

Fachhochschule Aachen
Campus Jülich

**Replay-based Performance Analysis of
Nonblocking Collective MPI Operations**

Masterarbeit
Katharina Haus

Fachbereich 9
Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik M.Sc.



Jülich, Juli 2026

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 06.07.2026

Unterschrift des Studierenden

Diese Arbeit wurde betreut von:

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

The efficient use of large-scale high-performance computing systems requires identifying and eliminating communication bottlenecks, which often manifest as wait states. The Scalasca trace tools automatically detect and quantify such wait states through a replay-based analysis of event traces recorded by Score-P, tracing them back to their root causes and computing the critical path of the execution. While this analysis is well established for blocking collective MPI operations, it does not cover their nonblocking counterparts introduced with MPI-3. This thesis extends Scalasca's replay-based analysis to nonblocking collective MPI operations. The wait-state detection is implemented in Scalasca, and a theoretical concept for synchronization point tracking and the delay and critical-path analyses is developed. The extended tool is evaluated across a range of test cases, demonstrating its capability to identify wait states in nonblocking collective operations and showcasing its practical benefit.

Contents

1	Introduction	1
2	Background and Context	3
2.1	Message Passing Interface (MPI)	3
2.1.1	Communication Operations	3
2.1.2	Point-to-Point Communication.....	5
2.1.3	Collective Communication	7
2.2	Performance Analysis	9
2.2.1	Performance Metrics	9
2.2.2	Performance Analysis Tools	17
3	Replay-based Performance Analysis	25
3.1	Scalasca Infrastructure Overview	25
3.2	Wait-state Analysis	29
3.2.1	Backward Wait-state Detection	30
3.2.2	Forward Wait-state Detection	31
3.2.3	Handling of Multi-completions	37
3.3	Critical Path and Cost Accounting	39
3.3.1	Delay Analysis	39
3.3.2	Critical-path Analysis	42
4	Analysis of Nonblocking Collective MPI Operations	45
4.1	Wait-state Analysis	45
4.1.1	Forward Replay Pass Implementation	49
4.2	Synchronization Points	56
4.3	Delay and Critical-path Analyses	60

5	Experimental Evaluation	65
5.1	Functional Correctness Verification	65
5.1.1	Wait at Barrier	66
5.1.2	Late Broadcast	67
5.1.3	Early Reduce	69
5.1.4	Early Scan	70
5.1.5	Wait at $N \times N$	70
5.1.6	Most-severe Instance Tracking in Multi-completion	71
5.2	Case Study	72
5.2.1	Experimental Setup	73
5.2.2	Reduce Variant	74
5.2.3	Allreduce Variant: Runtime	75
5.2.4	Allreduce Variant: Trace Analysis	75
6	Conclusion and Future Work	81
	References	83
	List of Figures	88
	List of Tables	89
	Appendix	90

1 Introduction

Modern high-performance computing (HPC) systems deliver enormous computing power. However, ever larger and more powerful systems are of little use if applications do not make efficient use of the available resources. Performance analysis is therefore a prerequisite for the effective use of large-scale parallel systems. On distributed-memory systems, processes communicate by exchanging messages, and the Message Passing Interface (MPI) [1] has become the de facto standard for this communication. As applications scale to thousands of processes, communication overhead often becomes an increasingly dominant factor in overall runtime, yet without dedicated tools it is difficult to detect and localize. Inefficiencies such as computational imbalance often surface as wait states at the next synchronization point between processes. A wait state is a period during which a process is blocked at a communication call rather than performing useful computation. Identifying and quantifying such wait states is a necessary step toward tracking down their root causes, which are the points where optimization has the potential to reduce runtime. The Scalasca trace tools [2, 3] provide automated wait-state detection and root-cause identification following a replay-based approach. Its analyzer operates on event traces recorded by Score-P [4] in the OTF2 [5] format and identifies wait states by replaying the recorded communication events during a parallel execution. For blocking collective operations, this analysis is well established. MPI-3, however, introduced nonblocking collective operations, which decouple the initiation of a collective from its completion. The primary motivation is to allow applications to overlap computation with communication to hide latency, so that by the time the completion call is reached, the operation has already progressed or finished in the background. However, this potential is only realized if the processes involved are sufficiently balanced in their progress through the program. If imbalances between participating processes are large enough that some have not yet initiated the operation while others are already attempting to complete it, wait states still occur. These are structurally analogous to their blocking counterparts but deferred from the operation itself to the separate completion call. Unfortunately, the identification of wait states in nonblocking collective operations and their root causes is not yet covered by existing performance analysis tools, including Scalasca. Thus, application developers using such tools may only get an incomplete picture of the performance of their codes. This thesis addresses this gap by extending Scalasca’s replay-based wait-state analysis to nonblocking collective MPI operations. The core challenge lies in exchanging the timestamps required for wait-state computation. The ordering constraints imposed by the MPI standard on collective operations, together with the possibility of completions occurring inside test calls, impose restrictions on where and how information can be exchanged during the analysis replay.

The specific contributions of this thesis are:

- **Extended wait-state definitions.** The wait-state classes established for blocking collective operations are extended to their nonblocking counterparts. Additionally, wait-state classes caused by the rendezvous protocol are introduced for both blocking and nonblocking collective operations.
- **Implementation of wait-state analysis for nonblocking collectives.** A two-pass approach is designed and implemented to detect and quantify *Wait at Barrier*, *Late Broadcast*, *Early Reduce*, *Early Scan*, *Wait at $N \times N$* and *Wait at Communicator Create* wait states in completion calls. Most-severe instance tracking in multi-completion calls is extended to incorporate nonblocking collective wait-state candidates alongside point-to-point candidates.
- **Theoretical concept for delay and critical-path analysis.** Approaches for tracking synchronization points for nonblocking collectives and the delay and critical-path analyses that build on them are discussed in detail. The key challenge of exchanging data during a backward replay without deadlocking with blocking MPI calls already present in the analysis is identified, and a point-to-point-based approach that resolves this is outlined. Implementation of these analyses is left as future work.
- **Experimental evaluation of wait-state analysis.** Functional correctness is verified against a dedicated test suite covering all wait-state classes. The practical benefit of the extended analysis is demonstrated using a parallel sandpile benchmark on two HPC systems across multiple MPI implementations.

The remainder of this thesis is structured as follows. Chapter 2 introduces the relevant MPI communication concepts, the performance metrics used throughout this work, and the performance analysis infrastructure comprising Score-P and the OTF2 event model. It also provides an overview of related work. Chapter 3 describes the replay-based concept of Scalasca, using blocking collective operations as a concrete example to illustrate how wait states are detected, root causes are identified, and the critical path is computed. Chapter 4 presents the implementation of the extended wait-state analysis for nonblocking collective operations and outlines the theoretical concept for synchronization point tracking and delay and critical-path analyses. Chapter 5 evaluates the wait-state analysis implementation in terms of functional correctness and practical benefit. Finally, Chapter 6 concludes the thesis and outlines directions for future work.

2 Background and Context

This chapter introduces the concepts and tools that form the foundation of this thesis. Section 2.1 provides an overview of the Message Passing Interface, covering the point-to-point and collective communication paradigms relevant to this work. Section 2.2 introduces the performance analysis methodology, including the metrics used to characterize communication inefficiencies, and presents the instrumentation and measurement infrastructure Score-P, together with the OTF2 event model.

2.1 Message Passing Interface (MPI)

The Message Passing Interface (MPI) [1] is an application programming interface for a message-passing library, standardized by the MPI Forum. MPI itself is a specification that defines library routines for distributed memory systems, with several implementations available. Processes in MPI are organized into logical groups. Within each group, every process is assigned a unique integer identifier called a *rank*, ranging from 0 to $N - 1$, where N is the size of the group. A rank is always relative to the group it belongs to. The same physical process may hold different ranks in different groups. A communicator defines a communication context. An *intra-communicator* handles communication among processes within a single group, while an *inter-communicator* handles communication between two distinct groups. By default, MPI provides a single group containing all processes in a given execution and a corresponding intra-communicator `MPI_COMM_WORLD`. MPI allows the definition of user-defined communicators containing a subset of processes. This thesis considers only intra-communicators, which will henceforth be referred to as communicators.

An MPI program begins by calling `MPI_Init` or `MPI_Init_thread`, which initializes the MPI execution environment. It ends with `MPI_Finalize`, which releases all resources held by the MPI library. All MPI communication calls must occur between these two boundaries. MPI provides four main communication classes: point-to-point (P2P), collective, one-sided, and I/O. This thesis focuses on point-to-point and collective operations, both of which are available in blocking, nonblocking, and persistent variants.

2.1.1 Communication Operations

Operation Stages

An MPI operation progresses through four stages: initialization, starting, completion, and freeing. In the first stage, the *initialization* stage, the argument list is passed to the operation. The user retains control of the data buffers and is still allowed to modify them. In the *starting* phase, the control of the data buffers is handed over to the MPI library. The user must not modify input buffers or read output buffers until control is returned. The

completion stage returns control of the data buffers and guarantees that output buffers and arguments have been updated. Finally, in the *freeing* stage, control of the remaining arguments, such as buffer addresses and array arguments, is returned to the user, who may then safely reuse or deallocate them.

The combination of the initialization and starting stages is called *initiation*. An operation that is initiated but not yet completed is called *pending*. Figure 1 illustrates how the stages are grouped into procedure calls depending on whether the operation is blocking, nonblocking, or persistent.

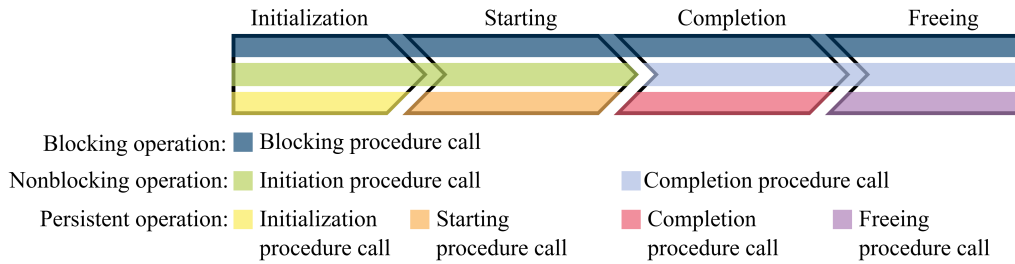


Figure 1: *Mapping of MPI operation stages to procedure calls for blocking, nonblocking, and persistent operations.*

A persistent operation exposes each stage as an individual procedure call, allowing the initialization and freeing stages to be performed once, while the starting and completion stages may be repeated for multiple invocations of the same operation.

Blocking Communication

A blocking procedure call traverses all four stages before returning to the caller. The calling process is blocked for the entire duration of the operation and cannot perform any other work until the call returns. A blocking procedure may be *local* or *nonlocal*. A local procedure can complete independently of actions on other processes, whereas a nonlocal procedure can only complete once a corresponding operation on another process has been initiated. A blocking operation is complete after the call returns and all buffers can be reused. However, it prevents any overlap of computation and communication. If a nonlocal blocking operation cannot complete immediately, for example, because a matching operation on a remote process has not yet been initiated, the calling process stalls and waits.

Nonblocking Communication

A nonblocking operation splits the stages into two calls: an initiation procedure call covering the initialization and starting stages and a separate completion procedure call covering the completion and freeing stages. The primary motivation is to allow the application to overlap computation with communication. Rather than blocking until

the operation is complete, the process can perform useful work between initiation and completion, potentially hiding communication latency. The initiation call is local. It returns immediately after handing control of the data buffers to the MPI library, without requiring any action on remote processes. The calling process receives an opaque request, which uniquely identifies the pending operation. Opaque objects are managed by the MPI library and not directly accessible to the user. They can be accessed through handles. After initiation, the process is free to perform other computations while MPI is then responsible for progressing the operation in the background. A separate completion call must be issued with the request handle to complete the operation. MPI provides two categories of completion calls. `MPI_Wait` blocks until a single specified operation is complete, while `MPI_Test` returns immediately and indicates through a flag whether the operation has completed or not. For managing multiple pending operations simultaneously, MPI provides `MPI_Waitall`, which blocks until all operations in a given set are complete. `MPI_Waitany` blocks until at least one has completed and returns that one. `MPI_Waitsome` also blocks until at least one has completed but returns all that have finished rather than just one. Multi-completion calls may complete operations of different types, for example, point-to-point and collective. The corresponding variants `MPI_Testall`, `MPI_Testany`, and `MPI_Testsome` behave analogously but return immediately and provide one or more flags (depending on the respective call) indicating successful completion, similar to `MPI_Test`. All test calls are local. A wait call can be nonlocal, as it may require that a corresponding operation on another process has been initiated before the operation can finish.

2.1.2 Point-to-Point Communication

Point-to-point communication involves exactly two processes. One process sends a message that the other one receives. Point-to-point messages in MPI carry an *envelope* and a *payload*. The envelope consists of the source rank, destination rank, a user-defined integer tag, and the communicator. It is used to match a send operation to its corresponding receive. On the receiver side, the utilization of wildcard options for both the tag and the source argument allows receiving messages with an arbitrary tag and/or from any sender. The actual source rank and tag can be retrieved from the `MPI_Status` object returned by the receive call. The payload describes the data being transferred, specified as a triple of a memory buffer, an MPI datatype, and a count. MPI provides a rich set of built-in datatypes, as well as mechanisms for defining derived datatypes.

The completion of a point-to-point operation provides certain progress guarantees on the program execution. The completion of a receive, for example, guarantees that a matching send on the peer process has been initiated. When using synchronous sends, the completion of a send additionally guarantees that a matching receive on the peer process has been initiated. This progress guarantee is referred to as *synchronization*. MPI distinguishes

between four send modes: standard, synchronous, buffered, and ready. In standard mode, a send operation may or may not synchronize, depending on the message size relative to an implementation-defined buffer size threshold. If the message is small enough to fit in the internal buffer, MPI copies it locally, and the send completes as a local operation. This is known as the eager protocol. If the message exceeds the threshold, MPI performs a rendezvous protocol, and the send can only complete once a matching receive has been initiated, making it a synchronizing operation. In synchronous mode, the send always synchronizes regardless of message size. In buffered mode, MPI is required to buffer the entire message, and the send always completes as a local operation. In ready mode, the send may only be issued if a matching receive is already posted. For point-to-point communication, send and receive operations of different variants may be freely matched. For example, a nonblocking send may be paired with a blocking receive and vice versa. Each of the four send modes is available in blocking (e.g., `MPI_Send`), nonblocking (e.g., `MPI_Isend`), and persistent (e.g., `MPI_Send_init`) variants, and correspondingly for receives (e.g., `MPI_Recv`, `MPI_Irecv`, `MPI_Recv_init`). In the persistent case, the communication arguments are bound to a reusable request object once. The operation can then be started repeatedly using `MPI_Start` or `MPI_Startall` and completed using the standard test or wait calls, amortizing the initialization overhead over multiple invocations. Once the persistent operation is no longer needed, the request must be freed with `MPI_Request_free`. Persistent communication is especially beneficial in applications with static communication patterns where the same argument list is reused across iterations. Persistent send and receive operations are independent and do not need to be paired with each other.

Figure 2 shows timeline diagrams of blocking and nonblocking point-to-point operations. The left diagram shows blocking point-to-point communication. The middle diagram shows nonblocking communication on both the sender and receiver sides. The right diagram shows a nonblocking receive paired with a blocking send. Time is denoted on the x-axis and processes on the y-axis. Each process corresponds to a horizontal line in the diagram. Rectangles indicate the time a process spends in a specific MPI call. Dark blue rectangles represent blocking procedure calls, green rectangles represent initiation calls for nonblocking operations, and light blue rectangles represent the corresponding completion calls. Grey rectangles correspond to non-MPI calls. For illustration purposes, the duration of nonblocking initiation and test calls is exaggerated; in practice, these calls return immediately. Arrows represent message transfer, starting at the earliest point a message may be sent and reaching to the completion of the reception.

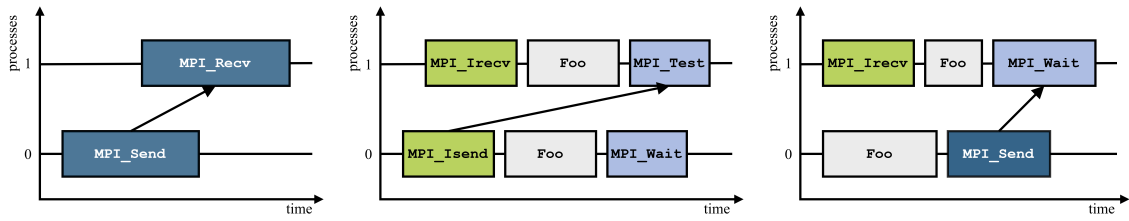


Figure 2: Timeline diagrams of point-to-point communication.

2.1.3 Collective Communication

Collective MPI operations involve a group of processes. Namely, all processes in a specified communicator participate in a collective operation. Most MPI collectives fall into three categories: *One-to-All*, *All-to-One*, *All-to-All* operations. Only a few specialized operations fall outside these three classes. Table 1 lists all collective operations along with their call parameters and a brief description.

	Operation	Parameters	Description	Options
One-to-All	Bcast	b, c, dt, r, comm	Root broadcasts buffer to all processes	
	Scatter	sb, sc, st, rb, rc, rt, r, comm	Root distributes distinct data to all processes	v
All-to-One	Gather	sb, sc, st, rb, rc, rt, r, comm	All processes send data to root	v
	Reduce	sb, rb, c, dt, op, r, comm	Reduction across all processes, result at root	
All-to-All	Alltoall	sb, sc, st, rb, rc, rt, comm	Each process sends distinct data to every process	v, w
	Allgather	sb, sc, st, rb, rc, rt, comm	Gather, result sent to all processes	v, w
	Allreduce	sb, rb, c, dt, op, comm	Reduce, result sent to all processes	
	Reduce_scatter	sb, rb, rcs, dt, op, comm	Reduce, result scattered across all processes	block
Other	Barrier	comm	Synchronizes all processes	
	Scan	sb, rb, c, dt, op, comm	Inclusive prefix reduction, result at rank i from ranks $0, \dots, i$	
	Exscan	sb, rb, c, dt, op, comm	Exclusive prefix reduction, result at rank i from ranks $0, \dots, i-1$	

Table 1: Overview of MPI collective operations, their parameters, and variants. Parameter abbreviations: b = buffer, sb = send buffer, rb = receive buffer, sc = send count, rc = receive count, rcs = receive counts (array), st = send type, rt = receive type, dt = datatype, c = count, op = operation, r = root, $comm$ = communicator.

In an *One-to-All* operation, a single root process communicates with all other processes in the communicator. A Broadcast distributes a buffer from root to all other ranks, while a Scatter distributes distinct portions of a buffer, sending a separate chunk to each rank. The inverse operations are classified as *All-to-One*. Gather collects data from all processes at the root, and Reduce applies a reduction operation such as sum, maximum, or user-defined functions across all processes and delivers the result to the root. *All-to-All* operations are characterized by every process both sending and receiving data. Allgather and Allreduce are variants of Gather and Reduce respectively, where the result is delivered to all processes rather than a single root. Alltoall generalizes Scatter and Gather simultaneously, where each process sends a distinct chunk of data to every other process and receives a distinct chunk from every other process. Reduce_scatter combines a reduction with a scatter of the result across all processes. The Reduce_scatter_block variant requires the result to be divided into equal-sized blocks before being scattered, such that each process receives the same number of elements, eliminating the need for a per-rank receive count array. Beyond these three classes, MPI provides a Barrier, which synchronizes all processes in the communicator without exchanging user-defined data, and two prefix reduction operations. Scan computes an inclusive prefix reduction, such that process i receives the reduction of values from all processes with rank $0, \dots, i$. Exscan computes the exclusive variant, where process i receives the reduction of values from ranks $0, \dots, i - 1$ only.

Blocking collective procedure calls are prefixed with MPI_, so the operations in Table 1 map to calls such as MPI_Bcast, MPI_Reduce, and so on. The corresponding nonblocking initiation calls are prefixed with MPI_I yielding, for example, MPI_Ibcast and MPI_Ireduce, and take an additional MPI_Request object as an argument to identify the operation at completion. Similar to the corresponding point-to-point operations, persistent variants are initialized with a call prefixed with MPI_Init_ and started with MPI_Start. The variants suffixed with v allow different data counts per rank, and those suffixed with w additionally allow different datatypes per rank. For operations that involve both send and receive buffers, MPI provides an MPI_IN_PLACE option, which allows the input buffer to serve as the output buffer, avoiding the need for a separate send buffer.

Beyond the operations listed in Table 1, MPI provides two further categories of collective operations. Communicator creation operations such as MPI_Comm_dup, MPI_Comm_split, and MPI_Comm_create are collective, as all processes in the parent communicator must participate. Of these, only communicator duplication provides a nonblocking variant, MPI_Comm_idup. Neighborhood collectives operate on a virtual process topology defined by a graph or Cartesian communicator, restricting communication to neighboring processes as defined by that topology. Available operations include MPI_Neighbor_alltoall and MPI_Neighbor_allgather, along with their v and w variants, all of which have nonblocking counterparts.

Unlike in point-to-point communication, it is not permitted to match different variants of operations. A blocking collective call on one process must be matched with a blocking collective call on all others, and the same applies to nonblocking and persistent variants. All ranks in a communicator must call the same collective operation, and collective operations on the same communicator must be initiated in the same order on all ranks. According to the MPI standard, “a correct, portable program must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not”¹. Furthermore, the relative ordering of point-to-point operations and collective operations must ensure freedom from deadlock even in cases where both types of operations are synchronizing. The order of completions may be arbitrary.

2.2 Performance Analysis

The communication operations described in the previous section incur costs beyond the transfer of data itself. Whenever processes synchronize or exchange messages, some may idle while waiting for others. Wall-clock time alone gives little insight into this behavior. It captures the total cost of a program, but not where time is lost or why. This section introduces the metrics and instrumentation infrastructure used throughout this thesis.

2.2.1 Performance Metrics

Wait State

A wait state occurs when a process arrives early at a synchronization point and is forced to idle until one or more peer processes reach that point. For blocking operations, wait states are implicit within the procedure call itself. For nonblocking operations, wait states are deferred to the completion call. When a nonblocking operation is completed via a successful test call, no wait state can occur, since test calls are local and do not block the calling process. When completed via a wait call, however, the calling process blocks until the operation completes, and therefore the same classes of wait states may occur as in the blocking analogues. The overhead associated with repeated test calls is not considered here.

Figure 3 illustrates wait states in blocking point-to-point communication. The timeline diagram in Figure 3a shows that process 1 posts a receive before process 0 has issued its send. Since no data can be received before it is sent, process 1 is forced to idle inside the receive call. The time process 1 spends waiting before process 0 enters the send is referred to as waiting time, and the corresponding wait state is called a *Late Sender* wait state. Figure 3b shows the symmetric case: a synchronous send is posted early. As a synchronous

¹Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 5.0*. June 2025, Chapter 6: Collective Communication, Section 6.14: Correctness, p. 294.

2 Background and Context

send may only complete once the receiver has initiated reception, the time spent in the send call before the peer process posts its receive constitutes waiting time. This wait state is called a *Late Receiver* wait state.

