleep(seconds);
    }
};

std::vector<int> fillArray(int size) {
    std::vector<int> result(size);
    for (int i = 0; i < size; ++i) {
        result[i] = i + 1;
    }
    return result;
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_rank != 0) {
        std::vector<int> send_data = fillArray(MSG_SIZE);
        int dest_rank = 0;
          custom::sleep(world_rank);
        MPI_Send(&send_data[0], MSG_SIZE, MPI_INT, dest_rank,
         ↪  world_rank, MPI_COMM_WORLD);
    } else {
          for (int i = 2; i < world_size; ++i) {
            MPI_Status status;
            MPI_Probe(i, i, MPI_COMM_WORLD, &status);

            std::vector<int> recv_data(MSG_SIZE);
            MPI_Recv(&recv_data[0], MSG_SIZE, MPI_INT, i, i,
             ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
          }
        MPI_Status status;
        MPI_Probe(1, 1, MPI_COMM_WORLD, &status);

        std::vector<int> recv_data(MSG_SIZE);
        MPI_Recv(&recv_data[0], MSG_SIZE, MPI_INT, 1, 1,
         ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    MPI_Finalize();
    return 0;
}

```

Listing 3: *Test Case: Late Sender Wrong Order*

```

#include <iostream>
#include <vector>
#include <mpi.h>
#include <unistd.h>

#define MSG_SIZE 10000000

void sleep1(int seconds) {
    sleep(seconds);
}

void sleep2(int seconds) {
    sleep(seconds);
}

std::vector<int> fillArray(int size) {
    std::vector<int> result(size);
    for (int i = 0; i < size; ++i) {
        result[i] = i + 1;
    }
    return result;
}