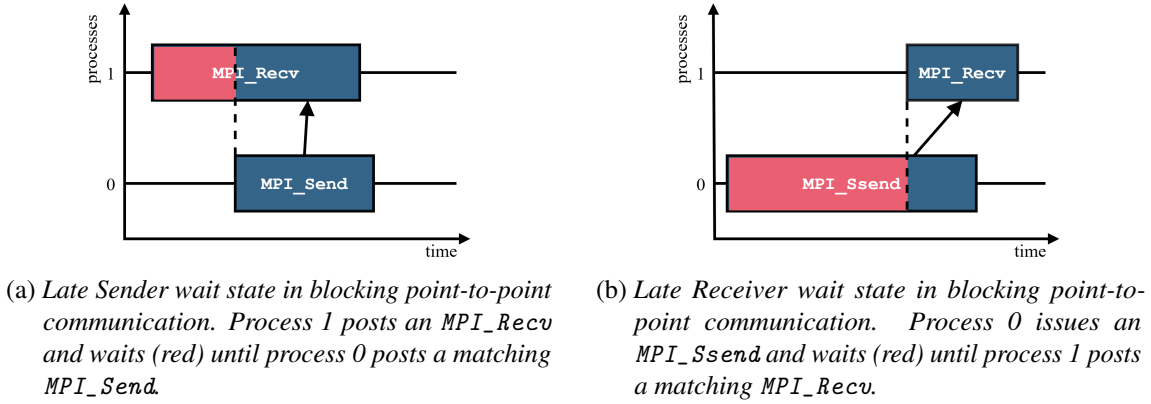


Figure 3: Wait states in blocking point-to-point communication.

The same wait states may also occur in the completion of a nonblocking operation. Figure 4 illustrates this for nonblocking point-to-point communication. Figure 4a shows a *Late Sender* wait state in the `MPI_Wait` call of a nonblocking receive. Process 1 initiates a receive and immediately calls `MPI_Wait`, but process 0 has not yet posted its send, forcing process 1 to idle inside the wait call. Figure 4b shows the corresponding *Late Receiver* wait state at the completion call of a nonblocking synchronous send. In both cases, the wait state is structurally identical to its blocking counterpart. Note that the timeline diagrams are intentionally constructed to illustrate wait states in isolation. In a real application, immediately following an initiation call with a completion call defeats the purpose of nonblocking communication, which is to overlap computation and communication. Nevertheless, such patterns do occur in practice.

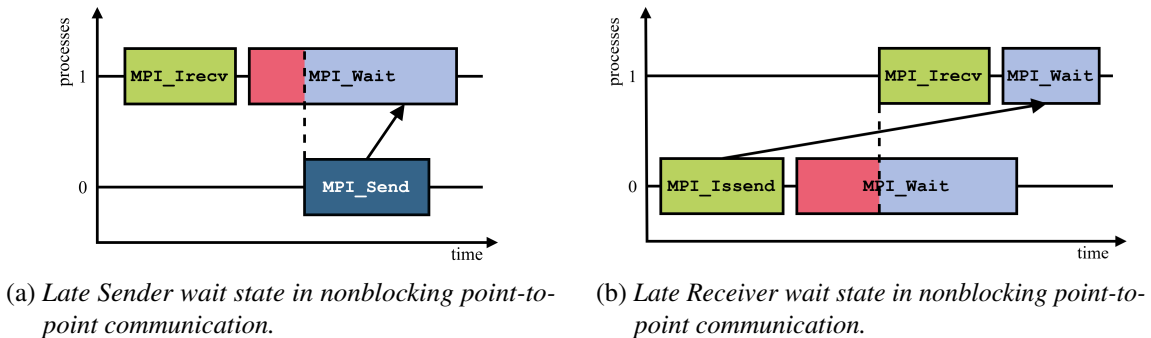


Figure 4: Wait states in nonblocking point-to-point communication.

For collective operations, the communication pattern determines which process can induce a wait state on others. The wait-state classes defined in this work extend the categories established in prior work on blocking collectives, adapting them to the nonblocking case

and introducing additional classes that arise from the rendezvous protocol. The categories of collective operations introduced earlier derive distinct classes of wait states.

Figure 5 illustrates *Late Broadcast* wait states in the completion of a nonblocking broadcast with process 0 serving as the root process. All non-root processes complete the operation in a wait call, while the root process completes it via a successful test call. The root process enters the initiation call later than the non-root processes. Since the root process is the source of the data, non-root processes cannot proceed until the root has initiated the broadcast. The time non-root processes spend waiting inside the completion call before the root enters the initiation therefore constitutes waiting time. This wait state is an example of a one-to-all wait state, which is the collective analog to the *Late Sender* wait state. In the blocking case, initiation and completion are merged into a single call, and the waiting time is therefore the time spent inside `MPI_Bcast` on non-root processes before the root enters the call.

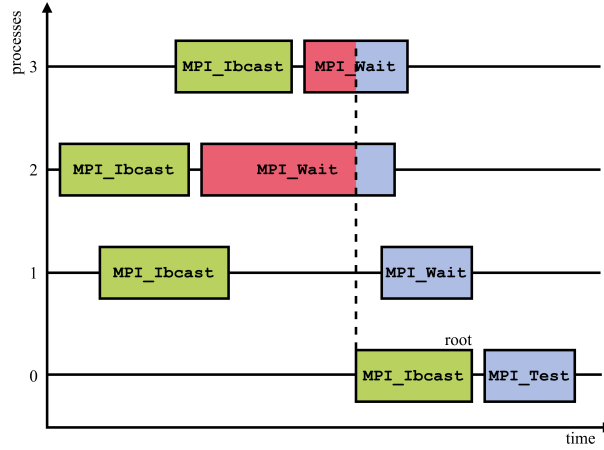


Figure 5: *Late Broadcast* wait states nonblocking collective communication with root on process 0.

It is important to note that the MPI standard only defines the semantics of collective operations but not their implementation. The wait state classification described above reflects the minimal synchronization guarantee imposed by the standard, that is, that a non-root process can only complete a broadcast after the root process has initiated it, establishing a synchronization point between the root and all non-root processes waiting for the root. In practice, however, the actual synchronization structure depends on the underlying collective communication algorithm. A tree-based broadcast implementation, for example, requires each process to wait for its parent before it can progress, meaning that non-root processes that have received the data may subsequently take on the role of sender for their child processes. Some intermediate processes may exhibit waiting time that can only be classified as wait states with knowledge of the implementation, but these are not observable from the MPI semantics alone.

Furthermore, if the implementation uses a rendezvous protocol to transfer the broadcast buffer, the sender (root in a flat algorithm, or each inner node in a tree-based algorithm)

must wait for its receivers to initiate the operation. This induces a wait state on the sender side that is analogous to the *Late Receiver* wait state in point-to-point communication, referred to here as an *Early Broadcast* wait state.

Figure 7 shows a timeline diagram of a nonblocking broadcast operation executed using the communication pattern illustrated in Figure 6 assuming a rendezvous protocol. Solid red areas represent *Late Broadcast* waiting time induced by a delayed root, dotted red areas represent *Late Broadcast* waiting time induced by inner nodes acting as senders, and dotted orange areas represent *Early Broadcast* wait-

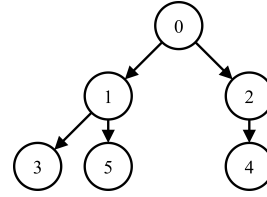


Figure 6: Example of a tree-based broadcast communication pattern.

ing time. Process 0 is the root and sends data to processes 1 and 2. Since process 0 completes the broadcast via a test call, no wait state occurs on the root. Process 1 enters its wait call before the root process has initiated the broadcast, and therefore exhibits a *Late Broadcast* wait state. The same applies to process 3. Although its wait call is entered after its parent process 1 has initiated the broadcast, it is entered before the root has done so. Since the parent cannot have received the data at that point, it cannot forward it further, making the initiation on the root the binding condition. Process 4 reveals a *Late Broadcast* wait state caused by a delayed broadcast initiation of its parent, process 2. Due to the rendezvous protocol, process 1 must additionally wait for process 5 to initiate the broadcast, causing the wait call on process 1 to block beyond the initiation on the root process and revealing an *Early Broadcast* wait state in addition to the *Late Broadcast* wait state. Given that the waiting time in a call is constrained by its own duration, the waiting time is determined by whichever wait state is more severe.

The *Late Broadcast* wait state can be understood as a generalization of the *Late Sender* wait state to the one-to-all communication pattern, while the *Early Broadcast* wait state generalizes the *Late Receiver* pattern. Both are not specific to broadcasts but apply to any one-to-all collective operation, including scatter. Considering scatter, not all processes necessarily use the same communication protocol. Depending on the algorithm employed, buffer sizes may vary between processes, such that transfers at a higher level may use a rendezvous protocol and processes closer to a leaf node use an eager protocol. This can occur even in a flat communication pattern when the `scatterv` option is used, which explicitly allows the buffer size to vary between processes.

As described above, waiting time arises at synchronization points where a process must idle until one or more peer processes arrive. In the case of a barrier, the MPI standard guarantees that no process can complete the operation before all processes in the communicator have initiated it. Every process therefore synchronizes with every other. The time a process spends inside a blocking `MPI_Barrier`, or inside the completion call of `MPI_Ibarrier`, before the last process has initiated the operation constitutes waiting time.

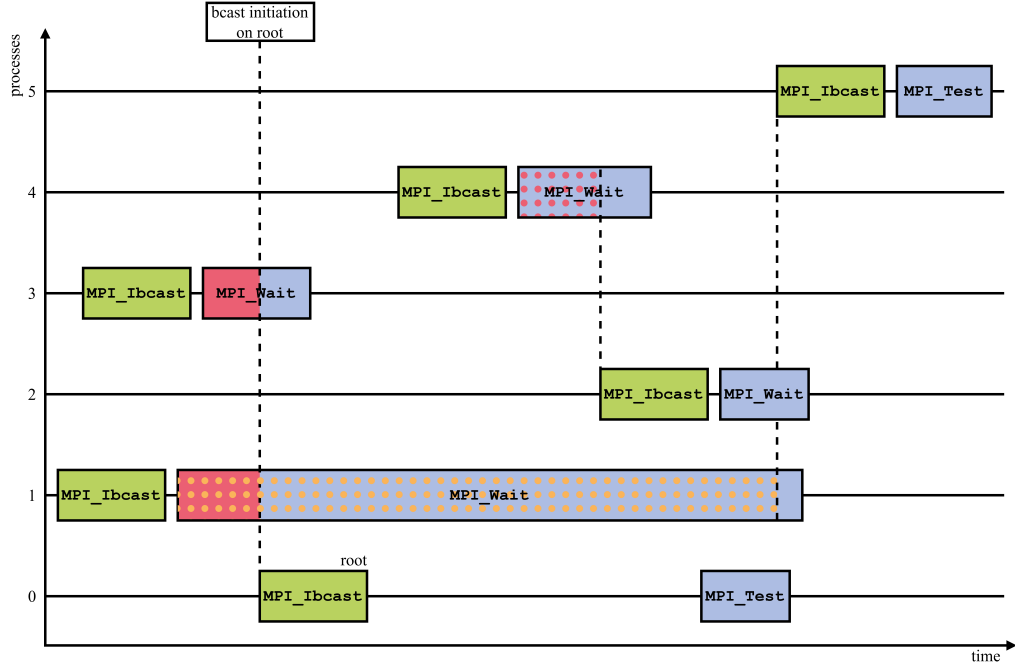
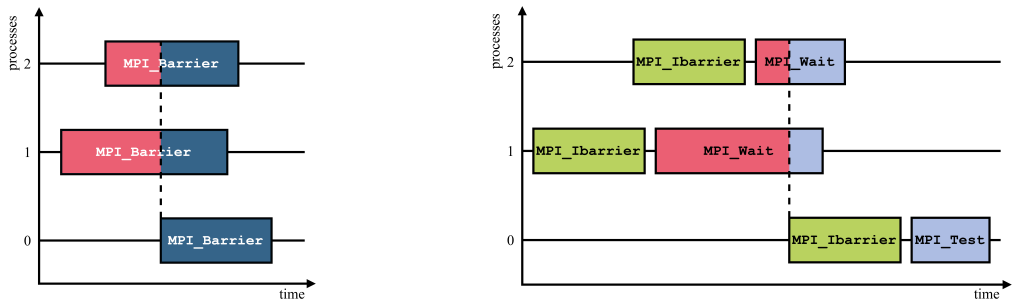


Figure 7: Wait states in nonblocking broadcast executed with the tree-based communication pattern from Figure 6, assuming a rendezvous protocol. Solid red marks Late Broadcast waiting time induced by a delayed root (rank 0), dotted red marks Late Broadcast waiting time induced by inner nodes acting as senders, and dotted orange marks Early Broadcast waiting time.

The corresponding wait state is called *Wait at Barrier*, illustrated in Figure 8. Communicator creation calls require all processes of the parent communicator to synchronize. Therefore, the wait state definition is similar to that of a barrier. However, the wait state is called *Wait at Communicator Create*.



(a) Wait at Barrier wait state in blocking collective communication.

(b) Wait at Barrier wait state in nonblocking collective communication.

Figure 8: Wait states in blocking and nonblocking barrier operations.

All-to-all wait states are defined in a similar manner, with the distinction that the MPI standard does not require all processes to synchronize; it merely permits it. If all processes synchronize, referred to as *full overlap*, the wait state is defined identically to the barrier case. If only a subset of processes synchronize, referred to as *partial overlap*, that subset

forms a group within which the same wait state definition applies. In both cases, the wait state is referred to as *Wait at $N \times N$* .

Early Reduce wait states are the counterparts to *Late Broadcast* wait states. They arise in all-to-one collective operations where many processes send data to a single root. The root cannot complete the operation until all non-root processes have contributed their data, so the time the root process spends inside a blocking all-to-one collective call, or inside the corresponding completion call, before the last non-root process has initiated the operation constitutes *Early Reduce* waiting time. Figure 9 illustrates this pattern.

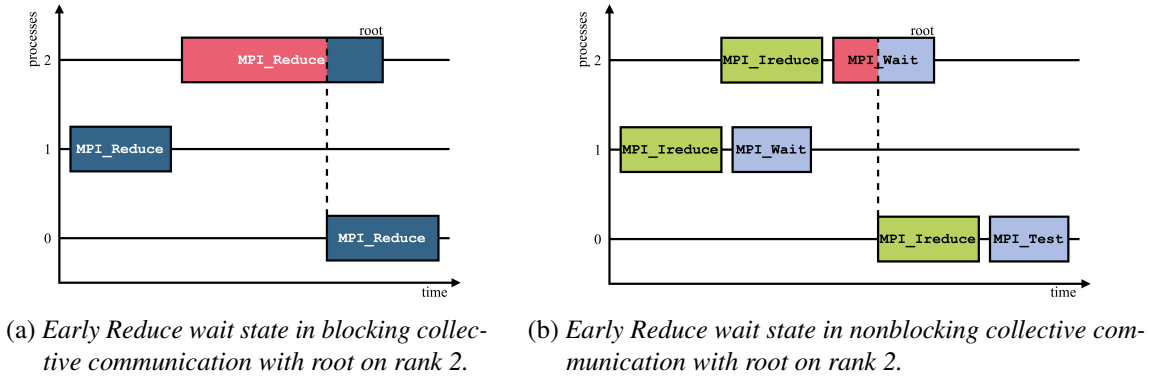


Figure 9: Wait states in blocking and nonblocking reduce operations.

As with broadcast, a rendezvous protocol may be used for some or all transfers, in which case a *Late Receiver* analogous wait state may be observed on non-root processes that must wait for the root to signal readiness before sending. Furthermore, the actual synchronization structure depends on the underlying algorithm. In a tree-based implementation, processes that take on root-like roles, collecting and forwarding partial results up the tree, may also become synchronization points, and the corresponding wait states are distributed across these processes rather than being confined to the root process alone.

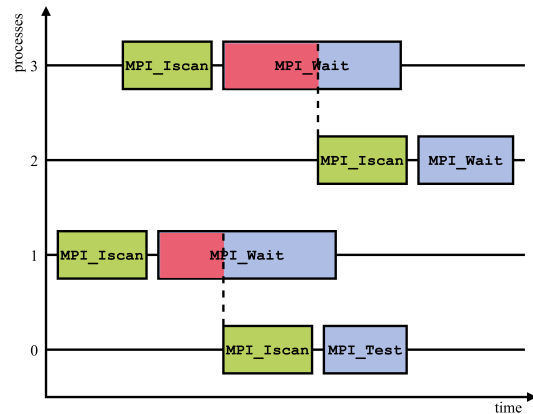


Figure 10: *Early Scan* wait state in non-blocking collective operation.

Scan and exclusive scan operations can exhibit *Early Scan* wait states, which are analogous to the *Early Reduce* wait state described above. In a scan operation, each rank i effectively acts as the root of a sub-communicator comprising ranks $0, \dots, i$. Consequently, the time that rank i spends waiting in a blocking scan or exscan call, or in the corresponding wait, before all ranks $0, \dots, i - 1$ have initiated the operation, is classified as *Early Scan* waiting time. Figure 10 illustrates this wait state. As with the reduce case, a more detailed

knowledge of the internal MPI synchronization structure could allow additional wait states to be identified, including a *Late Receiver* variant arising from the use of the rendezvous protocol for larger messages.

As already illustrated in Figure 6, a single call is not restricted to revealing only one class of wait state. This becomes particularly relevant for multi-completion calls. Figure 11 shows such a scenario, where multiple requests are completed in a single completion call. Message arrows have been omitted for clarity; identically colored initiation calls correspond to each other. Process 1 initiates a receive, a synchronous send, and a broadcast, then completes all three operations in a single MPI_Waitall call. The receive cannot complete until process 2 has initiated its send, revealing a *Late Sender* wait state (green, dotted). The broadcast cannot complete until the root (process 2) has initiated it, indicating a *Late Broadcast* wait state (orange, cross-hatched). Finally, the synchronous send can only be completed once process 3 has initiated the corresponding receive, exhibiting a *Late Receiver* wait state (red, solid). Since all three wait states occur within the same completion call, the waiting time of that call is determined by the most-severe wait state, in this case the *Late Receiver* wait state.

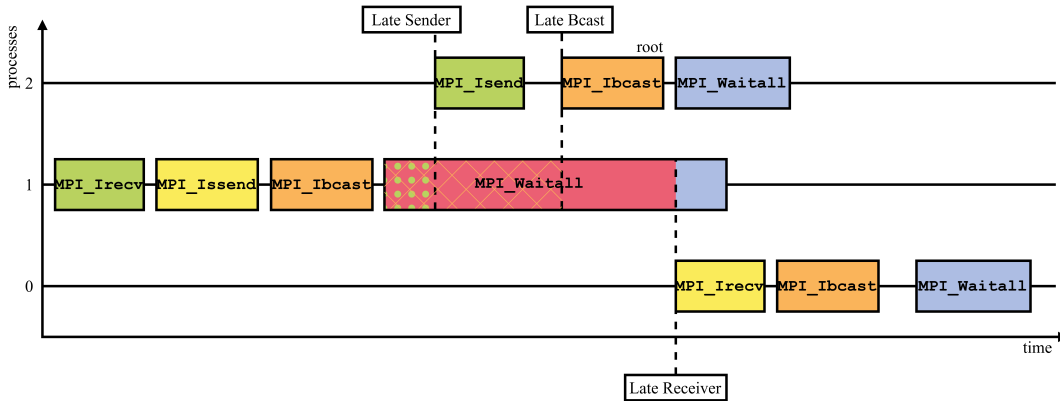


Figure 11: Wait states in a multi-completion call. Rank 1 incurs a *Late Sender* (green, dotted), *Late Broadcast* (orange, cross-hatched), and *Late Receiver* (red, solid) wait state. The call's waiting time is set by the most severe one.

Delay

A delay describes an interval during which one process performs additional work that its peer processes do not, causing it to arrive late at the next synchronization point with those processes. Such extra work can, for example, manifest as a computational imbalance, where one process spends more time in a region of code than the others, or it can arise from operations that run serially on only one process. A delay does not stay confined to the process where it occurs; it surfaces as wait states on other processes. This happens when a peer process idles because it reaches a synchronization point before the delaying process does. Collective communication amplifies this effect, since a single late process

can hold up every other participant at once. Wait states can, in turn, push back the start of subsequent communication operations and cause further wait states, further increasing the cost of the original delay. The cost of a delay is therefore defined as the total amount of waiting time it causes. Delay costs are further distinguished into short-term and long-term costs. Short-term delay costs are the amount of direct wait state the delay causes, that is, wait states caused directly by an additional activity on another process. Long-term delay costs are the amount of indirect wait state a delay causes, that is, wait states caused by another wait state. A wait state that causes another in this way is referred to as a propagating wait state. [6]

Figure 12 illustrates this. All ranks execute a routine `Foo`. Process 0 has an imbalance in that routine and stays there longer than the other ranks, causing it to enter the subsequent broadcast, where it serves as root, later than expected. The red zigzag area denotes the delay. Process 1 participates in that broadcast and incurs waiting time, marked in solid red, because process 0 starts it late. Process 1 then executes a reduce operation in which process 2 is the root. Since process 2 has already started the reduce operation after it leaves `Foo`, it incurs *Early Reduce* waiting time there. The wait state on process 1 is a direct wait state, reflecting the short-term cost of the delay on process 0. This wait state propagates to process 2, constituting the long-term cost of the same delay, and is therefore classified as an indirect wait state. The remaining execution time of the broadcast on process 1 is itself a further delay, causing the larger share of the *Early Reduce* waiting time on process 2. This portion is a direct wait state.

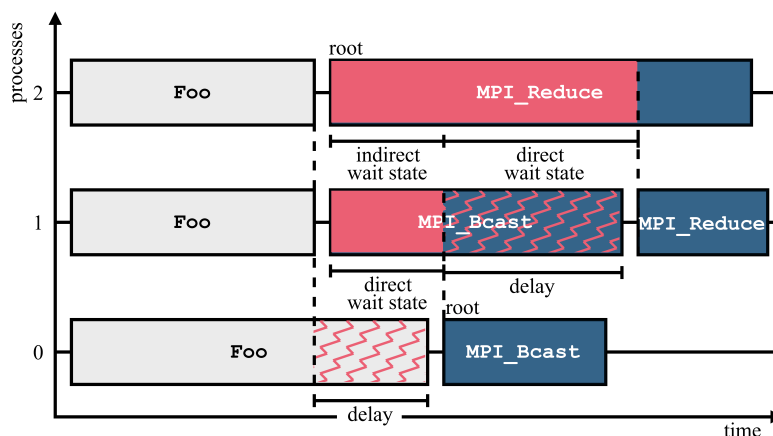


Figure 12: *Direct and indirect wait states resulting from a delay.*

Wait states can be understood as the symptoms and delays as the root cause. Given a global view of execution timing, diagnosing wait states is relatively straightforward, whereas identifying their root cause is harder, since a delay and the wait states it causes can be separated by a substantial distance in both time and rank.

Critical Path

The notion of the critical path originates from project management, where it denotes the longest sequence of dependent tasks that determines the minimum time needed to complete a project [7]. The same idea applies to parallel programs. The execution of a parallel program can be modeled as a program activity graph, which represents individual activities and both their duration and the order in which they must occur. An activity is defined as a single execution of a call path by a specific process. The critical path is the longest path through this graph and thus determines the program's total execution time.

By construction, the critical path itself contains no wait states. Activities on the critical path can, however, be the cause of wait states elsewhere in the program. A process that lies off the critical path may have to wait for an activity that lies on it. This has a direct consequence for optimization. Improving the runtime of activities that are on the critical path allows for potentially improving the total runtime of the application, whereas performance optimization of activities that are not on the critical path will only extend wait states but not improve the overall runtime. Shortening one of its activities can shift the critical path elsewhere in the graph so that the resulting reduction in overall execution time may be smaller than the time saved on the optimized activity itself. Knowing the critical path therefore identifies which activities are worth optimizing in principle, while predicting the actual benefit of an optimization requires knowing how it would reshape the critical path. [8]

2.2.2 Performance Analysis Tools

Identifying performance inefficiencies in parallel applications requires dedicated tools. Performance data gathered during execution can either be summarized as profile data or recorded in detail as event traces. Profiling tools accumulate statistics over the entire execution, reporting aggregated metrics such as total time per function, number of calls, or bytes transferred. The memory costs of profiling are roughly constant and independent of execution time. Tracing captures a chronological record of events during execution, preserving the dynamic application behavior. Measurement tools record timestamped events at well-defined points in the application execution, such as the entry and exit of code regions or for parallel API calls. The memory costs for tracing are proportional to the duration of the execution and the number of processes. Performance data can be collected via sampling or instrumentation. In sampling, the processor is interrupted at a fixed frequency, and the program counter is read to obtain the address of the current instruction. To get the total time per function, the sampling interval duration is added to the accumulated execution time of the current code region. It is also possible to get other metrics. The runtime overhead is controlled through the sampling rate. However, sampling

cannot ensure that every source code region is captured, which makes it less suited for tracing. [9]

Instrumentation adds measurement hooks to an application at specific points, ensuring that every instrumented code region is observable. Instrumentation introduces overhead at every instrumented point, and in programs with many short, frequently called functions, this overhead can accumulate significantly. To control this, measurement filters can be applied to exclude functions that would otherwise dominate the overhead without contributing useful information.

Well-established open-source tools for collecting HPC performance data include Score-P [4], Extrae [10], and HPCToolkit [11]. Score-P and Extrae support both instrumentation and sampling, while HPCToolkit is a sampling-only tool primarily used for profile generation. Score-P is the measurement infrastructure used in this work. It is compatible with the analysis tools Vampir [12], Scalasca [2, 3], Extra-P [13] and TAU [14]. Scalasca and Vampir are post-mortem tools that operate on OTF2 [5] trace files. The commercial tool Vampir provides interactive graphical trace visualization, while Scalasca automatically detects and categorizes performance bottlenecks, making it particularly suited to large-scale traces. Extra-P is a performance modeling tool aimed at identifying scalability bottlenecks. TAU is a comprehensive tool framework covering instrumentation, measurement, and analysis; it can be used standalone or integrated with Score-P. As a part of TAU, the Program Database Toolkit offers automatic instrumentation, ParaProf [15] provides profile visualization, and PerfExplorer [16] enables data mining. Extrae is the measurement backend of the Barcelona Supercomputing Center tool suite, whose post-mortem analysis tools include the trace visualizer Paraver [17] and the network simulator Dimemas [18]. Beyond open-source tools, major vendors provide their own performance analysis environments. The now-discontinued Intel Trace Analyzer and Collector (ITAC) [19] targeted CPU and MPI performance, while NVIDIA Nsight [20] and AMD ROCprofiler SDK [21] are primarily designed for GPU workloads, offering deep insight into kernel-level and CPU–GPU interaction performance. These tools generally provide more fine-grained hardware-specific information, but are tied to their respective platforms and are not designed for the large-scale multi-node analysis targeted by Score-P and Scalasca.

Score-P

Score-P [4] is an open-source, community-driven instrumentation and measurement infrastructure for parallel HPC applications. It was developed jointly by six institutions to address the fragmentation of the performance tools landscape, where each tool previously maintained its own proprietary measurement infrastructure. By providing a unified instrumentation layer with common output formats, Score-P enables interoperability between a range of complementary analysis tools. Score-P supports both profiling and event tracing.

The output depends on the user’s specifications. Figure 13 shows the Score-P workflow. The Score-P libraries and the general data collection infrastructure must be linked into the executable. Score-P intercepts the entry and exit points of functions and parallel API calls. For MPI applications, this is achieved via the MPI profiling interface (PMPI). When executing the instrumented application, Score-P collects the performance data. All measurement artifacts are stored in a dedicated experiment archive directory. Depending on the measurement configuration, this archive contains either a CUBE4 call-path profile, an OTF2 event trace, or both.

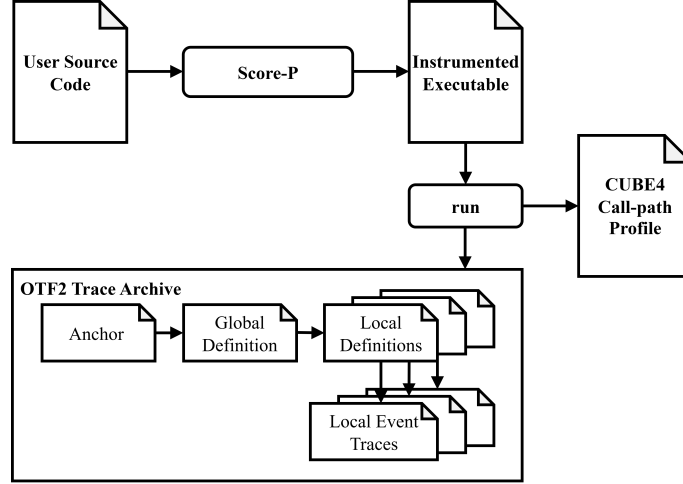


Figure 13: *Score-P usage workflow.*

Since clocks on distributed HPC systems are generally not synchronized, Score-P records clock offsets at MPI initialization and finalization and uses linear interpolation to align timestamps across processes in tracing mode. In practice, clock frequencies can vary during a measurement, leaving residual inconsistencies. The CLC library [22, 23], which is a Scalasca component, addresses this by exploiting happened-before relationships encoded in the trace events.

OTF2 Event Model

The Open Trace Format 2 (OTF2) [5] is a scalable, memory-efficient event trace data format designed for large-scale HPC applications. It represents performance data using two complementary concepts: *definitions* and *events*. Definitions describe objects that are relevant over a longer period of execution and do not carry a timestamp. They provide context for events, for example, MPI communicators or code regions. Region definitions describe function-like entities and can be associated with a programming paradigm such as MPI. Events, by contrast, are timestamped records of what happened at a specific point in time on a specific location. A location is a specific execution instance, such as a process or a thread. Location groups represent a single address space and correspond to an MPI process. Each event carries a numerical identifier, a location identifier, a timestamp, and

optionally additional event-specific attributes, which can reference definition objects. For every instrumented code region, Enter and Leave events are recorded, which carry a reference to a region definition (i.e., its numeric identifier) as an event-specific attribute.

The memory efficiency of OTF2 is achieved by separating data into distinct files to avoid redundancy. Figure 13 illustrates the structure of an OTF2 trace archive. It contains an anchor file to manage the trace data, a global definitions file, and separate local trace files and local definition files for each location. To avoid additional communication during data collection, each location uses separate identifiers in its event trace file to reference definition records. At the end of the measurement, Score-P aggregates and unifies the local definitions across all locations, removing redundancies and assigning unique global identifiers to each of the remaining definitions. These global definitions are stored in a separate file, and a mapping between the local identifiers and the global ones is stored in the local definitions file, which additionally holds the definitions that are specific to a location. [24]

The OTF2 event model defines dedicated events for MPI calls, including events for point-to-point messages and collective communication operations, with distinct events for the blocking and nonblocking variants, respectively.

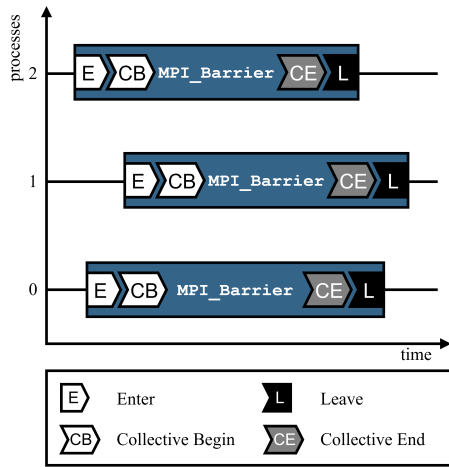


Figure 14: OTF2 events recorded for blocking collectives, shown for an MPI_Barrier.

Figure 14 shows the events recorded by Score-P for a blocking collective operation. Upon entering the procedure call, an Enter (E) event is recorded, carrying the identifier of a region definition, in this case that of MPI_Barrier. This is followed by a Collective Begin (CB) event that has no event-specific attributes. After the collective operation completes, a Collective End (CE) event is written, carrying call-specific information. This includes the collective operation type, the reference to a communicator definition, the root rank where applicable, and the number of bytes sent and received.

For a barrier, the root rank is not applicable, and both byte counts are zero. The collective operation type field identifies the kind of collective without having to look up the region name in the associated Enter event; in this case it is set to BARRIER. Finally, a Leave (L) event is recorded, carrying the region identifier.

For nonblocking collective operations similar events are recorded, as depicted in Figure 15. At the initiation call, a Collective Request (CR) event is recorded, holding a request identifier that uniquely identifies the pending operation. After completion, a Collective Complete (CC) event is recorded, holding the same information as the Collective End

event, and additionally the request identifier, which allows the completion to be matched to its corresponding initiation.

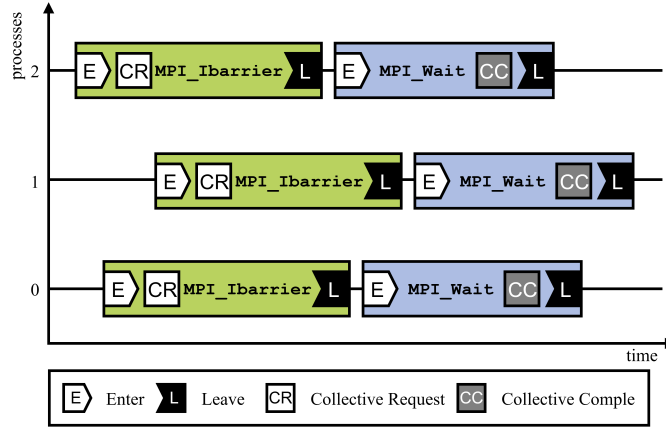


Figure 15: OTF2 events recorded for nonblocking collectives, shown for an `MPI_Ibarrier`.

The Collective Begin and Collective Request events are recorded before the corresponding PMPI call is issued and the Collective End and Collective Complete after the respective PMPI call returns. This ensures that the operation has completed at the time of recording. Consequently, the timestamps of the begin and request events represent the earliest possible time at which the operation could have started, while the end and complete timestamps represent the latest possible time at which it could have finished. Collective Complete events can be recorded in wait calls and in successful test calls. For every operation completed in a multi-completion call, such as `MPI_Waitall`, a separate complete event is recorded in the region of that call. The order of the complete events aligns with the order of the requests in the array that was handed over to the call and does not reflect the actual completion order. Since the complete events are written after the multi-completion call returns, their timestamps all fall at roughly the same point: the moment the last pending operation completed.

For persistent collective operations, the same event model can be applied. A Collective Request (`CR`) event would be recorded in the `MPI_Start` call and a Collective Complete (`CC`) in the corresponding wait or test call. These two events are not MPI-specific and could, in principle, be used for other parallel programming paradigms that offer nonblocking collectives. However, currently MPI is the only paradigm for which Score-P records these events.

The event model forms the foundation for the trace analysis described in Chapter 3. As the internal messages exchanged by the MPI library to carry out a collective operation are not recorded by Score-P, the analysis must rely on conservative assumptions. That is, the wait states defined earlier in Section 2.2.1, relying on the knowledge of internal communication patterns, cannot be detected. For those wait states that are characterizable from the introduced event model, Table 2 lists their formal definitions for nonblocking

collective operations. The following notations are used for the remainder of the thesis. Let $a_X(p)$ denote an attribute of event X on process p . For example, the timestamp of event X on process p is denoted by $t_X(p)$. Let E_Y denote the Enter event of the subscribed region Y .

Wait State	Coll. Op. Type	Formal Definition
Wait at Barrier Wait at Comm. Create	BARRIER, CREATE_HANDLE	$p_{\text{latest}} = \arg \max_{p \in \text{comm}} t_{\text{CR}}(p)$ $\text{WT}(p) = \begin{cases} t_{\text{CR}}(p_{\text{latest}}) - t_{E_{\text{wait}}}(p) & \text{if } t_{\text{CR}}(p_{\text{latest}}) > t_{E_{\text{wait}}}(p) \\ 0 & \text{otherwise} \end{cases}$
Late Broadcast	BCAST, SCATTER[V]	$\text{WT}(p) = \begin{cases} t_{\text{CR}}(p_{\text{root}}) - t_{E_{\text{wait}}}(p) & \text{if } t_{\text{CR}}(p_{\text{root}}) > t_{E_{\text{wait}}}(p) \wedge \text{bytesRcvd}_{\text{CC}}(p) > 0 \\ 0 & \text{otherwise} \end{cases}$
Early Reduce	GATHER[V], REDUCE	$p_{\text{latest}} = \arg \max_{\substack{p \in \text{comm} \setminus \{p_{\text{root}}\} \\ \text{bytesSent}_{\text{CC}}(p) > 0}} t_{\text{CR}}(p)$ $\text{WS}(p_{\text{root}}) = \begin{cases} t_{\text{CR}}(p_{\text{latest}}) - t_{E_{\text{wait}}}(p_{\text{root}}) & \text{if } t_{\text{CR}}(p_{\text{latest}}) > t_{E_{\text{wait}}}(p_{\text{root}}) \\ 0 & \text{otherwise} \end{cases}$
Wait at $N \times N$	ALLGATHER[V], ALLREDUCE, ALLTOALL, REDUCE_SCATTER[_BLOCK]	$p_{\text{latest}}^{\text{full}} = \arg \max_{p \in \text{comm}} t_{\text{CR}}(p), \quad p_{\text{latest}}^{\text{partial}} = \arg \max_{\substack{p \in \text{comm} \\ \text{bytesSent}_{\text{CC}}(p) > 0}} t_{\text{CR}}(p)$ $\text{WT}(p) = \begin{cases} t_{\text{CR}}(p_{\text{latest}}^{\text{full}}) - t_{E_{\text{wait}}}(p) & \text{if full overlap} \wedge t_{\text{CR}}(p_{\text{latest}}^{\text{full}}) > t_{E_{\text{wait}}}(p) \\ t_{\text{CR}}(p_{\text{latest}}^{\text{partial}}) - t_{E_{\text{wait}}}(p) & \text{if partial overlap} \wedge t_{\text{CR}}(p_{\text{latest}}^{\text{partial}}) > t_{E_{\text{wait}}}(p) \\ & \wedge \text{bytesRcvd}_{\text{CC}}(p) > 0 \\ 0 & \text{otherwise} \end{cases}$
Early Scan	EXSCAN, SCAN	$p_{\text{latest}}(p) = \arg \max_{\substack{q \in \text{comm}, q < p \\ \text{bytesSent}_{\text{CC}}(q) > 0}} t_{\text{CR}}(q)$ $\text{WT}(p) = \begin{cases} t_{\text{CR}}(p_{\text{latest}}(p)) - t_{E_{\text{wait}}}(p) & \text{if } t_{\text{CR}}(p_{\text{latest}}(p)) > t_{E_{\text{wait}}}(p) \\ 0 & \text{otherwise} \end{cases}$

Table 2: Formal definitions of wait states in nonblocking collective operations based on OTF2 events, assuming the absence of clock condition violations.

3 Replay-based Performance Analysis

This chapter introduces replay-based performance analysis as a technique for detecting and quantifying wait states in parallel programs, using the Scalasca Trace Tools as a concrete example.

Scalasca [2, 3] is an open-source software tool designed to assist users in improving 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 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 proportionally to the number of cores.

3.1 Scalasca Infrastructure Overview

Scalasca’s core component is the Scalasca Trace Analyzer (SCOUT). It comes in four variants: serial, pure MPI, pure OpenMP, and hybrid OpenMP+MPI. OpenMP support comes with several restrictions. Nested parallelism and untied tasks are not supported, and the number of threads must be consistent across all parallel regions. Hybrid OpenMP+MPI programs must have been measured using `MPI_THREAD_FUNNELED`, which restricts MPI calls to the main thread¹. Although traces containing OpenMP tasking events can be read, no task-specific analysis is performed, and wait states in OpenMP barriers as well as critical-path and root-cause results are known to be incorrect in the presence of tasks.

Figure 16 gives an overview of the SCOUT analysis workflow. Frames group the steps by the Scalasca component they belong to. Within each frame, the vertical rectangles on the left label analysis phases, and the horizontal rectangles are the individual sub-steps. Rectangles with an arrow tip represent trace replay steps. The direction of the arrow indicates whether the replay runs forward (right) or backward (left) through the trace. Gray arrow tips mark local replays with no MPI communication, while colored arrow tips denote replays that exchange data between analysis processes via MPI. Rectangles with a folded corner represent artifacts, that is, files read as input or written as output.

The input is an OTF2 experiment archive generated by Score-P. To read the trace data, SCOUT uses the Parallel Event Analysis and Replay Library (PEARL) [25]. It provides a set of parallel abstractions to access and process event trace data from OTF2 archives as well as functionality to transfer data between multiple processes. Reading the trace data

¹The thread that called `MPI_Init` or `MPI_Init_thread`.

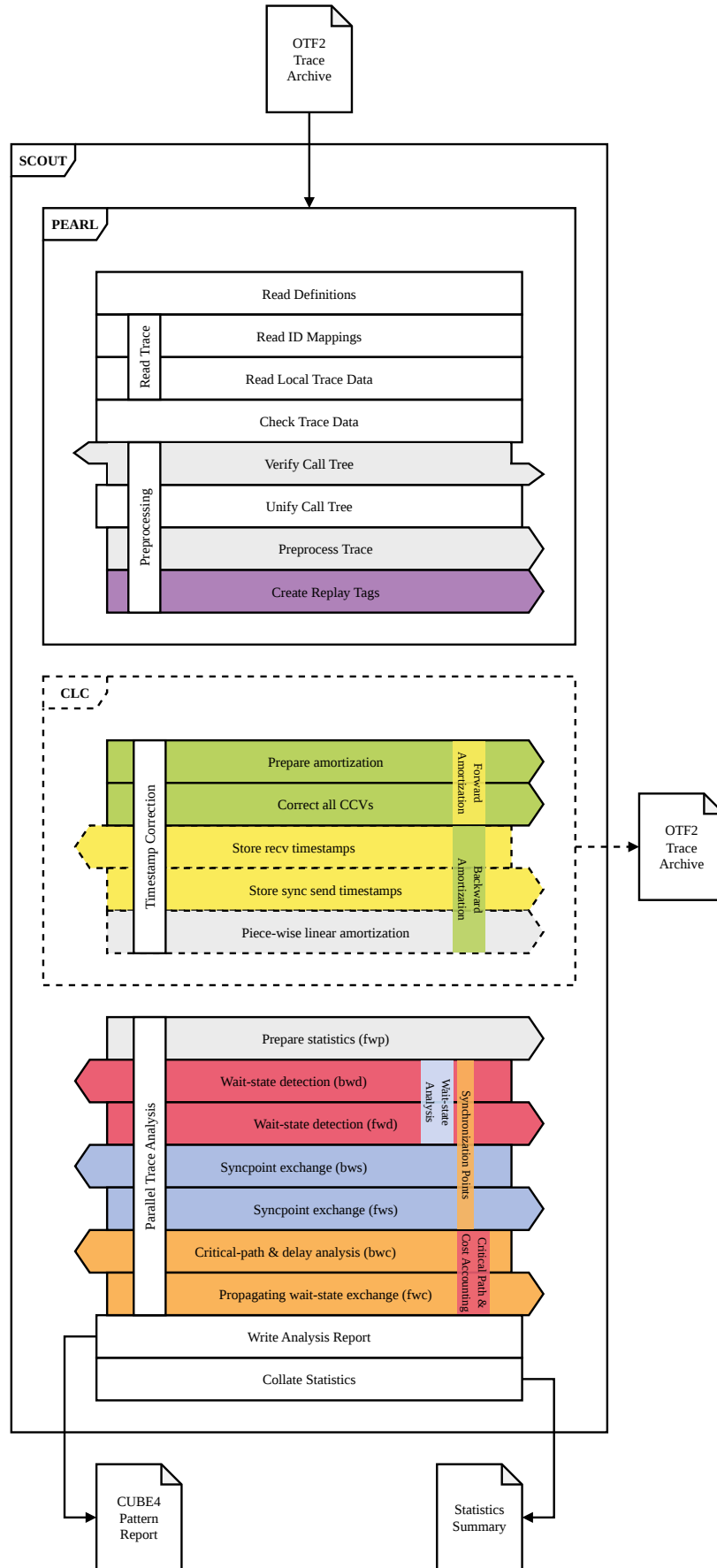


Figure 16: Overview of the SCOUT analysis workflow.

begins at the MPI process level. Each analysis process opens the OTF2 archive and reads in the global definitions. Based on the MPI rank of the analysis process, the corresponding location group is identified. From this point on, work becomes thread-parallel in the hybrid case. Each OpenMP thread within a process is responsible for one location. Each location opens its associated local definitions and trace container files from the OTF2 archive. As described in Section 2.2.2, OTF2 stores definitions using location-local identifiers in the local event streams. The identifier mappings, stored in the local definitions file, provide the translation between these local identifiers and the globally unified definition identifiers. Next, the actual event trace data for each location is read into memory. Each thread reads the event stream of its corresponding location, using the identifier mappings to resolve local references to global definitions and constructing a local trace object that holds the complete sequence of events for that location. Finally, the trace container is closed, and the file handles are released. At this point, all event data resides in memory, and no further file access is needed. In the subsequent *check trace data* step, each thread validates its trace against analysis requirements for OpenMP, for example, that MPI-specific events occur only on the main thread.

The preprocessing phase prepares the in-memory trace data and consists of four steps. First, the local call tree is verified. The trace is first traversed backward to move any buffer flush events to their correct positions. Then each thread performs a forward traversal of its local event stream to reconstruct the call tree from Enter and Leave events and checks for unbalanced Enter/Leave pairs. The resulting per-thread call trees are merged hierarchically per process. Next, the call-tree unification is performed on the main thread. If the call-tree verification step modified the local call tree, the call trees of all MPI processes are merged into a single global call tree that is identical on every rank. This uses a hierarchical reduction followed by a broadcast of the result. If the original call tree was already complete—which is the case when profiling was enabled during trace collection—this step is skipped. Next, the trace data itself is preprocessed, which establishes the full trace-access functionality. For events associated with nonblocking operations, an iterator syntax is set up that allows access to the offset to the previous/next event using the same request. In addition, each Enter/Leave event in the local event stream is linked to its corresponding node in the call tree. In the fourth step, a forward replay is performed on the main threads to generate artificial replay tags for point-to-point communication events. MPI permits multiple pending point-to-point messages to have the same message envelope. To facilitate correct message matching of the analysis messages during backward replays, unique artificial tag values are assigned to the events associated with each message. In addition, a reserved tag value is defined for use by additional messages exchanged during delay and critical-path analyses, preventing interference with the point-to-point replay messages.

Optionally, a timestamp correction can be applied before analyzing the trace. Event records collected on different compute nodes can have discrepancies due to clock drift or different clock speeds. The timestamp correction [22, 23] ensures the correctness of the logical order of events. The timestamp correction is performed by the Conditional Logical Clock (CLC) library and consists of two phases: forward amortization, which is always applied, and backward amortization, which is only performed if the forward pass detects violations of the logical event order. Those violations are referred to as clock condition violations (CCV), for example, if the timestamp of a send lies after that of its corresponding receive. If enabled, the corrected trace can optionally be written back to disk before analysis proceeds.

The core of the analysis is the parallel trace replay. Throughout the replay passes, the trace is traversed either chronologically or in reverse. When a communication event is encountered, the original communication operation is mimicked using a similar MPI call, transmitting performance data rather than the original buffer content. In a backward replay, the communication direction is inverted, resulting in a Send event triggering the posting of a receive and vice versa. The backward pass is required for delay and critical-path analyses and also enables the detection of *Late Receiver* patterns. The analysis performs a forward and backward replay to detect all wait states. Simultaneously, synchronization information is stored on the ranks exhibiting the wait states. In the subsequent synchronization point exchange phase, the locally stored synchronization point information is exchanged in two steps. First, the synchronization points for *Late Sender* wait states determined by the receivers are propagated to the corresponding sender ranks during a backward pass. Second, in the forward pass of that phase, the synchronization points for *Late Receiver* wait states, detected on the sender side, are transferred to the corresponding receiver ranks. For collective communication, all synchronization points are recorded during the forward wait-state detection pass. The correct identification of synchronization points is a prerequisite for delay and critical-path analyses, as they determine where analysis messages must be exchanged between processes and which time intervals are accountable for a given inefficiency. Next, the delay and critical-path analyses are conducted in a backward replay. The final replay is necessary to classify the detected wait states.