int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    if (world_rank % 2 == 0) {
        std::vector<int> send_data = fillArray(MSG_SIZE);
        int dest_rank = (world_rank + 1) % world_size;
        sleep1(1);
        MPI_Send(&send_data[0], MSG_SIZE, MPI_INT, dest_rank, 0,
         ↪  MPI_COMM_WORLD);
    } else {
        MPI_Status status;
        MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
        sleep2(2);
        std::vector<int> recv_data(MSG_SIZE);
        MPI_Recv(&recv_data[0], MSG_SIZE, MPI_INT, MPI_ANY_SOURCE, 0,
         ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    MPI_Finalize();
    return 0;
}

```

Listing 4: *Test Case: Late Sender Late Receiver*



File Display Plugins Help

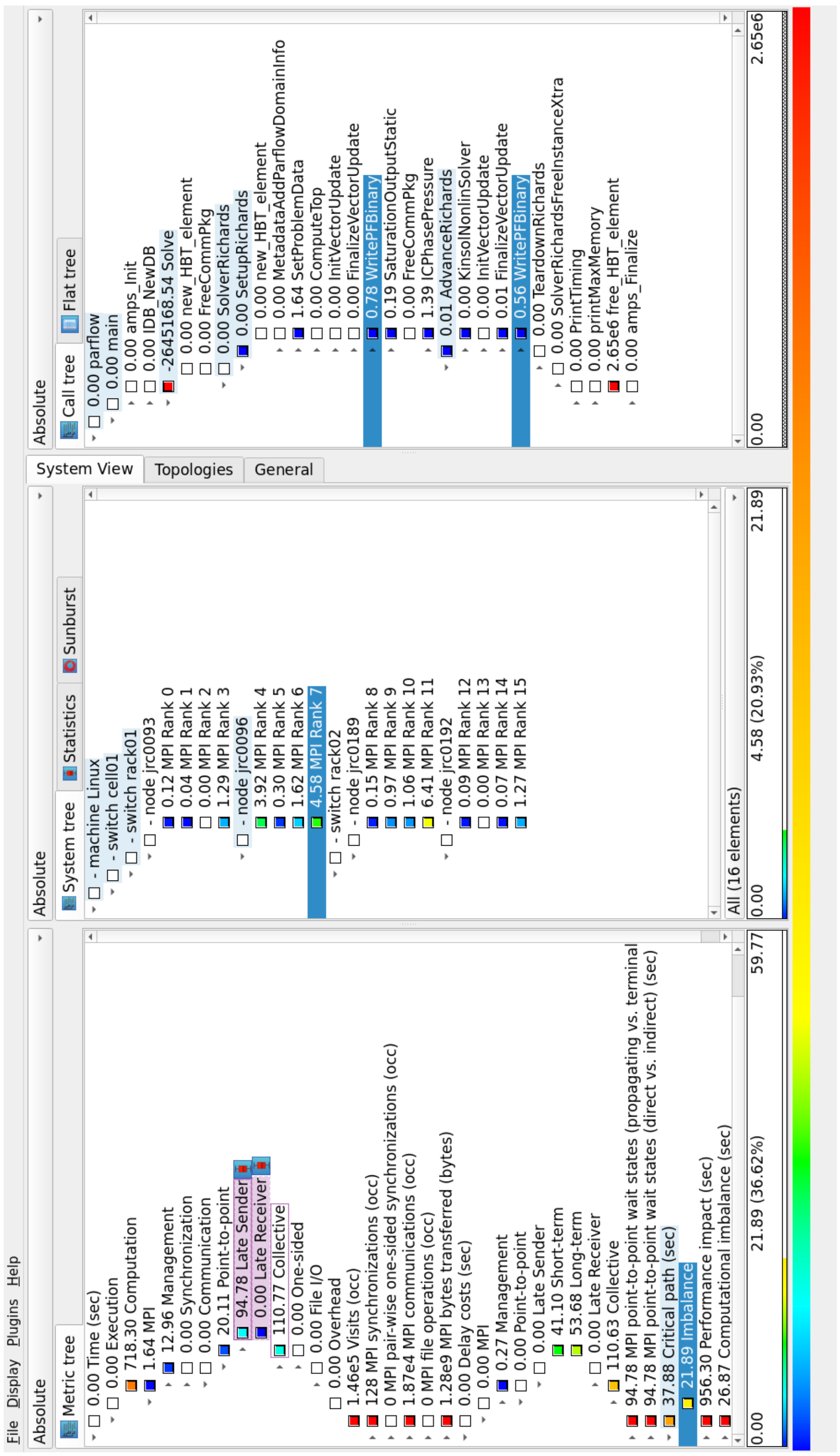
```
Absolute
Metric tree
0.00 Time (sec)
  0.00 Execution
    718.30 Computation
    1.64 MPI
      12.96 Management
      0.00 Synchronization
      0.00 Communication
        35.29 Point-to-point
          79.60 Late Sender
          0.00 Late Receiver
        110.77 Collective
        0.00 One-sided
      0.00 File I/O
  0.00 Overhead
1.46e5 Visits (occ)
128 MPI synchronizations (occ)
0 MPI pair-wise one-sided synchronizations (occ)
1.87e4 MPI communications (occ)
0 MPI file operations (occ)
1.28e9 MPI bytes transferred (bytes)
0.00 Delay costs (sec)
  0.00 MPI
    0.27 Management
    0.00 Point-to-point
      0.00 Late Sender
        38.49 Short-term
        41.10 Long-term
      0.00 Late Receiver
    110.63 Collective
79.60 MPI point-to-point wait states (propagating vs. terminal
79.60 MPI point-to-point wait states (direct vs. indirect) (sec)
38.47 Critical path (sec)
  21.30 Imbalance
956.30 Performance impact (sec)
26.87 Computational imbalance (sec)
0.00          21.30 (35.64%)          59.77
```

```
Absolute
System tree   Statistics   Sunburst
- machine Linux
  - switch cell01
    - switch rack01
      - node jrc0093
          0.05 MPI Rank 0
          0.00 MPI Rank 1
          0.00 MPI Rank 2
          1.14 MPI Rank 3
      - node jrc0096
          3.91 MPI Rank 4
          0.30 MPI Rank 5
          1.59 MPI Rank 6
          3.30 MPI Rank 7
    - switch rack02
      - node jrc0189
          0.27 MPI Rank 8
          0.81 MPI Rank 9
          1.37 MPI Rank 10
          6.41 MPI Rank 11
      - node jrc0192
          0.53 MPI Rank 12
          0.00 MPI Rank 13
          0.36 MPI Rank 14
          1.27 MPI Rank 15
All (16 elements)
0.00          3.30 (15.49%)          21.30
```

System View   Topologies   General