The replay mechanism uses a publisher-subscriber callback pattern. As each event is processed, all functions registered for that event type are called in sequence. These callbacks can trigger further user-defined events, whose own callbacks execute before the replay processes the next event. To pass information between callbacks processing the same event, a callback data object is available. The callback data is only valid during the processing of the current event of the local trace. Roles provide a way to attach labeled information to the callback data. For example, the role `ROLE_ENTER_SENDCMP` can be used to add the Enter event of a region containing either a Send or Send Complete event. This lets pattern detection algorithms handle nonblocking and blocking events similarly.

For point-to-point events, the data labeled with roles is transmitted to the peer process as part of the analysis message, making it available in the remote callback data on the receiving side. Different callbacks can attach data to the callback data object, which enables each message to be replayed only once even when it is relevant to multiple patterns. For point-to-point communication, a PRE user event is fired before the analysis message is sent, giving callbacks the opportunity to populate the callback data with the information to be transmitted. On the receiver side, the message is received, and a POST user event is fired, upon which registered callbacks can process the received data.

SCOUT implements several handlers that define callback functions, trigger user events, and map callbacks to the events they belong to. The communication handler processes MPI communication events, exchanges analysis messages between processes, and fires user events to notify downstream handlers. The pattern detection responds to these user events, identifies inefficiency patterns, computes their severity, and accumulates the results in a per-pattern, per-call-path matrix stored on the analysis thread/process responsible for the affected thread/rank. For simplicity, the following description uses MPI terminology (i.e., process and rank), but the concept applies equally to OpenMP threads. The syncpoint handler records synchronization point information for use in the subsequent delay and critical-path analysis pass. Once all replay passes are complete, the local results are combined into a three-dimensional matrix of pattern, call path, and rank, which is written to a CUBE4 [26] report and can be explored interactively in CubeGUI. In addition, a statistics file is produced containing the most-severe instances and additional statistics. These statistics are calculated using a quartile approximation [27], which requires values to be normalized to the range $[0, 1]$ to minimize numerical errors. To support this, the prepare statistics forward replay pass determines the maximum time spent in functions that may exhibit wait states. This pass is marked in grey, as it does not perform a typical trace replay involving communication; instead, it tracks the maximum time per location and obtains the global maximum via a single MPI_Allreduce after all events have been processed. The details of how the analyses detect and quantify specific inefficiency patterns are explained in the following paragraphs, focusing on blocking collective operations as an example.

3.2 Wait-state Analysis

The Scalasca wait-state analysis [2, 3] consists of a forward and a backward replay pass. The forward wait-state detection pass identifies all wait states except *Late Receivers*, including *Late Sender* instances and wait states in collective operations. The backward wait-state detection pass detects *Late Receiver* wait states. Quantifying wait states requires timestamp information of events stored on remote processes. In order to exchange these, Scalasca uses MPI communication mirroring the communication of the original application.

Once the relevant timestamps have been gathered, each process can compute its waiting time independently.

During both passes, synchronization points are recorded for each detected wait state. For this purpose, two maps are employed. The syncpoints map associates an event with a syncpoint info object containing the waiting time, completion time, and total time spent in the operation. The syncranks map associates the same event with the set of ranks it synchronizes with. For this purpose, a bitfield of size equal to the number of location groups (i.e., processes) is used to mark the synchronizing ranks, with memory optimizations applied depending on the number of synchronizing ranks. In all cases, ranks are stored as global ranks, that is, their rank in `MPI_COMM_WORLD`.

3.2.1 Backward Wait-state Detection

The backward wait-state detection replay identifies *Late Receiver* patterns. Figure 17 shows such an inefficiency in the completion of a nonblocking send. The black arrow represents the original communication, and the orange arrow indicates the analysis communication in the backward wait-state detection pass.

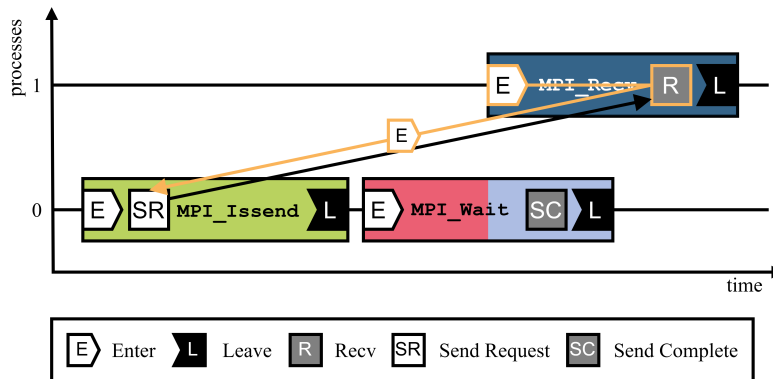


Figure 17: Detection of Late Receiver pattern in the backward wait-state detection replay pass.

When the analysis process responsible for rank 1 encounters the Recv `R` event, the communication handler fires the `PRE_RECV` user event, populating the callback data with the Recv event and the Enter and Leave events of the enclosing region. A nonblocking send is then initiated on rank 1 to transmit the callback data, using the communicator, source rank, and replay tag stored in the Recv event. Point-to-point analysis messages use a fire-and-forget approach: after the send is initiated, the corresponding request is added to a vector containing the pending requests. A progress callback registered for Send [Request], Recv [Complete] and Collective End events calls `MPI_Testsome` to complete pending requests and removes them from the vector. Once all events have been processed, a `FINISHED` user event is triggered to complete any remaining requests. If requests cannot be completed and must be cancelled, a warning is printed, indicating either a corrupted trace or a bug in the analysis. Meanwhile, the analysis process responsible

for rank 0 executes the first callback when encountering the Send Request $\boxed{\text{SR}}$ event. A PRE_SEND event is fired, populating the callback data with the relevant events required by subsequent callbacks, including the Enter and Leave events of the region containing the Send Complete event, in this case, MPI_Wait. To access the complete event when processing the request event, the iterator syntax established during preprocessing is used. Rank 0 posts a blocking receive to obtain rank 1's event data and fires the POST_SEND user event. A callback in the *Late Receiver* pattern detection then computes the waiting time as

$$\text{WT}(p_s) = \begin{cases} t_{\text{E}_{\text{recv}}}(p_r) - t_{\text{E}_{\text{wait}}}(p_s) & t_{\text{E}_{\text{recv}}}(p_r) > t_{\text{E}_{\text{wait}}}(p_s) \\ & \wedge t_{\text{L}_{\text{wait}}}(p_s) > t_{\text{E}_{\text{recv}}}(p_r) \\ 0 & \text{otherwise} \end{cases} \quad (3.1)$$

Here, p_s corresponds to rank 0 and p_r to rank 1. On rank 0, all timestamps required for this calculation are available through the callback data. The first condition $t_{\text{E}_{\text{recv}}}(p_r) > t_{\text{E}_{\text{wait}}}(p_s)$ ensures that no negative waiting time is computed, which would otherwise occur if the wait for send completion began after the receive. The second condition $t_{\text{L}_{\text{wait}}}(p_s) > t_{\text{E}_{\text{recv}}}(p_r)$ checks whether the send completion region overlaps with the receive, which indicates a rendezvous protocol was used. Only in that case, the definition for the *Late Receiver* wait state applies. In this example, the synchronous send enforces rendezvous, and since the receive was posted after the send completion began, waiting time is incurred on rank 0 (marked in red). To store a synchronization point, the completion time is required. If a positive waiting time is determined by Equation (3.1), the completion time can be computed as

$$\text{CT}(p_s) = \begin{cases} t_{\text{L}_{\text{wait}}}(p_s) - t_{\text{L}_{\text{recv}}}(p_r) & \text{if } t_{\text{L}_{\text{wait}}}(p_s) > t_{\text{L}_{\text{recv}}}(p_r) \\ 0 & \text{otherwise} \end{cases} \quad (3.2)$$

In the discussed example, the completion time on rank 0 is zero, since the condition $t_{\text{L}_{\text{wait}}}(p_s) > t_{\text{L}_{\text{recv}}}(p_r)$ is not satisfied. The LATE_RECEIVER user event is then triggered, upon which the *Late Receiver* pattern stores the waiting time under the call-path node of the send completion region. The syncpoint handler also listens for LATE_RECEIVER and stores a synchronization point on the sender side with the receiver's rank in syncranks. Although the wait-state detection is triggered at the Send Request event, the wait state belongs to the send completion region; therefore, the synchronization point is stored for the Send Complete event.

3.2.2 Forward Wait-state Detection

In the forward replay pass, the local event streams are traversed in chronological order. This section first briefly covers *Late Sender* detection before turning to collective wait states, using the *Wait at Barrier* pattern as a detailed example.

Late Sender Detection

The detection of *Late Sender* instances is conceptually similar to that of *Late Receivers*. Encountering a `Send` or `Send Request` event triggers the initiation of a send operation that transmits the associated event data required to calculate the waiting time on the receiver side. When the corresponding `Recv` or `Recv Complete` event is reached, the transmitted data is received and the `POST_RECV` user event is fired. The waiting time on the rank replaying the `Recv` event is then computed via

$$WT(p_r) = \begin{cases} t_{E_{lb}} - t_{E_{recv}}(p_r) & \text{if } t_{E_{send}}(p_s) > t_{E_{recv}}(p_r) \\ 0 & \text{otherwise} \end{cases} \quad (3.3)$$

where $t_{E_{lb}} = \min(t_{E_{send}}(p_s), t_{L_{recv}}(p_r))$ caps the waiting time to the time actually spent inside the receive region. This guard is necessary when $t_{E_{send}}(p_s) > t_{L_{recv}}(p_r)$ is fulfilled, which indicates the presence of a clock condition violation. In the case of nonblocking events, the corresponding `Enter` and `Leave` events of the send request and receive completion regions are used. The calculated waiting time corresponds to the time the receiver spends in its receive or receive completion call before the sender initiates its send. If waiting time is present, the analysis process responsible for the receiver side additionally computes the completion time, which is nonzero only if the receive call ended after the send. If a wait state is detected, the `LATE_SENDER` user event is triggered. The function called upon that event stores the waiting time in the *Late Sender* pattern alongside the call path of the `Recv` or `Recv Complete` region. Additionally, this event triggers the synchronization point handler to store a synchronization point on the receiver's process.

All detected *Late Sender* instances are stored in a ring buffer holding the most recent occurrences of the pattern. This serves to identify *wrong-order* situations, where messages are sent in a different order than they are received. A callback registered for `POST_RECV` compares the timestamp of the remote send associated with the currently processed receive event against those stored in the ring buffer to determine whether a *Late Sender* instance correlates with a wrong-order situation.

Another `POST_RECV` callback checks for clock condition violations. If timestamp correction was applied beforehand, such violations should no longer be present. The check verifies that

$$t_R(p_r) \geq t_S(p_s) \quad (\text{blocking}) \quad (3.4)$$

$$t_{RR}(p_r) \geq t_{SR}(p_s) \quad (\text{nonblocking}) \quad (3.5)$$

If this condition is violated, the `CCV_P2P` user event is fired, which accumulates the total number of violations and reports them to the user after the analysis completes.

Wait-state Detection in Blocking Collectives

For blocking collectives, the details of the detection mechanism differ. For explanation purposes, the example in Figure 14 (see page 20) is considered. There, a blocking barrier is executed on all three processes. Figure 18 illustrates the corresponding sequence of callbacks involved in the wait-state detection.

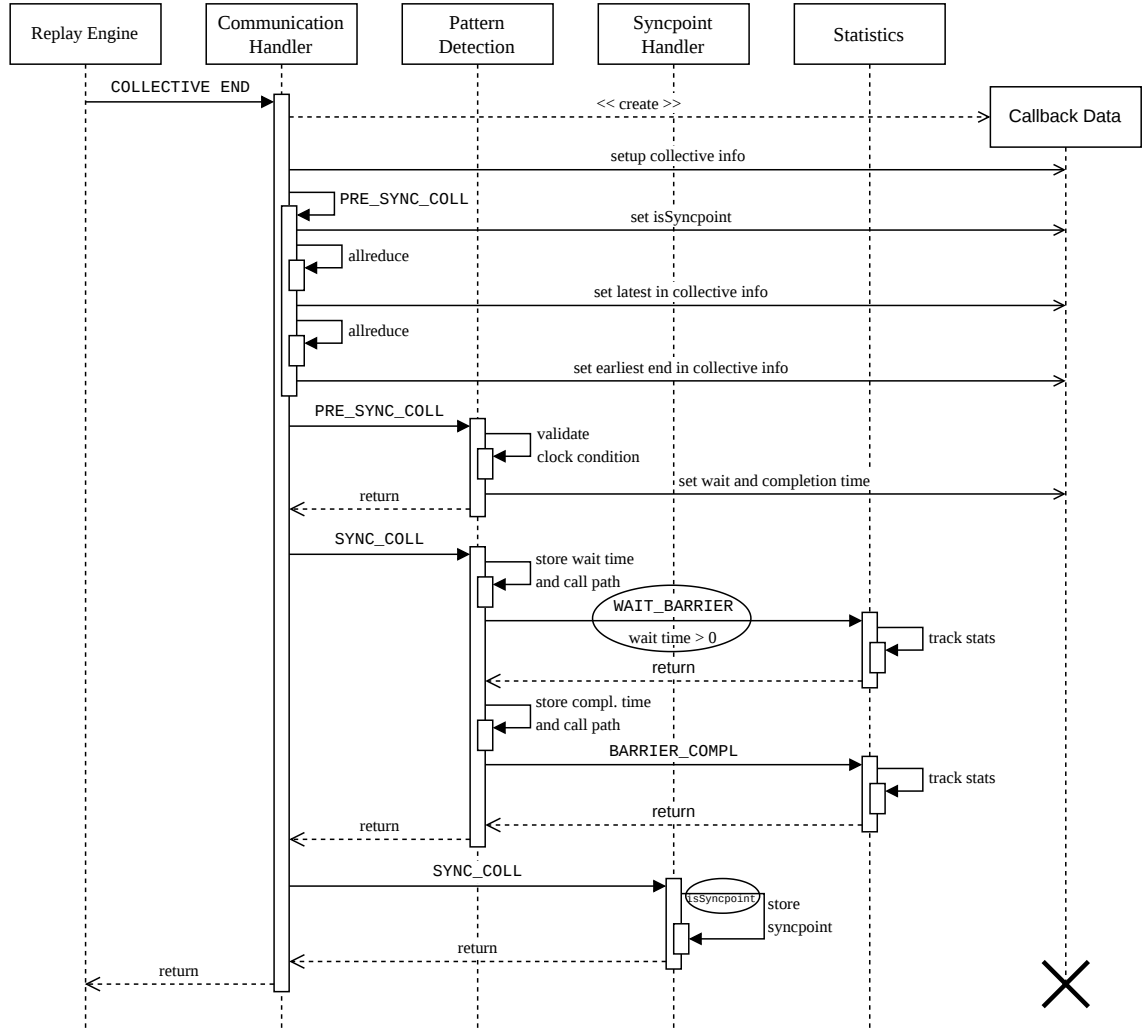


Figure 18: Callbacks involved in Wait at Barrier wait-state detection.

During the forward replay, the first callback relevant to collective wait-state detection is called on the Collective End event. At this point, the collective operation type is determined. For one-to-all, all-to-one, all-to-all, and prefix reduction operations, processes with zero payload exit the analysis early, since without a payload the synchronization conditions do not apply to them. The same is done for collectives on communicators involving only a single process. Then, the callback data is populated with the relevant events, including the Enter, Leave, Collective Begin, and Collective End events of the collective region, as well as a CollectiveInfo object. This object holds timestamp-rank pairs for several roles: my is initialized with the Collective Begin timestamp and

the global rank of the local process; latest, earliest_end, and root are assigned values during the subsequent callbacks. Depending on the collective operation type, specific user events are triggered. For a barrier, the PRE_SYNC_COLL and SYNC_COLL user events are fired. The PRE_SYNC_COLL callback in the communication handler first determines whether the current event is part of a synchronization point. For a barrier, all ranks synchronize by definition, so the Collective End events on all ranks have a synchronization point, and each rank stores a boolean value indicating this in its callback data. Two MPI_Allreduce calls are then issued to exchange the required timestamps for the waiting and completion time computation across all analysis processes. The first uses MPI_MAXLOC to determine the latest Collective Begin timestamp $t_{CB}(p_{\text{latest}})$ and its associated rank. The second uses MPI_MINLOC to find the earliest Collective End timestamp $t_{CE}(p_{\text{earliest}})$. Both reductions use the communicator stored in the event data. The pattern detection picks up the PRE_SYNC_COLL event, validates the clock condition $t_{CB}(p_{\text{latest}}) < t_{CE}(p)$, and, in case of a violation, fires the CCV_COLL event (not shown) to track the number of collective clock condition violations over the whole duration. It then computes the waiting and completion times as

$$t_{CB_{\text{latest}}} = \min(t_{CB}(p_{\text{latest}}), t_{CE}(p)), \quad p_{\text{latest}} = \arg \max_{p \in \text{comm}} t_{CB}(p)$$

$$WT(p) = t_{CB_{\text{latest}}} - t_{CB}(p) \tag{3.6}$$

$$t_{CE_{\text{earliest}}} = \max(t_{CE}(p_{\text{earliest}}), t_{CB_{\text{latest}}}), \quad p_{\text{earliest}} = \arg \min_{p \in \text{comm}} t_{CE}(p)$$

$$CT(p) = t_{L_{\text{barrier}}}(p) - t_{CE_{\text{earliest}}} \tag{3.7}$$

and stores it in the callback data. The minimum and maximum clauses in $t_{CB_{\text{latest}}}$ and $t_{CE_{\text{earliest}}}$ are to ensure that the waiting and completion time do not exceed the time spent in the call itself. This is only an issue if a clock violation is present, as all ranks are expected to synchronize and therefore have an overlap in the execution time of the call. In the regular case, each rank computes its waiting time in Equation (3.6) by comparing its own Collective Begin timestamp against that of the latest process, which is available in the callback data. The subsequent SYNC_COLL event triggers the pattern detection to check whether waiting time was computed and, if so, stores a *Wait at Barrier* instance under the call-path node of the MPI_Barrier region. The WAIT_BARRIER user event is subsequently triggered for the most-severe instance tracking and statistics collection. Completion time is recorded separately and triggers a BARRIER_COMPL callback for the same purpose. Finally, the syncpoint handler responds to SYNC_COLL by storing a synchronization point for the Collective End, including all ranks of the communicator in the syncranks map. For barriers and all-to-all operations, the completion time is not only computed for storage in the synchronization point but is also reported as a metric in the same way as the wait-state

results. Although completion time is not a wait state in the strict sense of the definition, it reflects asymmetries in when processes leave the operation. Since all ranks are expected to leave such collectives at roughly the same time, this remains an informative metric, even though the user has no direct means of reducing it.

In principle, the same detection algorithm applies to the other collective operation types. The communication handler triggers different callbacks depending on the operation type, but operations that induce the same class of wait state, as listed in Table 2, are handled identically. The main difference lies in which MPI operation is used to exchange the required data.

For *Late Broadcast* wait states, the analysis exits early if no payload was transferred in the original application, as no synchronization can have occurred in that case. To determine this, the root broadcasts its send byte count to all processes in the communicator; if it is zero, no process can have received any data—even in the case of `MPI_Scatterv`, where varying counts are permitted—and the analysis returns immediately. Otherwise, the root’s `Collective Begin` timestamp is distributed to all ranks via a blocking broadcast to enable waiting time calculation via:

$$WT(p) = \begin{cases} t_{CB}(p_{root}) - t_{CB}(p) & \text{if } t_{CB}(p_{root}) > t_{CB}(p) \\ 0 & \text{otherwise} \end{cases} \quad (3.8)$$

For simplicity, capping the waiting time to the duration of the call, which prevents clock-condition violations, is omitted here.

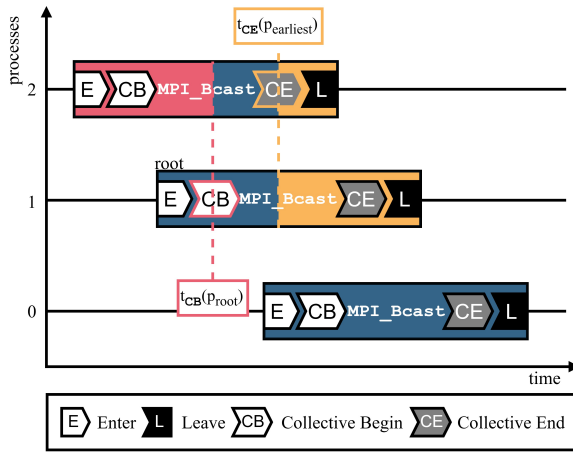


Figure 19: *Late Broadcast* wait state with root on process 1.

Figure 19 shows a *Late Broadcast* wait state on rank 2, marked as a solid red area. As the `Collective Begin` and `Enter` timestamps differ only negligibly in practice, the wait state is marked from the beginning of the call for simplicity. The relevant timestamp for wait-state calculation, $t_{CB}(p_{root})$, is indicated by the dashed red line. To compute the completion time, the earliest `Collective End` timestamp among the synchronizing receivers is required, marked by the orange dashed line, with the solid orange area indicating the resulting completion time. For both waiting and completion time, only processes that both entered the operation before the root and have a non-zero byte count are considered. This is necessary because operations such as `MPI_Scatterv` may distribute different payload sizes to individual ranks, and those that receive no data do not need to synchronize. Processes

not meeting this condition contribute a predefined maximum value to an MPI_Allreduce, while the remaining processes contribute their Collective End timestamp; the reduction returns the earliest qualifying timestamp. The root process is considered to have a synchronization point if at least one receiver entered the operation before it. For all synchronizing ranks, the completion time is then computed as

$$CT(p) = t_{CE}(p) - t_{CE}(p_{\text{earliest}}), \quad p_{\text{earliest}} = \arg \min_{\substack{p \in \text{comm} \\ \text{bytesRcvd}_{CE}(p) > 0}} t_{CE}(p) \quad (3.9)$$

The pattern detection records the waiting and completion times and triggers statistics collection. The syncpoint handler uses an MPI_Reduce to collect the synchronizing ranks onto the root: each non-root synchronization point sets its own bit in a local bitfield, which is reduced to the root via bitwise OR. The root then populates its syncranks from the resulting bitfield, while each non-root synchronization point stores only the root rank in its own syncranks. In the example in Figure 19, only ranks 1 and 2 are considered synchronizing.

To compute Early Reduce (see Figure 9a, page 14) wait states, the analysis process responsible for the root rank first broadcasts the root's Collective Begin timestamp to all ranks. Subsequently, an MPI_Allreduce operation is used to determine the latest Collective Begin timestamp among all non-root ranks that contribute a non-zero payload. The root and the latest sender are considered the only relevant synchronization partners, since the completion of the reduction depends on the arrival of the last contributing sender. A synchronization point between these two ranks exists only if the root entered the operation before the latest sender. The latest sender can verify this condition locally using root's Collective Begin timestamp, which was distributed via the preceding broadcast. To compute the completion time, the earliest Collective End timestamp among the synchronization partners is required. Although only the root and the latest sender contribute to this value, an MPI_Allreduce over the full communicator is used rather than a point-to-point exchange, to avoid interference with the point-to-point analysis messages. In case of a synchronization, the root process computes its waiting time as the difference between the latest sender's Collective Begin timestamp and its own. Both the root and the latest sender then compute their completion time, and the syncpoint handler records a synchronization point for each, with the root and the latest sender storing each other's rank in their respective syncranks.

For all-to-all operations, the analysis first checks for *full overlap*: whether the earliest Collective End timestamp across all processes is later than the latest Collective Begin timestamp, that is, whether every process was still inside the operation when the last process entered. If full overlap holds, all processes have synchronization points and the wait-state computation proceeds as for a barrier. Otherwise, the analysis falls back to

partial overlap: only receivers (non-zero receive payload) that entered before the latest sender (non-zero send payload) and the latest sender itself are treated as synchronization points. For all ranks that have a synchronization point stored in their callback data, the pattern detection computes the waiting time as the difference between the latest sender's `Collective Begin` timestamp and the rank's own, and the completion time as the difference between the rank's own `Collective End` timestamp and that of the earliest end. In the syncpoint handler, each synchronizing rank sets its own bit in a local bitfield, which is then reduced across all participating ranks via `MPI_Allreduce` with bitwise OR to assemble the complete set of synchronizing ranks on every process and store a synchronization point. `MPI_Alltoallv` and `MPI_Alltoallw` are excluded from the analysis, as these variants can be used to implement arbitrary communication patterns (e.g., pairwise message exchange), and the trace data does not provide sufficient information to reconstruct the actual communication pattern.

For `MPI_Scan` and `MPI_Exscan`, process p can only receive its result once all processes with rank less than p have contributed their data. The latest begin timestamp among all sending processes with rank less than p is determined via `MPI_Scan` with `MPI_MAXLOC`, which computes the prefix maximum. The *Early Scan* waiting time on rank p is then computed as the difference between the latest `Collective Begin` timestamp among ranks $0, \dots, p-1$ and p 's own. Completion time and synchronization points are not recorded for scan operations, meaning they are excluded from the subsequent delay and critical-path analyses.

For all collective operation types handled in the communication handler, the `Collective-Info` object is stored in a map keyed by the identifier of the `Collective Complete` event. This is to make the information available in the critical-path and delay analyses, which are conducted in a separate replay pass.

3.2.3 Handling of Multi-completions

Most-severe instance tracking requires both the backward and forward wait-state detection passes to determine which wait state to report when multiple wait states occur within the same call. Consider the example in Figure 20, where process 1 initiates two synchronous send operations and one receive, all completed in a single `MPI_Waitall` call. Process 3 sends a message to process 1, while processes 0 and 2 each receive a message from process 1. Communication events belonging to the same operation are colored identically. The solid red region marks the waiting time of the *Late Receiver* instance that should be reported, while the dashed lines indicate the waiting times associated with the other two operations. For wait states in multi-completion calls, the pattern is not confirmed immediately upon detection. Instead, a candidate is created, and the most-severe candidate seen so far in the current region is tracked. The most-severe candidate is only confirmed

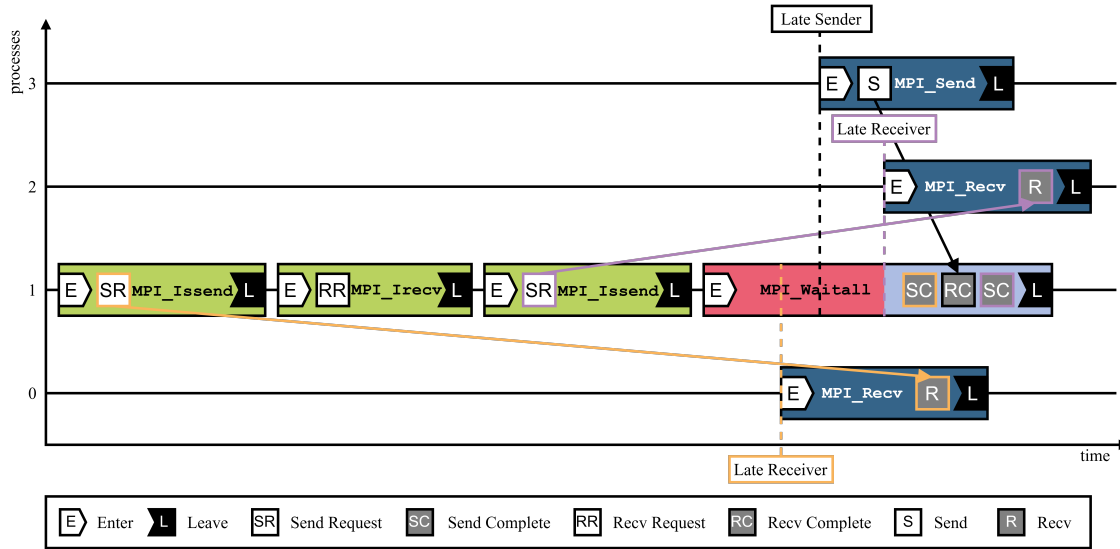


Figure 20: *Most-severe instance tracking in multi-completion regions.*

when the Leave event of the multi-completion region is processed in the forward pass. The corresponding user event to store the waiting time is then fired. For *Late Sender* instances this is straightforward since the forward pass processes events chronologically, and all candidates within a region are seen before its Leave event is reached.

Late Receiver detection is more involved. It happens in the backward pass, but confirmation must be deferred to the forward pass, where it can be compared against other wait states such as *Late Sender* that are only detectable in the forward direction. The backward pass therefore tracks the most-severe candidate per region. Since the trace is traversed back to front, Send Complete events are encountered before their corresponding Send Request events, but *Late Receiver* detection is triggered at the request event, where the analysis message carrying the receiver’s timestamp becomes available. A callback registered for Send Complete events checks whether the completion falls inside a multi-completion region. If so, the *Late Receiver* pattern creates an entry in a map keyed by the Enter event of that region and increments a send count. In Figure 20, the entry is created upon the purple Send Complete event on rank 1, and the send count is incremented to two when the orange Send Complete is encountered. When a Send Request event is reached whose completion falls inside a multi-completion region, the send count is decremented, and the waiting time is compared against the current maximum candidate. In Figure 20, the purple Send Request event is encountered first: a *Late Receiver* candidate holding the waiting time marked in red is stored as the current maximum, and the send count is decremented to one. When the orange Send Request event is subsequently encountered, the corresponding waiting time—indicated by the orange dashed line—is computed and compared against the current maximum. Since the purple *Late Receiver* candidate is more severe, it remains the maximum, and the send count is decremented to zero. Once the send count reaches zero, all Send Request events for that region have been processed, and

a `LATE_RECEIVER_CANDIDATE` user event is fired, storing the most-severe candidate for access during the subsequent forward pass. The *Late Receiver* candidates are collected in a double-ended queue and, after the backward pass completes, sorted in descending order by send completion enter timestamp to allow correct stack-like processing in the forward pass so that the next relevant candidate is always at the back during the forward pass.

In the forward pass, processing the `Recv Complete` event on rank 1 stores a *Late Sender* candidate as the current maximum. When the `Leave` event of the multi-completion region is encountered, the *Late Receiver* candidate from the backward pass is retrieved and compared against the *Late Sender* candidate. Since the *Late Receiver* waiting time is greater, it is confirmed, which fires the `LATE_RECEIVER` user event.

3.3 Critical Path and Cost Accounting

The delay and critical-path analysis pass traces previously identified wait states back to their root cause, quantifies the resulting cost, and computes a critical-path profile. Delay analysis [6, 28] attributes each wait state to the delay that caused it, while critical-path analysis [8] identifies the sequence of activities that determines the program’s overall execution time. Both analyses are carried out within the same replay pass, though they are triggered by distinct user events. The delay analysis is discussed first.

3.3.1 Delay Analysis

This section covers which data is required and how it is exchanged during the delay analysis [6, 28]. It provides a conceptual overview of how delay costs are computed, without going into the details of the actual calculation. For blocking collectives, the delay analysis currently handles the same operation types as the wait-state detection, excluding scan and exscan operations, for which the stored synchronization point information is incomplete.

It builds on the concept of synchronization intervals. A synchronization interval is the interval between two successive synchronization points involving the same processes. The delay of a call path within such an interval is the positive difference in the execution time spent in that call path between the participating processes over the interval, as only excess workload can contribute to delays. Intuitively, if the process that arrives late spent more time in a particular call path than the processes waiting for it, that call path is a candidate cause of the resulting wait state.

Two properties of the analysis determine its structure. First, delay costs must be attributed to the rank that causes the wait state, not the rank that observes it; therefore, the relevant data must be transferred to the causing rank. Second, a wait state can itself delay later communication and produce further, propagating wait states downstream. To account for this, the costs must be accumulated and traced back towards their origin. This is only

possible in a backward replay, where processing the trace from end to start allows costs identified at a later synchronization point to be carried back to the earlier delay that caused them. Whenever synchronization point information indicates a wait state identified during the previous replays, the algorithm determines the corresponding synchronization interval, identifies the delays and propagating wait states responsible for the waiting time, and computes the short-term and long-term delay costs.

The mechanism is illustrated using a blocking barrier exhibiting *Wait at Barrier* wait states, shown in Figure 21. All processes first execute a blocking barrier. Its waiting time is not relevant to the walkthrough and is therefore left unmarked. Its completion time is marked in orange, indicated by the orange dashed line. During wait-state detection, this completion time was computed and stored in the synchronization point information, with a synchronization point recorded on each of the three ranks. The processes then execute a routine *Foo*, after which process 2 sends a message to process 0, producing a *Late Sender* wait state and thus a synchronization between the two. The Enter and Leave events for the point-to-point message exchange are omitted for clarity. All ranks then run a routine *Bar* before synchronizing again in a second barrier, where processes 1 and 2 incur the waiting time shown by the solid red areas.

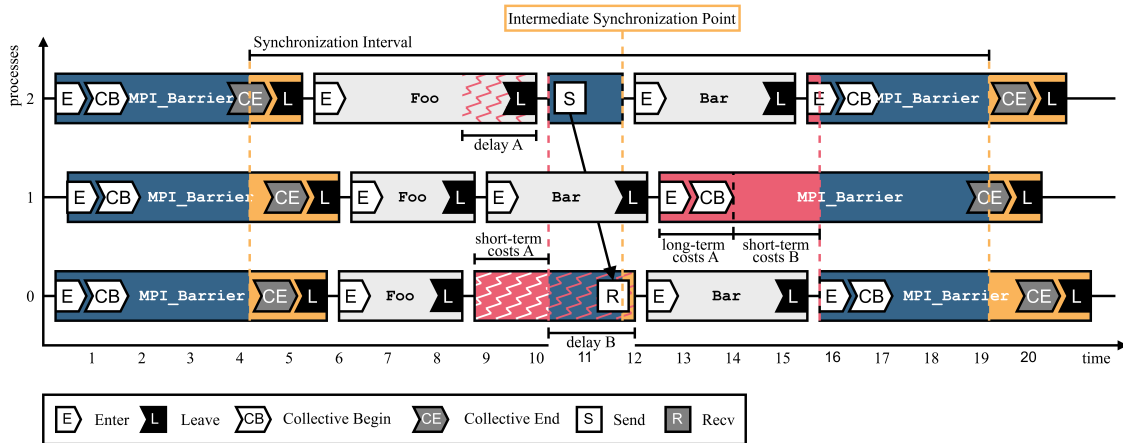


Figure 21: Delay analysis for a sequence of two blocking barriers and a *Late Sender* wait state between them.

During the backward replay, the Collective End event of the second barrier is the first encountered by each rank. The communication handler determines the collective type and fires the corresponding user event, in this case SYNC_COLL, and the delay pattern checks whether a synchronization point is stored for it. Since this is the case in the example, each rank determines its synchronization interval with the causing rank, bounded by the previous synchronization point it shares with that rank. Process 1's previous synchronization point is the Collective End event in the first barrier, while process 2's is the Send event, giving it a shorter interval than process 1. The latest arriving rank, here process 0, is treated differently from the others since it is the one causing the *Wait at Barrier* wait state. It

determines its last synchronization point with all of its synchronizing ranks. Process 0 last synchronized with all other ranks at the first barrier but has an additional intermediate synchronization point with process 2 at the Recv event. To ensure that all ranks compare the same interval boundaries, the completion time is included at the start of the interval and excluded at its end so that the boundaries align consistently across all processes.

Each rank then constructs a profile of the time spent per call path within its interval, excluding the communication operation at the end of the interval (here, the portion of the second MPI_Barrier marked in blue). The waiting time is annotated separately to be able to distinguish between delays and propagating wait states. Because process 0 has an intermediate synchronization point with process 2, it creates two profiles: one over the full interval for comparison with process 1, and a shorter one up to the intermediate synchronization point for comparison with process 2. The causing rank, process 0, first distributes the full-interval profile to all ranks via a broadcast, then corrects the data for ranks with an intermediate synchronization point, here process 2, by sending the shorter profile via a point-to-point message using a reserved tag value to not interfere with other point-to-point communication during the replay. This point-to-point data overwrites the previously broadcast values locally so that every process has received the correct interval profile for its own comparison, which can then be performed in parallel. Each waiting process compares, for every call path, the time received from the causing rank against the time it spent locally in the same call path, excluding waiting time. A positive difference is a delay contribution, corresponding to the time the causing rank spent more in that section of the program than the waiting process. A process that is not itself a synchronization point for the event contributes nothing.

In the interval bounded by the intermediate synchronization point and the Collective End event of the second barrier, the only difference in execution time between process 0 and process 2 is the completion time spent in the receive region on process 0. Process 2 therefore identifies its waiting time in the barrier as a direct wait state, caused by the orange-marked completion time in the receive. Process 1 compares the full-interval profile of process 0 against its own and finds that the only difference in execution time lies in the receive region. The portion not classified as a wait state is identified as delay (red zigzag), accounting for a fraction of the wait state on process 1, which constitutes that delay's short-term costs. The *Late Sender* waiting time within the receive itself is identified as a propagating wait state (white zigzag), and the remaining fraction of the wait state on process 1 contributes to the long-term costs of the delay causing that wait state; though this causing delay has not yet been identified. The wait state is therefore annotated with the amount of its long-term costs, to be resolved once the corresponding delay is found in an earlier interval. Both delay contributions are reduced onto the causing process, process 0, where the identified delay is stored.

A similar procedure is then applied one interval further back. When process 0 reaches the Recv event and process 2 the Send event, both again construct profiles, this time for the interval beginning at the Collective End event of the first barrier and ending at the intermediate synchronization point. Process 0 sends its profile, including the long-term costs already accumulated for the receive region, to process 2, following the inverted communication direction used during the backward replay, and process 2 computes the difference. Since the communication operation at the end of the interval is excluded from the comparison, the only difference lies in the execution time of routine Foo. The completion time in the MPI_Barrier does not constitute a delay, as the duration on rank 2 is shorter than on rank 0. Therefore, only the difference in Foo is identified as a delay, marked with red zigzag lines. It accounts for the short-term costs from the *Late Sender* wait state on process 0, while the corresponding fraction of the wait state on process 1 is attributed to its long-term costs.

3.3.2 Critical-path Analysis

The critical-path analysis [8] is conducted in the same backward replay as the delay analysis. It identifies the chain of activities that determines the program's total execution time. As introduced in Section 2.2.1, the critical path is the longest execution path without wait states.

The critical path is constructed incrementally starting from the end of the trace. The endpoint of the critical path is determined by the MPI rank that entered MPI_Finalize last. The critical path can only be on one process at a time; therefore, an ownership flag is set. From there, the critical path is followed backward through the trace. At each synchronization point, the analysis checks whether ownership needs to switch to another process. If the process currently holding the critical path incurred waiting time in the region associated with that synchronization point, the path transitions to whichever synchronizing process arrived latest. For collective operations, this is determined through a reduction with the rank of the latest Collective Begin timestamp as root, where each synchronizing rank contributes its ownership flag. The reduction still uses the original communicator, but ranks without a synchronization point do not contribute. For point-to-point events, the flag is instead sent to its peer process using point-to-point communication. Once a process gains ownership, the critical path continues backward through its local sequence of activities until the next synchronization point, where the same check is repeated. The result is a profile containing the total amount of time each call path spends on the critical path.

In the example used for the delay analysis (Figure 21, page 40), the critical path ends on process 0, since it is the last to arrive at the barrier and therefore the one causing the wait state on the other two ranks. Following the path backward, it reaches the Recv event,

where the *Late Sender* wait state forces a switch. The ownership moves to process 2 and stays there until the `Collective End` event of the first barrier, at which point it transitions back to process 1, since process 1 is the last to enter that barrier.

4 Analysis of Nonblocking Collective MPI Operations

Currently, Scalasca does not analyze nonblocking collective operations. Although the Score-P measurement system already records the event data described in Section 2.2.2, no callbacks are implemented in Scalasca that process these events. This chapter describes how the existing event model can be exploited to extend Scalasca’s analysis to nonblocking collectives. Section 4.1 discusses the wait-state analysis and presents the implemented approach. Section 4.2 then develops the concept for synchronization point recording, followed by Section 4.3, which outlines the theoretical concept for the delay and critical-path analyses.

4.1 Wait-state Analysis

This section discusses how the recorded events can be incorporated into Scalasca’s wait-state analysis with the goal of detecting the same classes of wait states in nonblocking collective communication as in the blocking case. Three approaches for exchanging the timestamps required for wait-state computation are examined in turn. The first reuses the blocking analysis directly, and the second mirrors nonblocking collectives with non-blocking analysis operations in a single replay pass; both are shown to fail. Their limitations motivate the two-pass approach that is adopted and presented in Section 4.1.1.

The most straightforward approach to analyzing nonblocking collectives would be to treat them the same way as their blocking counterparts, that is, triggering the same algorithms on `Collective Complete` events as are triggered on `Collective End` events. However, this does not work, as a simple counterexample illustrates. Consider two nonblocking barriers performed on different communicators, as depicted in Figure 22. The events marked in dark blue belong to a nonblocking barrier on `MPI_COMM_WORLD` and the ones marked in purple to one on a duplicate of it. This is a valid MPI program, and Scalasca must therefore be able to analyze its trace without deadlocking. Even though both communicators involve all ranks in the same order, they are distinct. The MPI standard only requires that collective initiation calls on the same communicator appear in the same order on all ranks, while completion calls may occur in arbitrary order. In the recorded trace, this translates to `Collective Request` events for a given communicator appearing in the same order on all ranks, while the corresponding `Collective Complete` events may appear in any order. This is particularly the case within multi-completion calls such as `MPI_Waitall`, where the order of the recorded complete events depends on the order of the requests in the passed array. In Figure 22, ranks 1 and 2 initiate the two barriers in the same order but complete them in different order, while rank 0 initiates them in the opposite order but completes them in the same order as rank 1. The current analysis for blocking barriers determines the latest begin timestamp of the operation across all participating ranks using

an MPI_Allreduce operation. For nonblocking barriers, the relevant timestamp $t_{CR}(p_{latest})$ is that of the latest Collective Request event. In the example, the first wait call on rank 1 incurs the waiting time shown in red. Computing this waiting time on rank 1 requires the Collective Request timestamp of the second MPI_Ibarrier on rank 0, indicated by the dark blue dashed line. Issuing a blocking MPI_Allreduce at the request event would cause ranks 1 and 2 to block on the MPI_COMM_WORLD allreduce, while rank 0 is blocked on the allreduce of the duplicate communicator, resulting in a deadlock. Issuing it at the Collective Complete event leads to the same outcome: ranks 0 and 1 would block on the MPI_COMM_WORLD allreduce, while rank 2 is blocked on that of the duplicate, again causing a deadlock. This rules out reusing the blocking analysis unchanged, motivating a second approach.

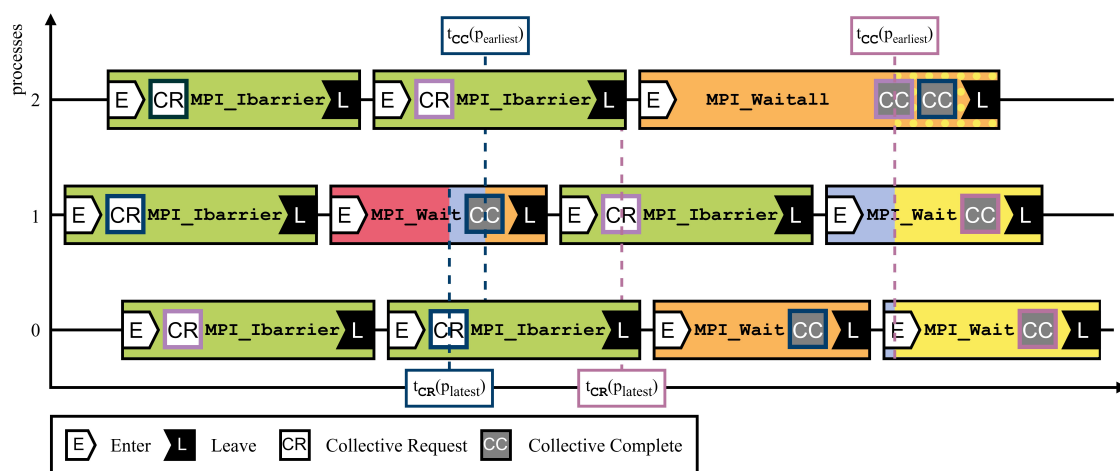


Figure 22: Two nonblocking barriers executed on distinct communicators with differing initiation and completion orders across ranks. Dark blue events belong to the barrier on MPI_COMM_WORLD, purple events to the barrier on a duplicate of it.

The natural alternative is to use nonblocking collectives for the analysis communication itself. This means initiating a nonblocking collective operation at the Collective Request event and completing it when the corresponding Collective Complete event is encountered. This avoids the deadlock in the example of Figure 22 but introduces another one. The MPI standard guarantees that a correct program must invoke collectives such that deadlock cannot occur regardless of whether the operations are synchronizing or not. This means that during the analysis replays, any number of blocking collectives can safely be issued at a Collective End event recorded for blocking collectives. For nonblocking collectives, however, this guarantee does not apply in the same way because Collective Complete events can occur inside test calls. Listing 4 (see Appendix, page 91) shows a valid MPI program in which rank 0 initiates a broadcast, tests for completion, sends a message, and only calls MPI_Wait if the test was unsuccessful. Rank 1 posts a receive, tests, initiates the broadcast, waits, and only waits for the receive to complete if the test was unsuccessful. If the test calls succeed, the recorded sequence of MPI-specific

events, shown in Figure 23, is indistinguishable from that of an incorrect program relying on the eager protocol. As the analysis is triggered solely by the recorded MPI-specific events, and this sequence of MPI-specific events can be recorded with a valid MPI program, it must therefore be handled correctly by Scalasca.

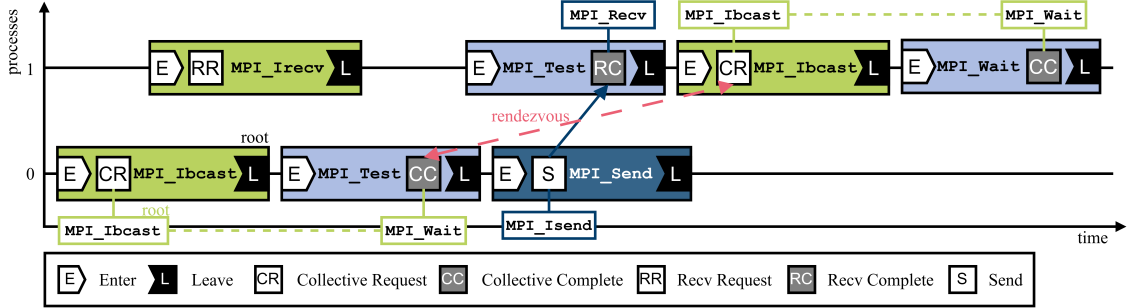


Figure 23: Recorded event sequence for the program in Listing 4 when test calls succeed.