```
Absolute
Call tree   Flat tree
0.00 parflow
  0.00 main
    0.00 amps_Init
    0.00 IDB_NewDB
    0.00 Solve
      0.00 new_HBT_element
      0.00 FreeCommPkg
      0.00 SolverRichards
        0.00 SetupRichards
          0.00 new_HBT_element
          0.00 MetadataAddParflowDomainInfo
          1.64 SetProblemData
          0.00 ComputeTop
          0.00 InitVectorUpdate
          0.00 FinalizeVectorUpdate
          0.00 WritePFBinary
          0.00 SaturationOutputStatic
          0.00 FreeCommPkg
          1.39 ICPhasePressure
        0.05 AdvanceRichards
          0.00 KinsolNonlinSolver
          0.00 InitVectorUpdate
          0.01 FinalizeVectorUpdate
          0.21 WritePFBinary
        0.00 TeardownRichards
      0.00 SolverRichardsFreeInstanceXtra
    0.00 PrintTiming
    0.00 printMaxMemory
    0.00 free_HBT_element
    0.00 amps_Finalize
0.00                                  3.30
```

Figure 25: *Cube Visualization of the Critical Path in ParFlow with Scalasca Master.*



```
File  Display  Plugins  Help

Absolute
Metric tree
  0.00 Time (sec)
    0.00 Execution
      718.30 Computation
      1.64 MPI
        12.96 Management
        0.00 Synchronization
        0.00 Communication
          20.11 Point-to-point
            94.78 Late Sender
            0.00 Late Receiver
          110.77 Collective
          0.00 One-sided
        0.00 File I/O
    0.00 Overhead
  1.46e5 Visits (occ)
  128 MPI synchronizations (occ)
  0 MPI pair-wise one-sided synchronizations (occ)
  1.87e4 MPI communications (occ)
  0 MPI file operations (occ)
  1.28e9 MPI bytes transferred (bytes)
  0.00 Delay costs (sec)
    0.00 MPI
      0.27 Management
      0.00 Point-to-point
        0.00 Late Sender
          41.10 Short-term
          53.68 Long-term
        0.00 Late Receiver
      110.63 Collective
  94.78 MPI point-to-point wait states (propagating vs. terminal
  94.78 MPI point-to-point wait states (direct vs. indirect) (sec)
  37.88 Critical path (sec)
    21.89 Imbalance
  956.30 Performance impact (sec)
  26.87 Computational imbalance (sec)
0.00        21.89 (36.62%)        59.77

Absolute
System tree | Statistics | Sunburst
  - machine Linux
    - switch cell01
      - switch rack01
        - node jrc0093
          0.12 MPI Rank 0
          0.04 MPI Rank 1
          0.00 MPI Rank 2
          1.29 MPI Rank 3
        - node jrc0096
          3.92 MPI Rank 4
          0.30 MPI Rank 5
          1.62 MPI Rank 6
          4.58 MPI Rank 7
      - switch rack02
        - node jrc0189
          0.15 MPI Rank 8
          0.97 MPI Rank 9
          1.06 MPI Rank 10
          6.41 MPI Rank 11
        - node jrc0192
          0.09 MPI Rank 12
          0.00 MPI Rank 13
          0.07 MPI Rank 14
          1.27 MPI Rank 15
All (16 elements)
0.00        4.58 (20.93%)        21.89

System View | Topologies | General

Absolute
Call tree | Flat tree
  0.00 parflow
    0.00 main
      0.00 amps_Init
      0.00 IDB_NewDB
      -2645168.54 Solve
        0.00 new_HBT_element
        0.00 FreeCommPkg
        0.00 SolverRichards
          0.00 SetupRichards
            0.00 new_HBT_element
            0.00 MetadataAddParflowDomainInfo
            1.64 SetProblemData
            0.00 ComputeTop
            0.00 InitVectorUpdate
            0.00 FinalizeVectorUpdate
            0.78 WritePFBinary
            0.19 SaturationOutputStatic
            0.00 FreeCommPkg
            1.39 ICPhasePressure
          0.01 AdvanceRichards
            0.00 KinsolNonlinSolver
            0.00 InitVectorUpdate
            0.01 FinalizeVectorUpdate
            0.56 WritePFBinary
          0.00 TeardownRichards
        0.00 SolverRichardsFreeInstanceExtra
      0.00 PrintTiming
      0.00 printMaxMemory
      2.65e6 free_HBT_element
      0.00 amps_Finalize
0.00        2.65e6
```

Figure 26: *Cube Visualization of the Critical Path in ParFlow with Probe Handling.*