For the example in Figure 23, it is considered what happens during the forward wait-state pass if nonblocking collectives are used for the analysis messages. The MPI operations issued during the analysis are indicated by the rectangles connected to the corresponding events at which they are issued. Green rectangles correspond to the proposed nonblocking collective operations, whereas dark blue rectangles represent the point-to-point messages exchanged in the analysis. Process 1 posts a blocking receive when processing the Recv Complete event, while process 0 initiates a nonblocking broadcast to distribute its Collective Request timestamp and waits for its completion via MPI_Wait when encountering the subsequent Collective Complete event. However, process 1 cannot initiate the broadcast until it has received the event data from process 0, which only becomes available when process 0 reaches its Send event. If the broadcast uses a rendezvous protocol, indicated by the dashed red line, a deadlock occurs, making the analysis replay an invalid MPI program. Even if one could rely on the eager protocol to avoid the deadlock, the analysis would still be restricted to mirroring the broadcast with a broadcast, since only an operation with the same synchronization semantics can safely replay the original. Using a nonblocking allreduce, as is done with a blocking allreduce when processing a Collective End event related to a blocking broadcast to determine the earliest end timestamp, would require all processes to synchronize and would likewise deadlock. Since this only becomes an issue when complete events occur inside test calls, one might ask why it is relevant for wait-state detection at all, as Section 2.2.1 stated that test calls are local and therefore cannot incur waiting time. However, the same operation may complete inside a wait call on other ranks, where waiting time can occur. The analysis communication must therefore be triggered on every participating rank. Whether a given rank can incur waiting time is only known locally, so a rank completing the operation in a test call cannot simply be excluded from the collective exchange. Using a test to complete the analysis operation at that point is equally unsafe, since the analysis replay has entirely different

timing from the original execution, so the fact that the original operation completed within a test call does not guarantee that the analysis operation will. To reliably trigger the subsequent wait-state detection callbacks, the required data must be guaranteed to be available. Mirroring nonblocking collectives with nonblocking analysis operations in a single replay is therefore not viable either.

Given these limitations, a single replay pass is insufficient to detect wait states in non-blocking collectives. Before outlining the two-pass solution, several intermediate options can be dismissed. The first concerns the choice of communication operation used to exchange the timestamps required for the waiting time calculation. Here, collective communication is the most straightforward choice. Point-to-point communication would only complicate matters, since the event data of a collective event carries no message envelope, raising the problem of how to match the analysis messages. The second observation is that the deadlock problem cannot be avoided simply by restricting *which* events a replay pass processes. A dedicated replay pass ignoring point-to-point events would rule out the broadcast example above, but this does not help, since the point-to-point communication could simply be replaced by a broadcast or reduction between the same two ranks, leading to the same situation. The problem is therefore inherent to how nonblocking collective analysis operations are initiated and completed, not to the presence of point-to-point events. This leaves the question of where the analysis operations may safely be initiated and completed. Collective operations must be initiated in the same order on all ranks of a communicator. Assuming the trace originates from a valid MPI program, this leaves only one option: the collective analysis operation must be initiated at the Collective Request event, since the order of these events is guaranteed to be consistent across ranks. Initiating at the complete event is not viable, because complete events may occur in arbitrary order and would potentially make the analysis itself an invalid MPI program. Completing the operation with a wait call at the Collective Complete event is not possible, since complete events can occur inside test calls. Whether these constraints can be reconciled depends on when the exchanged data is actually needed. To compute its local waiting time, each rank requires the timestamps listed in Table 2, depending on the collective operation type. In general, these timestamps do not need to be exchanged in the same replay pass in which they are processed. However, to incorporate nonblocking collective wait states into the most-severe instance tracking for multi-completion calls, the wait states must be detected during the forward wait-state pass so that they can be compared against *Late Sender* instances. Thus, the calculation is confined to the forward pass, while the timestamp exchange is free to be performed earlier. This is exactly what motivates a two-pass approach. By decoupling the timestamp exchange from the waiting time calculation in time and separating them into two distinct replay passes, the exchanged data only needs to be available in the second pass. This allows the completion of the analysis operations to be deferred to any later point in the first replay. At the latest, it may

be deferred to its finalization, where all pending operations can safely complete. The next section describes this two-pass approach, which is also the one currently implemented.

4.1.1 Forward Replay Pass Implementation

Since timestamp exchange is a prerequisite for wait-state detection, the existing prepare statistics pass, henceforth referred to as the fwp pass, is reused for this purpose. Until now, this pass has performed no communication during event processing, only at finalization. The communication handler is therefore extended with a callback for `Collective Request` events, which initiates the analysis operations. The details are described below. First, the infrastructure for tracking nonblocking collective analysis operations must be established. The existing mechanism for nonblocking point-to-point analysis sends is largely reused: the communication handler maintains a vector of pending send requests and a separate vector of send buffers, since buffers may only be freed once the corresponding send has completed. Upon completion, both the request and its associated buffer are removed from their respective vectors. For collective analysis operations, a separate vector holds the pending collective requests. To make the exchanged data available in the subsequent replay pass, the communication handler uses a map associating each `Collective Complete` event with its `CollectiveInfo` object. This is necessary because the `CollectiveInfo` object is otherwise only accessible within the callback data of the event that created it. This map is already used in the wait-state analysis of blocking collectives to make the `CollectiveInfo` objects for `Collective End` events accessible during the delay and critical-path analysis pass. The buffers of the nonblocking collective analysis operations directly reference fields of the `CollectiveInfo` object, so no additional buffer management is required. A progress callback registered for `Send [Request]`, `Recv [Complete]`, and `Collective End/Complete` events calls `MPI_Testsome` to complete any requests that are ready at that point. Once all events have been processed, the remaining collective requests are completed via busy-waiting. `MPI_Testsome` is repeatedly called until all entries in the collective request vector have been completed.

When a `Collective Request` event is encountered, the collective operation type is not directly available from the request event itself. Instead, the iterator established during preprocessing is used to look up the corresponding `Collective Complete` event, from which both the operation type and the communicator are retrieved. If the communicator contains only a single process, the event is skipped, since collectives on self-like communicators cannot synchronize with other ranks and require no analysis. Otherwise, a `CollectiveInfo` entry keyed by the identifier of the complete event is inserted into the map, with the `my` field initialized with the local rank in `MPI_COMM_WORLD` and the `Collective Request` timestamp. Depending on the collective operation type, different nonblocking collective operations are issued to exchange the timestamps required for

waiting time computation. Each issued request is added to the pending collective request vector and completed asynchronously by the progress callbacks.

For barrier and communicator creation operations (see Figure 8b, page 13), all processes that enter a wait call earlier than the latest rank initiated its collective operation incur waiting time. To calculate the waiting time

$$p_{\text{latest}} = \arg \max_{p \in \text{comm}} t_{\text{CR}}(p)$$

$$\text{WT}(p) = \begin{cases} t_{\text{CR}}(p_{\text{latest}}) - t_{\text{E_wait}}(p) & \text{if } t_{\text{CR}}(p_{\text{latest}}) > t_{\text{E_wait}}(p) \\ 0 & \text{otherwise} \end{cases} \quad (4.1)$$

each process needs $t_{\text{CR}}(p_{\text{latest}})$, which is the latest Collective Request timestamp across all participating ranks. This is determined via `MPI_Iallreduce` with `MPI_MAXLOC` and `MPI_IN_PLACE`, using the `latest` field of the `CollectiveInfo` object as the buffer. This field is initialized with `my`, which comprises a timestamp–rank pair of the Collective Request timestamp and global rank. A second `MPI_Iallreduce` with `MPI_MINLOC` determines the earliest complete timestamp using the `earliest_end` field, which is required for the completion time computation. Both requests are added to the pending collective request vector. This completion time does not correspond to the completion time used for synchronization point recording but is analogous to the completion metric reported in the blocking case: after the earliest completion has occurred, no rank is expected to spend significant time in the completion call. To compute the completion time stored in the synchronization point, the earliest completion timestamp among the ranks incurring waiting time, as well as that of the rank with the latest request timestamp, is required. Determining these depends on the result of the first `MPI_Iallreduce`, which is not yet available when the operations are initiated during the fwp replay pass. Synchronization point tracking is therefore deferred and discussed in Section 4.2.

Equation (4.1) assumes the absence of clock condition violations. Should such a violation occur, where $t_{\text{L_wait}}(p) < t_{\text{CR}}(p_{\text{latest}})$, the waiting time computation caps the timestamp by using $\min(t_{\text{L_wait}}(p), t_{\text{CR}}(p_{\text{latest}}))$ in place of $t_{\text{CR}}(p_{\text{latest}})$, ensuring the waiting time does not exceed the time spent in the call. For simplicity, the waiting time formulas presented in the remainder of this chapter omit the capping term for clock condition violations, though the implementation does check for them.

For one-to-all operation types that may induce *Late Broadcast* wait states (see Figure 5, page 11), such as BCAST and SCATTER[V], waiting time is defined as

$$WT(p) = \begin{cases} t_{CR}(p_{root}) - t_{E_{wait}}(p) & \text{if } t_{CR}(p_{root}) > t_{E_{wait}}(p) \wedge \text{bytesRcvd}_{CC}(p) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (4.2)$$

for each non-root rank. Therefore, all ranks require the root's Collective Request timestamp, which is distributed via a nonblocking broadcast using the root field of the CollectiveInfo object as the buffer. Early exit in case all ranks have zero payload is not yet implemented, but the mechanism is outlined in the following. Since MPI_Iscatterv may assign zero payload to some ranks even when the root sends data, the decision of whether to continue analysis cannot be made locally. The root's sent byte count must therefore be distributed to all ranks. As no dedicated buffer is available for this, the CollectiveInfo object can be repurposed by storing the byte count in place of the timestamp field.

Operations types that may induce *Early Reduce* wait states (see Figure 9b, page 14) require the analysis process responsible for the root rank to obtain the latest Collective Request timestamp among all sending ranks with a non-zero payload. In the blocking variant, three collectives are executed when handling all-to-one related Collective End events. An allreduce is used to find the latest begin among all senders, a broadcast is used to distribute the root's begin timestamp to all ranks so that each sender can determine locally whether it is a synchronization point, and a second allreduce to be able to compute the completion time. Since the completion time depends on results of the first allreduce and the root's begin timestamp, it is deferred to Section 4.2. In order to calculate the waiting time on the root process, a single nonblocking reduce over the Collective Request timestamps with MPI_MAXLOC is sufficient. Senders with zero payload contribute a predefined minimum value, ensuring that only ranks with payload influence the result, and the latest timestamp is written into the latest field of the CollectiveInfo object on root. To allow early exit for zero-payload ranks in the subsequent pass, the root process could broadcast its received byte count via the CollectiveInfo object, as described for the one-to-all case. Since this information is only available once the operation has completed, the other reduce operation must still be issued regardless.

For prefix reduction operations (see Figure 10, page 14), the waiting time depends on the latest Collective Request timestamp among all lower-ranked processes. To obtain this value, an MPI_Iscan with MPI_MAXLOC is issued, where only ranks with a non-zero send payload contribute their request timestamp; the other ranks contribute a predefined minimum value.

For all-to-all operations, the blocking analysis distinguishes between full and partial overlap. To enable the same distinction, the timestamps needed to determine the overlap case must be exchanged here. Three nonblocking allreduces are issued: the first uses the `latest` field to determine the latest Collective Request timestamp across all ranks via `MPI_MAXLOC`; the second uses the `earliest_end` field to determine the earliest Collective Complete timestamp via `MPI_MINLOC`. These two values together determine whether full overlap holds. For partial overlap, the latest request timestamp among ranks with a non-zero send payload is additionally needed. Since the `root` field of the `CollectiveInfo` object is not used for all-to-all operations, it is repurposed to hold this value via a third `MPI_Iallreduce` with `MPI_MAXLOC`, where ranks with zero send payload contribute a predefined minimum value.

In the subsequent forward wait-state detection pass, the actual wait-state detection and waiting time computation take place, using the timestamps stored in the `CollectiveInfo` map during the preceding fwp pass. A callback registered for Collective Complete events mirrors the role of the Collective End callback in the blocking analysis. As in the fwp pass, events on single-process communicators are skipped. Otherwise, the collective operation type is retrieved, and depending on the type, corresponding user-defined events are fired. Operations that induce the same class of wait state are handled together, following the same grouping as in the blocking case.

The analysis is illustrated using the example in Figure 22 (see page 46). During the fwp pass, two `MPI_Iallreduce` operations are initiated at each Collective Request event to determine the latest Collective Request timestamp and the earliest Collective Complete timestamp. The resulting timestamps are indicated by the correspondingly colored dashed lines in the figure. For the barrier on `MPI_COMM_WORLD`, marked by the dark-blue-framed events, rank 0 has the latest request timestamp $t_{CR}(p_{latest})$, and rank 1 has the earliest complete timestamp $t_{CC}(p_{earliest})$. For the barrier on the duplicate communicator, marked by the purple-framed events, rank 1 has the latest request timestamp and rank 2 has the earliest complete timestamp. After the fwp pass, these values are available on all ranks through the `CollectiveInfo` map, keyed by the identifier of the corresponding Collective Complete event. When an analysis process encounters a Collective Complete event during the forward wait-state detection pass, the communication handler fires the `PRE_SYNC_COLL_COMP` and `SYNC_COLL` user events. The `SYNC_COLL` event is also fired during the analysis of blocking collectives to store the computed waiting time. Since this logic is not specific to blocking operations, the same code can be reused for the nonblocking analysis. The subsequent analysis proceeds similarly to the blocking case. A callback in the communication handler registered for `PRE_SYNC_COLL_COMP` retrieves the associated `CollectiveInfo` object from the map and stores it in the callback data, making it available to subsequent callbacks of other handlers. It also stores the relevant events, including the Enter and Leave events of the completion region and the Collective

Request event in the callback data. In general, a wait state occurs when a process reaches a synchronization point later than others. Synchronization points are recorded on the rank that incurs the waiting time as well as on the rank that causes it. For blocking barriers, all participating ranks must overlap inside the call, so every rank except the one of the latest beginning incurs waiting time, even if only by a small margin. Consequently, a synchronization point is tracked on all ranks. For nonblocking barriers, this is not necessarily the case. Complete events can occur in test calls, which are local and cannot incur waiting time, and even blocking completion calls issued after the latest collective initiation do not inherit waiting time. Synchronization points should therefore only be recorded on the rank with the latest request timestamp and on those ranks that actually incur wait time. The synchronization point flag in the callback data determines which ranks participate in the waiting time calculation. For a nonblocking barrier, the flag must only be set on ranks satisfying $t_{\text{E}_{\text{wait}}}(p) < t_{\text{CR}}(p_{\text{latest}})$, that is, those that entered the completion call before the latest Collective Request was issued. For the MPI_COMM_WORLD barrier in the example, this applies to rank 1, whose wait call is entered before rank 0's request event.

A callback registered in the pattern detection validates the clock condition. If the latest request timestamp exceeds the timestamp stored in the `earliest_end` field of the `CollectiveInfo` object, a `COLL_CCV` user event is fired to accumulate the clock condition violations. For all ranks with the synchronization point flag set, the waiting time is computed via Equation (4.1). Both timestamps required for the calculation are available through the callback data: $t_{\text{CR}}(p_{\text{latest}})$ via the `latest` field of the `CollectiveInfo` object and $t_{\text{E}_{\text{wait}}}(p)$ via the `Enter` event of the completion region. If this region corresponds to a test call rather than a wait call, no wait time is computed for that rank. In the example illustrated in Figure 22 (see page 46), the computed waiting time on rank 1 is marked in red. For the barrier on the duplicate communicator, no waiting time is computed since no rank has the synchronization point flag set, because the latest initiation, that on rank 1, occurs before the `Enter` event of any of the completion calls.

The completion time metric is computed on all ranks via

$$\text{CT}(p) = t_{\text{L}_{\text{wait}}}(p) - t_{\text{CC}}(p_{\text{earliest}}), \quad p_{\text{earliest}} = \arg \min_{p \in \text{comm}} t_{\text{CC}}(p) \quad (4.3)$$

For the barrier on MPI_COMM_WORLD, the calculated completion time is marked in orange. It covers the entire execution of the MPI_Waitall region on rank 2, the second wait call on rank 0, and a small fraction on rank 1 between its completion and the `Leave` event; though in practice this fraction is negligible since the two timestamps differ only marginally. For the barrier on the duplicate communicator, the completion time is marked in yellow. It covers almost the entire second wait region on rank 0, a fraction of the second wait on rank 1, and the yellow dotted portion of the MPI_Waitall region on rank 2. To avoid accounting for a fraction of the MPI_Waitall region twice, most-severe instance tracking

must also be enabled for nonblocking collectives. This is described in the following section.

The callback in the communication handler listening for SYNC_COLL proceeds analogously to the blocking case, storing the waiting time and completion time as metrics under the call-path node of the completion region. The syncpoint handler is not triggered at this point, since determining the synchronizing ranks and the completion time to be stored with the synchronization point requires additional collective communication. Statistics collection is likewise deferred, as it also relies on additional collective communication. Neither can be performed during a single pass for the reasons discussed earlier.

With the timestamps already available in the `CollectiveInfo` objects, computing the wait states for the remaining collective types is relatively straightforward. When the communication handler fires the pre-event for a given collective type, the corresponding `CollectiveInfo` object is retrieved from the map using the identifier of the `Collective Complete` event currently being processed. The synchronization point flag is then set for those ranks that need to compute waiting time. Unlike in the blocking case, the flag is not set for all synchronizing ranks. In particular, not for the rank causing the wait state on the others, since determining this would require further communication based on results that have only become available in the current replay pass. For all-to-all operations, the completion time is computed analogously to the barrier case for reporting as a metric, but is not used for synchronization point tracking.

Handling of Multi-completions

Completion events for nonblocking collectives can also occur inside multi-completion regions such as `MPI_Waitall` or `MPI_Waitsome`. To ensure that only the single most-severe wait state within such a region is reported, the existing most-severe instance infrastructure is extended. A dedicated collective wait state candidate is introduced, following the same candidate interface used for point-to-point wait state candidates. The actual storage of the wait state instance is deferred from the collective-specific user events, such as `SYNC_COLL` or `COLL_12N`, to a later point. When one of these events fires, it is first checked whether the completion occurs inside a multi-wait region. If not, the wait state is reported directly as before. If it does, a candidate event is fired instead, deferring the decision of which instance to report until the `Leave` event of the region is processed.

To illustrate how most-severe instance tracking extends to collective operations, consider the example in Figure 24. Message arrows are omitted for clarity, and initiation calls sharing the same color correspond to one another. Process 1 initiates a receive, a synchronous send, and a broadcast, and then completes all three operations within a single `MPI_Waitall` call. The receive cannot complete until process 2 initiates its send, resulting in a *Late Sender* wait state (green, dotted). The synchronous send can only complete once process 3 initiates

which is the *Late Sender* instance. Since the *Late Broadcast* waiting time is greater, it becomes the new maximum candidate. When the Leave event of the MPI_Waitall region is encountered on rank 1, the current maximum candidate is compared against the (maximum) *Late Receiver* candidate stored for the region during the preceding backward wait-state analysis pass (see Section 3.2.3). Although a *Late Receiver* candidate is present, the *Late Broadcast* is more severe and is therefore confirmed. Confirmation writes the candidate's data back into the callback data and fires the user event recorded under the dedicated role, which is the LATE_BCAST event. The wait state is then stored under the call-path node of the completion region.

4.2 Synchronization Points

As discussed in Section 3.3.1, the delay analysis relies on previously recorded synchronization points. These must store not only the synchronizing ranks but also the completion time among them, so that the same synchronization interval can be compared consistently across ranks. This section therefore covers how the implementation introduced in Section 4.1.1 can be extended to track these synchronization points. As in the blocking case, scan and exscan operations are omitted due to infrastructure limitations in storing the synchronization points for these operations. While the wait-state analysis described in the preceding section has been implemented, the synchronization point recording presented here, as well as the delay and critical-path analyses discussed in Section 4.3, remain conceptual.

In general, a synchronization point is recorded for a specific event on the ranks that incur waiting time, and, if any rank incurs waiting time, also on the rank causing it. From the perspective of the delay analysis, what matters is that every process incurring waiting time stores the rank of the process that caused it, that is, the rank whose request timestamp was used in the waiting time computation. Conversely, the causing process must store the ranks of all processes that incurred waiting time, as these are the ranks that will compare their synchronization interval profiles against its own in the delay analysis.

The general approach for exchanging the information required to record synchronization points is to initiate nonblocking collectives during a replay pass that precedes the delay analysis, so that the results are available once that pass completes. However, this exchange cannot be initiated during the forward wait-state detection pass itself, since it requires knowledge of whether a given event is associated with waiting time. During the forward wait-state detection pass, the waiting time computation happens when the Collective Complete event is processed. However, the nonblocking collective used to transfer the synchronization point information would need to be initiated already at the corresponding request event, at which point it is not yet known whether waiting time has been incurred. Since the wait-state computation itself requires no communication, triggering the computation at the request event would in principle be possible. Unfortunately, this does not work

for wait states in multi-completion regions, as these are only confirmed once the Leave event of the multi-completion region is processed. Instead, the backward syncpoint exchange pass following the wait-state detection pass can be used to initiate the nonblocking collectives. Although this is a backward pass, the initiation of a nonblocking collective at the point of encountering a Collective Request event should still be possible, since the condition that these operations be initiated in the same order per communicator continues to hold. The order in which the requests are now encountered is reversed relative to the original trace, but since this reversal applies across all processes, the relative order between processes remains unchanged. Once a wait state for a nonblocking collective is stored during the wait-state detection, a flag in the CollectiveInfo object must be set to make this information available during the backward syncpoint exchange pass.

For operations that indicate *Wait at Barrier*, *Wait at Communicator Create*, or *Wait at $N \times N$ wait states*, the causing rank is that of the latest Collective Request timestamp or, for all-to-all operations with partial overlap, that of the latest sender's request timestamp. For all ranks incurring waiting time, the synchronization point must be recorded for the Collective Complete event, whereas the causing rank stores it at the Collective Request event since these events are located in the regions whose execution times overlap. The completion time is computed relative to an earliest reference timestamp t_{ref} , defined as the minimum of the Collective Complete timestamps of all ranks incurring waiting time and the Leave timestamp of the causing rank's initiation call:

$$p_{\text{earliest}} = \arg \min_{\substack{p \in \text{comm} \setminus \{p_{\text{latest}}\} \\ \text{WT}(p) > 0}} t_{\text{CC}}(p),$$

$$t_{\text{ref}} = \min(t_{\text{CC}}(p_{\text{earliest}}), t_{\text{L}_{\text{initiation}}}(p_{\text{latest}})) \quad (4.4)$$

For each synchronizing rank its completion time is then defined as the difference between its own Leave timestamp and this reference via

$$\text{CT}(p) = \begin{cases} t_{\text{L}_{\text{initiation}}}(p) - t_{\text{ref}} & \text{if } p = p_{\text{latest}} \\ t_{\text{L}_{\text{wait}}}(p) - t_{\text{ref}} & \text{if } p \in \text{comm} \setminus \{p_{\text{latest}}\} \wedge \text{WT}(p) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (4.5)$$

When a Collective Request event is processed during the backward syncpoint exchange replay, it must be checked whether the flag indicating waiting time is set in the CollectiveInfo object associated with the corresponding complete event. Two non-blocking collectives can then be issued. The first is an MPI_Iallreduce with MPI_MINLOC and MPI_IN_PLACE to determine the reference timestamp t_{ref} required for completion time calculation, using the earliest_end field of the CollectiveInfo object as the buffer. Ranks other than the causing rank that have not set the waiting time flag would contribute

a predefined maximum value, ensuring they do not influence the result. Additionally, the causing rank needs to know which ranks actually synchronized with it. For this, a nonblocking reduce can be issued in which each qualifying rank sets its own bit in a bitfield that is collected on the latest rank via a bitwise-OR operation. This part of the exchange could be handled by the syncpoint handler, which would then need to maintain its own tracking structure for the pending operation. The pending collective requests would be handled analogously to those in the fwp pass: progress callbacks would complete requests that are ready during event processing, and any remaining requests would be completed via busy-waiting once the replay pass finishes.

By the time the delay and critical-path analysis replay pass is performed, both operations would be guaranteed to have been completed. However, since this pass traverses the trace backward and looks up previously recorded synchronization points, the necessary information would already be available but not yet stored into synchronization points. The recording of the synchronization points must therefore happen in another replay pass that precedes the delay and critical-path analyses. For this purpose, the forward syncpoint exchange pass could be employed. For `Collective Complete` events, this pass would only perform local event processing without communication, since the required information was exchanged in the preceding backward syncpoint exchange pass and is now available through the `CollectiveInfo` map. The primary purpose of this pass—propagating *Late Receiver* synchronization points to the receiver side via point-to-point communication—remains unaffected by the additional callbacks for nonblocking collectives.

When a `Collective Complete` event is encountered, all ranks retrieve the associated `CollectiveInfo` object from the map into their callback data. A callback in the synchronization point handler, triggered for the latest rank and all ranks with the waiting time flag, would then handle the actual synchronization point tracking. For the causing rank, the bitfield obtained from the `MPI_Ireduce` is first checked to determine whether any rank synchronized with it. If so, it computes its completion time and stores a synchronization point at its request event, containing the synchronizing ranks obtained from the bitfield. Every other rank with the waiting time flag set likewise computes its completion time and stores a synchronization point at the `Collective Complete` event, containing the causing rank.

In the example depicted in Figure 25, the described approach stores synchronization points at the events marked in yellow. Only ranks 0, 1, and 3 share a synchronization point, since the barrier on process 2 completes without causing or incurring waiting time. The waiting time on processes 0 and 3 is caused by the delayed request on process 1. Process 1 therefore stores a synchronization point at its `Collective Request` event, containing the synchronizing ranks 0 and 3, and computes the completion time marked in orange. Rank 3 records a synchronization point at its `Collective Complete` event, containing the rank

of the latest request, rank 1, and computes the orange-marked completion time; process 0 does the same.

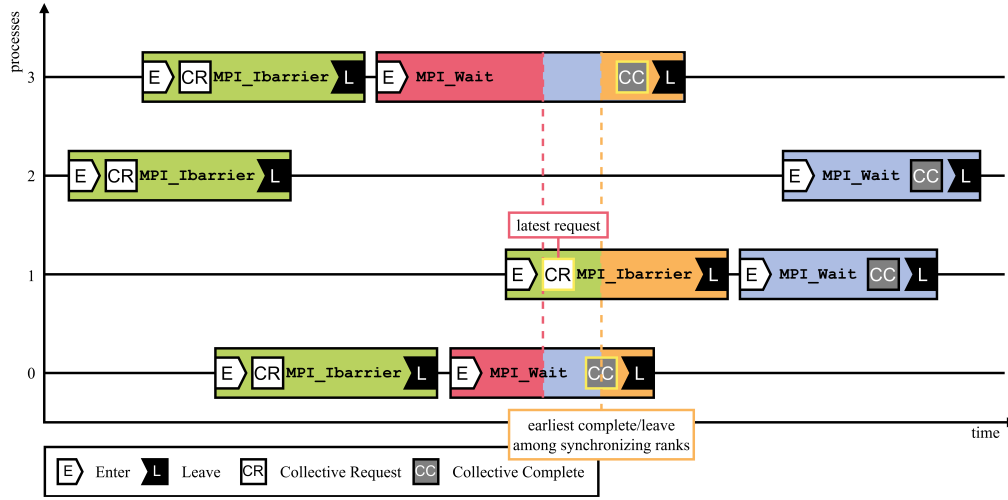


Figure 25: Synchronization points in a nonblocking barrier.

For collective operation types that correspond to one-to-all operations, the same method as described above can be applied. Here, the root rank takes over the role of the causing rank. For all-to-one operation types, only the root rank and the rank of the latest sender synchronize. Because this only involves two processes, point-to-point messages are sufficient to exchange the information required for the completion time calculation. These messages have to use a dedicated tag value to prohibit interference with the messages of the *Late Sender* synchronization point exchange. When a `Collective Request` event for a one-to-all operation is encountered during the backward syncpoint exchange replay pass, the associated `CollectiveInfo` object is checked to determine whether the current rank corresponds to the latest sender. If so, the rank sends its own `Leave` timestamp of the initiation region to the root and initiates a nonblocking receive to obtain the root's completion timestamp; completion of this receive must be handled by the pending operation tracking. Since the latest sender does not yet know whether the root rank incurred waiting time, the root only sends its own `Collective Complete` timestamp if the waiting time flag is set. If the flag is not set, it instead sends a predefined value indicating that no synchronization occurred. The root rank also initiates a nonblocking receive to obtain the message from the latest sender. During the next forward syncpoint exchange replay, a synchronization point is recorded only if the root sent an actual timestamp rather than the predefined sentinel value. If so, the completion time is computed, and a synchronization point is stored on the root at the `Collective Complete` event, containing the rank of the latest sender. The latest sender, in turn, records a synchronization point at its request event, containing the root's rank.

4.3 Delay and Critical-path Analyses

The previous section discussed how synchronization points for nonblocking collectives can be tracked. With these synchronization points in place, this section addresses the delay and critical-path analyses, which rely on them. In principle, the same algorithms as for blocking collectives can be applied to both the delay and critical-path analyses for nonblocking collectives. What differs is how the required information, namely the synchronization interval data and the critical-path inheritance flag, is exchanged among the replaying processes. This section first discusses general limitations of mirroring nonblocking collective operations in a backward replay before outlining an approach of how the required data can be transferred.

As already discussed in Section 4.1, nonblocking collectives must be initiated in the same order per communicator on all participating ranks. Consequently, the only point at which a collective analysis operation can be initiated during the backward replay is when the corresponding request event is processed, since the order of complete events may be arbitrary. However, it is unclear when such an initiated operation may safely be completed. The earliest point at which completion is safe is once the latest rank has initiated the operation, which, in a backward replay, corresponds to the rank with the earliest request event in time. This event can lie at any point in the preceding trace. The data required to proceed with the analysis, though, must already be available when the synchronization point is processed, which can occur at a `Collective Complete` event that, during the backward traversal, is encountered before its corresponding request event. A two-pass approach, as used for wait-state detection, is not viable here either, since the exchanged information must be processed within the same replay pass in which it is received. For instance, whether a given rank participating in the collective holds the critical path that must be handed off to the causing rank is not known in advance of the replay. Furthermore, the long-term delay costs must be propagated backward into earlier intervals, which requires immediate processing directly after the data is received. For these reasons, mirroring nonblocking collectives directly with nonblocking collectives is not viable for this analysis.

Figure 26 shows a timeline diagram of such a situation. All processes execute a non-blocking barrier; ranks 1 and 2 wait for its completion before rank 0 initiates the operation. Therefore, waiting time is incurred on the two ranks, and the events marked in yellow have a synchronization point. Rank 1 issues a blocking broadcast, in which it serves as root, before calling `MPI_Wait` to complete the barrier. This broadcast uses the eager protocol, since it completes before rank 0, which also participates in the broadcast, has entered it. This nonetheless represents the event sequence of a valid MPI program. If the broadcast instead used the rendezvous protocol, it would block until the broadcast call on rank 0 was reached, causing the broadcasts' execution times to overlap and introducing a

synchronization point there. In that case, rank 1 would no longer incur a *Wait at Barrier* wait state and would therefore have no synchronization point for the barrier.

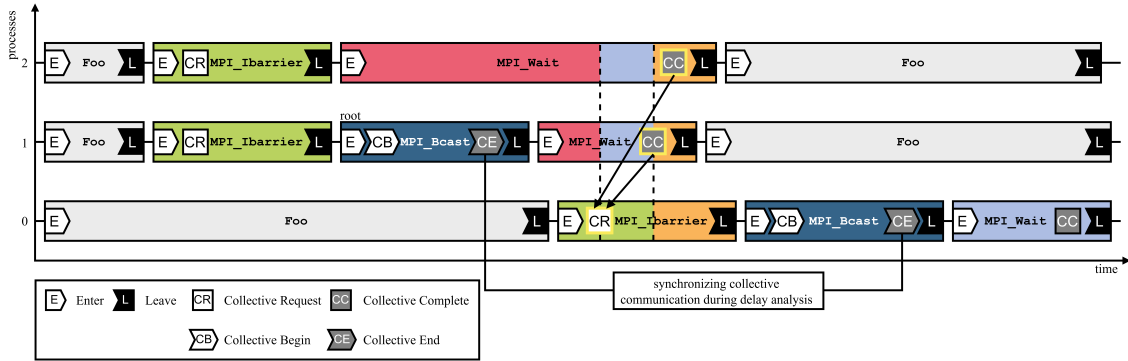


Figure 26: Data exchange at a synchronization point for a nonblocking barrier. A blocking broadcast is executed over a synchronization point. Communication back to the causing rank is not possible, as it would deadlock with blocking collective calls.

Even with blocking collectives being replayed regardless, some communication remains possible. The ranks incurring *Wait at Barrier* waiting time, marked in red, can send their synchronization interval profile to the causing rank via point-to-point messages using a nonblocking send. The causing rank, however, cannot communicate back to the other synchronizing ranks, since doing so would require a blocking receive there, which would deadlock with the blocking collective calls issued at the Collective End events. That is, rank 1 would sit in the receive call issued upon the Collective Complete event, while rank 0 would never reach the send triggered at the corresponding Collective Request, since it is itself stuck inside its blocking collective call, executed at the Collective End event, waiting for rank 1 to participate in the collective operation it can never reach.

This one-directional exchange of the synchronization interval profile, illustrated by the arrows in Figure 26, is not specific to the barrier example; it generalizes to an approach for handling nonblocking collective events in the delay analysis. The synchronization interval profile needed for comparison is sent from the waiting ranks to the causing rank using a nonblocking point-to-point send operation, while no message is sent in the reverse direction. This should in principle be deadlock-free, as only the causing rank has to perform a blocking operation, whereas the other ranks can proceed once the nonblocking message is initiated. For these messages, an envelope must still be constructed. The original communicator can be reused, and the destination or source rank can be retrieved from the synchronizing ranks stored together with the synchronization point. To label the message, a dedicated reserved tag has to be used. Since messages are exchanged only at the synchronizing event, no mismatch with other point-to-point analysis messages that use the same artificial tag may happen, as the messages have to occur in order per synchronizing rank groups.

This approach, however, limits the delay analysis with respect to parallelization. Since a single delayed request can potentially cause waiting time in many completion regions, the delay computation for blocking collectives is performed in parallel on all waiting ranks. The ranks incurring waiting time obtain the relevant interval from the delayed rank, compare it against their own, and send the result back to it. This is necessary because only the delayed rank can store the results, since it is where the delay, or the propagating wait state, actually occurred. As communication from the delayed rank to the waiting ranks is not possible for nonblocking collective events, the delay analysis must instead be performed sequentially on the delayed rank.

To overcome this restriction, the existing delay analysis for blocking collectives must itself be adapted so that execution can be decomposed into disjoint synchronization intervals, with communication confined to the boundary at which each interval ends; that is, only ranks that actually have a synchronization point would communicate. To still reuse the same blocking collective operations for data transfer described in Section 3.3.1, a subcommunicator containing only the synchronizing ranks would have to be defined. Currently, for synchronization points in a broadcast, for instance, only the root has the full bitfield of synchronizing ranks stored, while the other ranks only know the root. Making the bitfield available to all ranks can be easily incorporated into the forward wait-state detection replay. The construction of the subcommunicators would be the greater challenge, since each MPI implementation imposes an upper limit on the number of communicators that can be defined. To avoid exceeding this limit, communicators could be created on the fly during the replay pass and destroyed once no longer needed. However, since communicator creation itself requires synchronization across the parent communicator, this would not help confine communication to the ranks having a synchronization point. Constructing the subcommunicators upfront, in a preceding forward replay, would in principle be possible, but could be problematic with respect to the upper limit, since in theory each instance of a blocking collective could require a separate subcommunicator, as the set of synchronizing ranks can differ between instances.

Another option would be to use point-to-point communication for this analysis as well. Here, the same approach as for nonblocking collectives could be applied, exchanging point-to-point messages only between synchronizing ranks. Since this restricts communication to the end of synchronization intervals, communication between the synchronizing processes should be deadlock-free at the time of processing the synchronizing event. Consequently, an arbitrary number of blocking point-to-point operations can be exchanged in either direction, enabling parallel delay computation even for nonblocking collectives. An additional benefit of this approach is that blocking and nonblocking collectives could then be handled identically in the delay analysis.

Unlike the delay analysis, the critical-path analysis does not perform any parallel computation, so serializing it on the causing rank is less critical. That rank only needs to compare

the received flags locally to decide whether ownership must be transferred. The collective reduction used in the blocking case can therefore be replaced by individual point-to-point messages: when processing a `Collective Complete` event with a synchronization point, each waiting rank sends a flag indicating whether it holds ownership of the critical path to the causing rank. The causing rank receives these flags individually from all ranks stored in its `syncranks` bitfield when processing the corresponding `Collective Request` event. It then determines ownership transfer by checking whether any of the received flags indicates that a waiting rank held the critical path. To avoid mismatching these messages with those used for the delay data exchange, a separate reserved tag must be used to label the critical-path analysis messages.

5 Experimental Evaluation

The implemented wait-state analysis is evaluated both in terms of correctness and practical benefit. The first section covers the functional correctness of the implementation, followed by a case study that demonstrates the benefit of the extended functionality in practice. All Scalasca analyses presented in this chapter were performed with the prototype implementation¹, which currently supports only wait-state detection; delay and critical-path analysis for nonblocking collectives are not yet implemented, and the corresponding replay passes were therefore not executed.

5.1 Functional Correctness Verification

Functional correctness of the implemented wait-state analysis was tested on an internal test platform with the following configuration: two Intel Xeon X5660 six-core processors (Westmere-EP, 2.80 GHz, 2-way SMT), 24 GB of memory (DDR3, 1333 MHz), a QLogic InfiniBand 4x SDR adapter (10 Gbit/s), a Voltaire InfiniBand 4x DDR switch (20 Gbit/s, 24-port). The software stack included MPICH 4.2.2, GCC 14.2.0, and Score-P 9.4 [29]. The tests build on the existing Scalasca test code infrastructure², which already provides test cases for blocking collectives. Since replacing a blocking collective with its non-blocking counterpart immediately followed by a wait is expected to produce the same wait states, these existing test cases serve as a baseline correctness check for the nonblocking variants. Additional test cases were added covering scenarios specific to nonblocking collectives. This includes test cases in which only a subset of processes overlaps their completion with the latest initiation, as well as cases that introduce computation between initiation and completion. The Scalasca test code infrastructure provides several predefined communicators, along with utility functions that simulate computation for a given duration, and handles MPI initialization, finalization, and communicator creation. All test cases were executed using 8 processes on MPI_COMM_WORLD, as well as on odd/even and upper/lower split communicators. The following discussion focuses on MPI_COMM_WORLD for clarity. A blocking barrier is executed between test functions to ensure all processes are aligned at the start of each test case; any waiting time caused by processes leaving the previous test function at different times is attributed to this barrier and is not considered part of the test case itself.

For each wait-state class, the following paragraphs present representative test cases. The *Wait at Barrier* class is discussed first and in somewhat greater detail than the remaining classes. For each of the other wait-state classes, one representative test case is discussed, focusing on scenarios specific to nonblocking collectives. In all single wait-state cases,

¹<https://gitlab.jsc.fz-juelich.de/perftools/scalasca/-/tree/145-handle-nonblocking-collectives> [only accessible for authorized accounts]

²<https://gitlab.jsc.fz-juelich.de/perftools/tests/scalasca/testcodes> [only accessible for authorized accounts]

the current Scalasca main branch is unable to detect any wait state, since no callbacks are implemented for nonblocking collective events. The wait-state analysis implementation described in this thesis correctly identifies the expected wait states in all cases. Section 5.1.6 then covers most-severe instance tracking in multi-completion calls, where the main branch detects only the more severe of the point-to-point candidates, missing the collective wait state.

Tests confirming the absence of wait states and tests analogous to the blocking variants were also conducted for all wait-state classes, and across the different MPI calls that belong to the same class and are therefore handled identically by the algorithm. For example, `MPI_Ibcast`, `MPI_Iscatter`, and `MPI_Iscatterv` all induce *Late Broadcast* wait states. No-wait-state test cases include a variant where a nonblocking collective is immediately followed by a wait call after the same balanced computation on every rank, so that all ranks are expected to enter the calls at the same time, and a variant that applies a staggered delay before initiation but introduces an equal amount of computation between initiation and completion on every rank, long enough to absorb the staggered delay, so that every rank reaches `MPI_Wait` only after the latest rank has already initiated the operation. Additional test cases for the *v* variants, including ranks contributing no data and therefore expected to incur no waiting time, were also performed. These cases are not discussed individually.

5.1.1 Wait at Barrier

The first test case examined corresponds directly to a blocking case. All ranks overlap in their completion, since completion immediately follows initiation. In Listing 1, each rank performs a staggered delay before initiating the barrier, computed as

$$0.1 \cdot (\min(\text{size}, \text{REQUIRED_MPI_RANKS}) - (\text{rank} \bmod \text{REQUIRED_MPI_RANKS}))$$

with `REQUIRED_MPI_RANKS` set to 8. With 8 processes, this reduces to $0.1 \cdot (8 - \text{rank})$ seconds³, giving rank 0 the longest delay (0.8 s) and rank 7 the shortest (0.1 s). Rank 0 is therefore guaranteed to initiate the barrier last, causing a *Wait at Barrier* wait state on all other ranks. The Scalasca analysis confirmed this: the observed waiting time on rank *i* was 0.*i* seconds for $i = 1, \dots, 7$, with rank 0 itself showing no waiting time.

In the blocking case, the *Wait at Barrier* wait state manifests on every rank that enters the call before the latest rank. This no longer holds in the nonblocking case: even if a rank initiates the operation earlier than the latest rank, its completion may happen asynchronously and therefore does not need to overlap with the latest rank's initiation. Listing 2 demonstrates this. All ranks perform the same staggered delay via `comp2`, but

³The modulo and minimum operations have no effect at this process count and only serve to bound the waiting time when the test is run with more than 8 processes.

only even ranks perform additional computation between initiating and completing the barrier, guaranteeing that their completion call falls after the latest initiation.

```
void
Test_ibarrier_wait_at_barrier(MPI_Comm comm, int size, int rank)
{
    double delay = 0.1 * (MIN(size, REQUIRED_MPI_RANKS) - (rank % REQUIRED_MPI_RANKS));
    comp2(delay);
    MPI_Request request;
    MPI_Ibarrier(comm, &request);
    MPI_Wait(&request, MPI_STATUS_IGNORE);
}
```

Listing 1: *Test case for Wait at Barrier wait states on all ranks except the latest (rank 7 on MPI_COMM_WORLD with 8 ranks).*

```
void
Test_ibarrier_waitstate_some(MPI_Comm comm, int size, int rank)
{
    double delay = 0.1 * (MIN(size, REQUIRED_MPI_RANKS) - (rank % REQUIRED_MPI_RANKS));
    comp2(delay);
    MPI_Request request;
    MPI_Ibarrier(comm, &request);
    if (rank % 2 == 0)
    {
        comp3(0.1 * MIN(size, REQUIRED_MPI_RANKS));
    }
    MPI_Wait(&request, MPI_STATUS_IGNORE);
}
```

Listing 2: *Test case for Wait at Barrier wait states on odd ranks only.*

Figure 27 shows the execution timeline for the code in Listing 2. The green circles mark the barrier initiation calls; the purple MPI_Barrier execution is not part of the test function itself but the subsequent blocking barrier that re-synchronizes all processes before the next test function is executed. Only odd ranks are expected to exhibit wait states. The amount of expected waiting time is the time spent in the wait call before the latest initiation, marked by the red dashed line. Interestingly, even ranks also spend significant time in the wait call, suggesting their completion is not fully asynchronous. This reflects an internal behavior of the MPI implementation and is not a synchronization guaranteed by the MPI standard; it is therefore not classified as waiting time based on the OTF2 event model, though it is reflected in the completion time metric, since the first rank already left its wait call.

The Scalasca analysis confirmed the expected wait states: ranks 1, 3, 5, and 7 showed waiting times of 0.1 s, 0.3 s, 0.5 s, and 0.7 s respectively, while ranks 0, 2, 4, and 6 showed no waiting time. Ranks 2, 4, and 6 instead showed completion times of 0.2 s, 0.4 s, and 0.6 s, consistent with the execution times visible in the timeline view.

5.1.2 Late Broadcast

The representative test case for *Late Broadcast* wait states mirrors the structure of the second nonblocking barrier test. Here, only a subset of ranks overlaps their completion

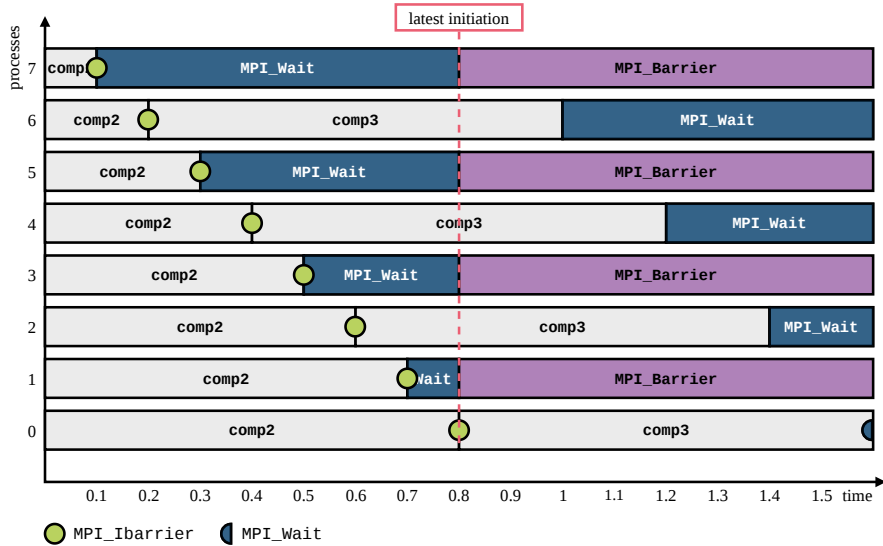


Figure 27: Timeline view of execution for Listing 2.

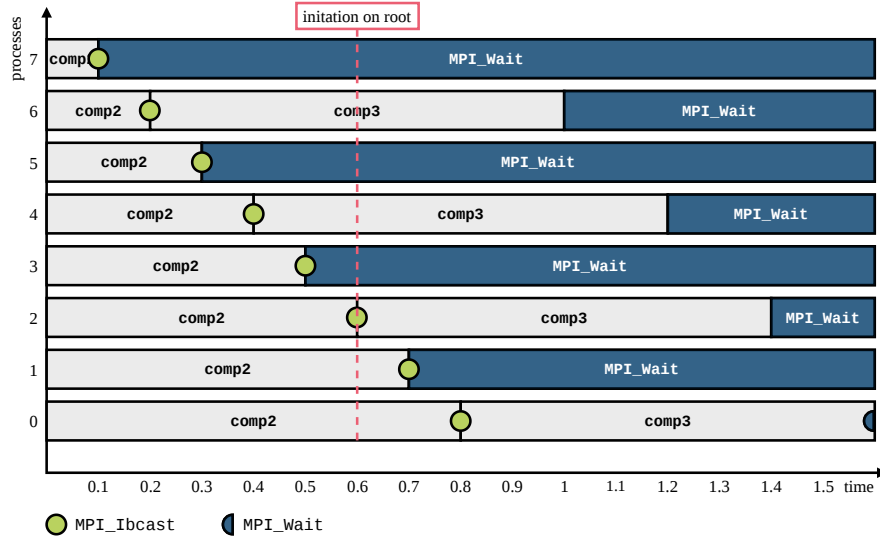
with the root’s initiation, so that waiting time is incurred only on those ranks. In Listing 3 rank 3 serves as root for a broadcast. All ranks perform the same staggered delay via comp2 as in the barrier test case, and even ranks additionally perform computation in comp3 before completion, ensuring that their completion call falls after root’s initiation.

```
void
Test_ibcast_late_broadcast_some(MPI_Comm comm, int size, int rank)
{
    const int root = 3;
    int* buffer = (int*)malloc(1 * sizeof(int));
    buffer[0] = (rank == root) ? size : -1;
    const double delay = 0.1 * (MIN(size, REQUIRED_MPI_RANKS) - (rank % REQUIRED_MPI_RANKS));
    comp2(delay);
    MPI_Request request;
    MPI_Ibcast(buffer, 1, MPI_INT, root, comm, &request);
    if (rank % 2 == 0)
    {
        comp3(0.1 * MIN(size, REQUIRED_MPI_RANKS));
    }
    MPI_Status status;
    MPI_Wait(&request, &status);
    free(buffer);
}
```

Listing 3: Test case for Late Broadcast wait states on a subset of ranks.

Figure 28 shows the execution timeline for the code in Listing 3. The timeline suggests that some synchronization takes place within the wait call itself, which is not guaranteed by the MPI standard. However, the underlying MPI implementation of the nonblocking operation is not visible at the level of the recorded event model. Only the time spent in the wait call before root’s initiation, marked by the red dashed line, is expected to be classified as waiting time by Scalasca.

The Scalasca analysis confirms this expectation, reporting *Late Broadcast* waiting times of 0.2 s on rank 5 and 0.4 s on rank 7. On even ranks, the computation in comp3 is sufficient

Figure 28: *Timeline view of execution for Listing 3.*

to cover the delayed initiation on root, so no waiting time is reported there. The same applies to ranks 0 and 1, whose staggered delay in comp2 moves their initiation past that of root on rank 3.

5.1.3 Early Reduce

For *Early Reduce* wait states, the root rank incurs waiting time because the last non-root rank initiates the operation after the root has already entered the completion call. Listing 5 (see Appendix, page 92) constructs this scenario as follows. Rank 7 serves as root, and all ranks perform a staggered delay via comp2 before initiating the reduce, followed by the same staggered delay in comp3 between initiation and completion. An *Early Reduce* wait state is expected on root, corresponding to the difference between root's entry into the wait call and the latest non-root initiation.

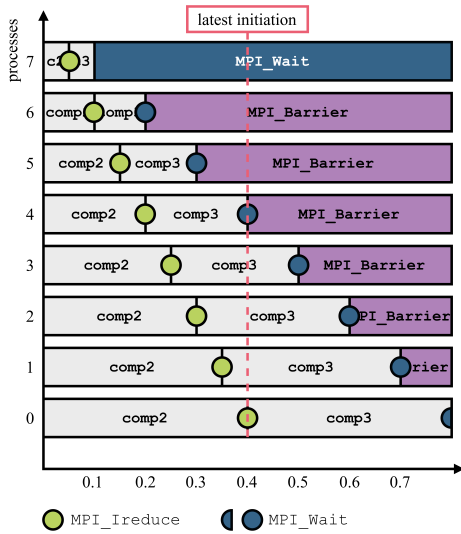
Figure 29: *Timeline view of execution for Listing 5.*

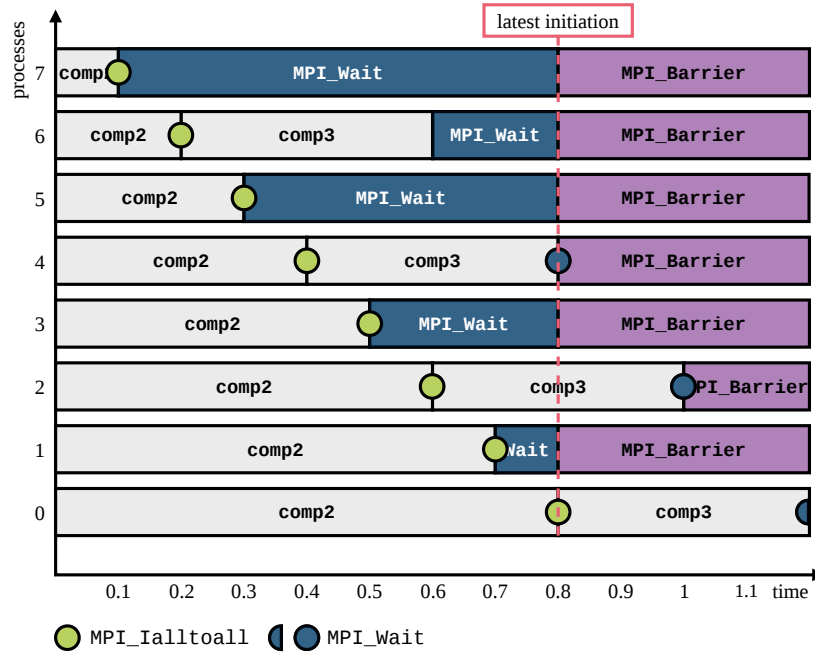
Figure 29 shows the execution timeline for the code in Listing 5. At first glance, root's wait call appears to block until the completion of the latest non-root rank rather than merely until its initiation. But only the portion of time spent in the wait call before the latest non-root initiation, marked by the red dashed line, is expected to be classified as waiting time. The non-root ranks use the eager protocol, so their reduce operations complete asynchronously. The Scalasca analysis reports an *Early Reduce* wait state of 0.3 s on root, matching the difference between root's entry into the wait call and the latest non-root initiation.

Figure 1 is a Gantt chart illustrating the execution of MPI_Isend and MPI_Wait on 8 processes (0-7). The x-axis represents time from 0 to 0.8, and the y-axis represents processes. The chart shows the timeline of communication components (comp2, comp3) and MPI_Wait. A vertical dashed red line at time 0.5 indicates the latest initiation among lower-ranked processes. MPI_Isend is represented by green circles, and MPI_Wait by blue circles. The chart demonstrates that MPI_Wait on process 0 completes at time 0.5, while MPI_Isend on process 7 is still in progress at that time.

Figure 30 shows the execution timeline of the code in Listing 6. The red dashed line marks the latest initiation among the lower-ranked processes. Interestingly, ranks 0, 1, and 6 appear to synchronize within their wait calls, although such synchronization is not mandated by the MPI standard and reflects an internal behavior of the MPI implementation. Since the Scalasca analysis is based on the event model introduced in Section 2.2.2, only the waiting time on ranks 5 and 7 is expected to be classified as a wait state. The Scalasca analysis confirms this, reporting *Early Scan* waiting time of 0.2 s on rank 5 and 0.4 s on rank 7.

To demonstrate the detection of *Wait at $N \times N$* wait states, the code in Listing 7 (see Appendix, page 93) is considered. All ranks execute a staggered delay in `comp2` before initiating an all-to-all operation, and even ranks perform additional computation in `comp3` between initiation and completion. This additional computation only covers the delayed initiation on some of the ranks.

70

Figure 31: *Timeline view of execution for Listing 7.*

the latest initiation, marked by the red dashed line, is expected to be detected as *Wait at $N \times N$ waiting time.*

The Scalasca analysis reports waiting times of 0.1 s on rank 1, 0.3 s on rank 3, 0.5 s on rank 5, 0.2 s on rank 6—where comp3 does not fully cover the delayed initiation—and 0.7 s on rank 7, matching the expectations.

5.1.6 Most-severe Instance Tracking in Multi-completion

In addition to testing each wait-state class individually, tests for most-severe instance tracking in multi-completion calls were conducted. These covered each wait-state class, being the most-severe instance to verify the functionality across all combinations. The code in Listing 8 (see Appendix, page 94) is considered here, where the *Late Broadcast* wait state is the most-severe instance.

Figure 32 shows the execution timeline. The relevant ranks are 0, 1, and 2; the remaining ranks 3 through 7 only execute the broadcast and complete it via a wait. Rank 0 serves as the root for the broadcast. It first spends 0.5 s in comp2, then initiates a send to rank 1, spends 0.1 s in comp3, and finally initiates the broadcast before completing both operations in a single MPI_Waitall call. Rank 1 spends 0.1 s in computation between each of three operations: initiating a receive for the message from rank 0, initiating a synchronous send to rank 2, and initiating the nonblocking broadcast, before entering its own MPI_Waitall. Since rank 1 posts its completion call earlier than rank 0 initiates the matching send for the

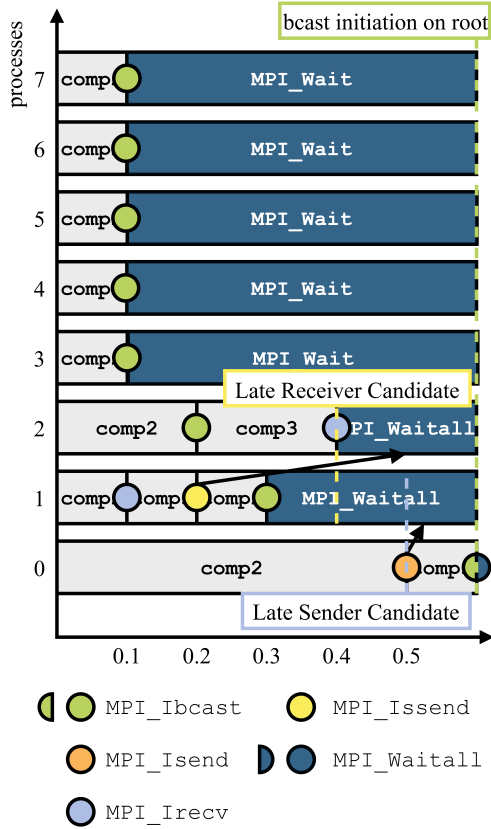


Figure 32: Timeline view of execution for Listing 8.

Broadcast wait state of 0.2 s is expected on rank 2, and 0.5 s on ranks 3 through 7. The Scalasca analysis confirmed these expectations: the reported wait state on rank 1 was indeed classified as *Late Broadcast*, with the wait times on ranks 2 and 3 through 7 matching the values derived from the timeline.

5.2 Case Study

To evaluate the extended wait-state analysis, a parallel 2D abelian sandpile simulation is used as a benchmark application. The sandpile model was first introduced by Bak et al. [30]. The code is based on an MPI programming exercise developed by David Henty at the Edinburgh Parallel Computing Center (EPCC) at the University of Edinburgh. It was originally used as part of an MPI programming course. It implements a simple cellular automaton on a two-dimensional integer grid of size $L \times L$. The simulation starts with placing a single large pile of sand at a user-defined position on the grid. In each time step, every cell that exceeds a threshold height of three loses four of its grains by distributing one grain to each of its four direct neighbors, while simultaneously receiving grains from neighboring cells that also topple during the same iteration. This is repeated for a fixed

initiated receive, part of its execution time constitutes *Late Sender* time, indicated by the light blue dashed line. The green dashed line marks the initiation of the broadcast on root and therefore indicates the *Late Broadcast* wait-state candidate, which spans the entire execution of the waitall call. Rank 2 executes 0.2 s of computation before initiating the broadcast and a further 0.2 s before initiating the receive matching the synchronous send from rank 1; this later initiation, marked by the yellow dashed line, defines the portion of rank 1's *MPI_Waitall* that constitutes the *Late Receiver* candidate. Ranks 3 through 7 each perform 0.1 s of computation before initiating the broadcast and subsequently completing it via a wait call.

Given these timings, the *Late Broadcast* candidate on rank 1 is expected to be the most-severe instance and therefore the one reported, taking precedence over both the *Late Sender* and *Late Receiver* candidates. Additionally, a *Late*

number of steps. The simulation converges when no cell changes its height in a given step, at which point the iteration loop could be stopped early.

The grid is decomposed across MPI processes using a one-dimensional decomposition, where each process owns a vertical stripe of $\frac{L}{P}$ columns, with P denoting the total number of processes. The decomposition is assumed to be exact, that is, L is divisible by P . Since the update rule involves nearest-neighbor interactions, each process requires one row of data from its lower-ranked and one row from its higher-ranked process at the start of every timestep. These inter-process boundary rows are stored in ghost cells, referred to as halos, which are appended at the indices 0 and $\frac{L}{P} + 1$. Processes at the physical boundary of the domain use the special MPI rank `MPI_PROC_NULL` to represent the absent neighbor so that the corresponding send and receive operations become no-ops, and the halo rows remain at their initialized value of zero. After the loop completes, each process copies its cells into a local array without halos, and an `MPI_Gather` collects the full grid on rank 0 for output.

The main time-stepping loop is shown in Listing 9 (see Appendix, page 95). It starts with the halo exchange of the inter-process boundary rows using two calls to `MPI_Sendrecv`. Each process iterates over its inner cells and applies the update rule using two arrays: one for accessing the heights from the latest iteration, and one to insert the updated heights. A process-local change counter is incremented for each cell whose height differs from the previous iteration. The local change counts are aggregated across all processes to obtain a global count on rank 0. The code can be compiled in three variants, either using a blocking `MPI_Reduce`, a nonblocking `MPI_Ireduce` immediately followed by an `MPI_Wait`, or an overlapping variant with a deferred call to `MPI_Wait`. The updated array is copied into the old one for the subsequent step. In the overlap variant, this happens between the reduce initiation and the completion, whereas both of the other variants complete the reduction prior to the data copy. Since the *Early Reduce* wait state detected by Scalasca is inherently limited to the root process, an analogous variant of the code was created where the reduction has been replaced by an allreduce operation, involving all processes in the collective synchronization.

5.2.1 Experimental Setup

The measurements for the reduce variant were run on ARCHER2 [31], the UK National Supercomputing Service. ARCHER2 is an HPE Cray EX system comprising 5,860 CPU compute nodes, each with two 64-core AMD EPYC 7742 processors and interconnected via an HPE Slingshot network. To avoid shared-memory optimizations within a node, all runs used one process per node across 32 nodes. Four problem sizes were evaluated: $L \in \{512, 1024, 2048, 4096\}$, with 5 repetitions per configuration. The elapsed time was measured using `MPI_Wtime`. Barriers are inserted before starting and stopping the timer. The timer measures the time spent in the time-stepping loop, excluding MPI initialization

and file output time. For the allreduce measurements, two HPC systems were used to compare results across different MPI implementations and network architectures. In addition to ARCHER2 (Cray MPICH 3.1.27), measurements were conducted on the JUWELS Cluster Module [32] at the Jülich Supercomputing Centre. The JUWELS Cluster Module has 2271 standard compute nodes, each equipped with two 24-core Intel Xeon Platinum 8168 processors and an Infiniband EDR interconnect. Measurements on JUWELS were carried out with both OpenMPI 5.0.3 and ParaStationMPI 5.13.0, an MPICH-based implementation. As on ARCHER2, all runs used one process per node with 32 processes in total, and the same four problem sizes were evaluated with 10 repetitions per configuration. On both systems, the traces were collected using Score-P 9.4 [29] and the wait-state metrics were computed with a Scalasca prototype⁴. The instrumentation overhead was measured on ARCHER2 by comparing instrumented and uninstrumented runtimes across all configurations. It ranges from below 1 % for the largest problem size ($L = 4096$) to at most 9.3 % for the smallest ($L = 512$), with an absolute difference below 0.2 s in all cases, and is considered acceptable.

5.2.2 Reduce Variant

Table 4 (see Appendix, page 93) shows the elapsed time per run and the Scalasca wait-state metrics for the reduce variant on ARCHER2. The runtime is nearly identical across all three variants at each problem size. Counterintuitively, the total time spent across all ranks in the `MPI_Reduce` or `MPI_Wait` call is lower for the blocking variant than for both nonblocking variants at all problem sizes except $L = 4096$, where the overlap variant shows a mean more than four seconds lower. This result must be interpreted with care given the high standard deviations of around 2.5–3.0 seconds across all three variants at that size, though. Since Scalasca detects *Early Reduce* waiting time only on the root process, the analysis focuses on the time the root process spends in the completion call. This time stays roughly constant across all variants for different problem sizes, including $L = 4096$. However, the fraction classified as waiting time drops sharply in the overlap variant, and increasingly so at larger problem sizes. At $L = 4096$, the blocking variant incurs 1.412 seconds of *Early Reduce* waiting time (74% of root time), while the overlap variant reduces this to 0.315 seconds (17%), which is a reduction of over 75%. The nonblocking variant without overlap shows only marginal reduction in waiting time. Despite this, the runtime remains nearly identical across all variants, even at $L = 4096$ where a decrease in collective completion call time is evident. A closer inspection of the traces at $L = 4096$ reveals that the overlap variant incurs increased time in `MPI_Sendrecv`, which nullifies the reduction in collective completion time and accounts for the absence of an overall runtime improvement. Since the *Early Reduce* wait state is inherently limited to the root process, a complete picture

⁴<https://gitlab.jsc.fz-juelich.de/perf/tools/scalasca/-/tree/145-handle-nonblocking-collectives> [only accessible for authorized accounts]

of the synchronization behavior across all participating ranks cannot be obtained from this variant. To include all processes in the synchronization and allow wait-state detection across all participating ranks, the allreduce variant is examined next.

5.2.3 Allreduce Variant: Runtime

Table 3 lists the total elapsed time per run for the allreduce variant. As expected, the blocking variant and the nonblocking variant without overlap behave similarly, since in the latter case the `MPI_Wait` is called immediately after `MPI_Iallreduce` without any computation in between. Only the overlap variant has the potential to reduce runtime by hiding communication latency behind computation. However, the results show that the application does not benefit from this in practice. In fact, the blocking variant is the fastest in most configurations. This motivates a closer look at the trace data. Notably, the nonblocking variant without overlap is consistently slower than the blocking baseline across all problem sizes and MPI implementations, which is likely due to the overhead of the additional nonblocking initiation call.

System (MPI)	L	blocking (s)	nonbl. no overlap (s)	nonbl. overlap (s)
JUWELS (OpenMPI)	512	1.159 (0.006)	1.377 (0.005)	1.368 (0.004)
	1024	2.632 (0.010)	2.859 (0.023)	2.787 (0.025)
	2048	8.753 (0.051)	9.020 (0.056)	8.904 (0.080)
	4096	35.987 (0.074)	36.470 (0.167)	36.254 (0.118)
JUWELS (ParaStationMPI)	512	1.280 (0.009)	1.386 (0.012)	1.400 (0.009)
	1024	2.795 (0.013)	2.894 (0.010)	2.895 (0.012)
	2048	9.107 (0.049)	9.230 (0.054)	9.252 (0.065)
	4096	36.365 (0.193)	36.436 (0.166)	36.314 (0.140)
ARCHER2 (Cray MPICH)	512	1.653 (0.019)	1.740 (0.012)	1.724 (0.016)
	1024	2.663 (0.017)	2.786 (0.019)	2.753 (0.022)
	2048	6.916 (0.049)	7.083 (0.048)	7.075 (0.033)
	4096	22.726 (0.189)	22.837 (0.121)	22.884 (0.059)

Table 3: Overall elapsed time for the sandpile benchmark (allreduce variant) on JUWELS (OpenMPI and ParaStationMPI) and ARCHER2 (Cray MPICH). Values are means with standard deviation in parentheses. 10 runs per configuration, 32 processes.

5.2.4 Allreduce Variant: Trace Analysis

Table 5 (see Appendix, page 96) shows the time spent in the completion call, summed over all 32 processes, along with the portion classified as *Wait at $N \times N$* and *$N \times N$ Completion*. Additionally, the time spent in `MPI_Sendrecv` is reported, together with the fraction attributed to *Late Sender* wait states, which becomes particularly relevant for the Cray MPICH runs on ARCHER2.

Although total execution time is largely unaffected, the trace analysis results reveal clear shifts in waiting times. The total time spent in the collective call remains roughly constant across all three variants for a given configuration, but the time classified as a wait state decreases substantially in the nonblocking overlap variant. This is particularly visible at larger problem sizes.

Figure 33 shows the wait-state metrics for $L = 4096$ on JUWELS, with ParaStationMPI results in the left panel and OpenMPI in the right. The dark blue bars correspond to the blocking variant, the light blue bars to the nonblocking immediate-wait variant, and the green bars to the overlap variant. The five metrics from Table 5 are displayed, whereby *Wait at $N \times N$* and *$N \times N$ Completion* are sub-components of the collective total time, and *Late Sender* is a sub-component of the MPI_Sendrecv total time. For the blocking variant and the nonblocking immediate-wait variant, the collective total time can be decomposed almost entirely into waiting time and completion time. All metrics except *Wait at $N \times N$* are consistent across all three variants, considering the red error bars representing ± 1 standard deviation. In the left panel, the *Wait at $N \times N$* drops from 85% to 9%. OpenMPI shows the same pattern, with comparable magnitudes.

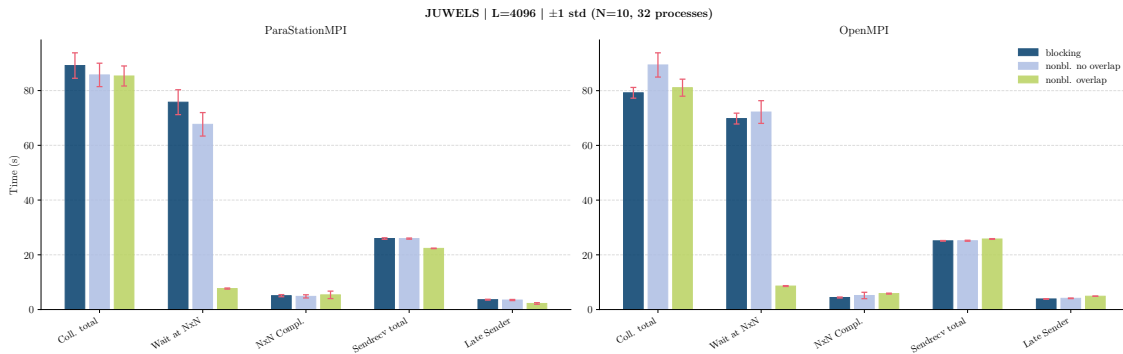


Figure 33: *Wait-state metrics for the sandpile benchmark (allreduce variant) on JUWELS, $L = 4096$.*

Figure 34 shows the corresponding bar chart for ARCHER2 with Cray MPICH. The bar chart displays the same five metrics and three-variant grouping, with the y -axis extending to 120 seconds compared to 80 seconds on JUWELS. The higher absolute times on ARCHER2 reflect the different system and are not directly comparable to JUWELS. The absolute values are only comparable across the three variants within the same system, and cross-system comparisons should be interpreted with caution. At $L = 4096$ on ARCHER2, the *Wait at $N \times N$* metric drops from approximately 101 seconds in the blocking case to just 8.7 seconds in the overlap variant. What stands out is that at the same time *$N \times N$ Completion* time increases from 6.8 to 28.8 seconds. This increase is specific to ARCHER2. The *Late Sender* wait states in MPI_Sendrecv are consistently low on JUWELS across all variants and problem sizes, accounting for roughly 10–20% of the total MPI_Sendrecv

time. On ARCHER2, however, the overlap variant shows a striking increase in *Late Sender* time, reaching approximately 47% of MPI_Sendrecv time at $L = 4096$.

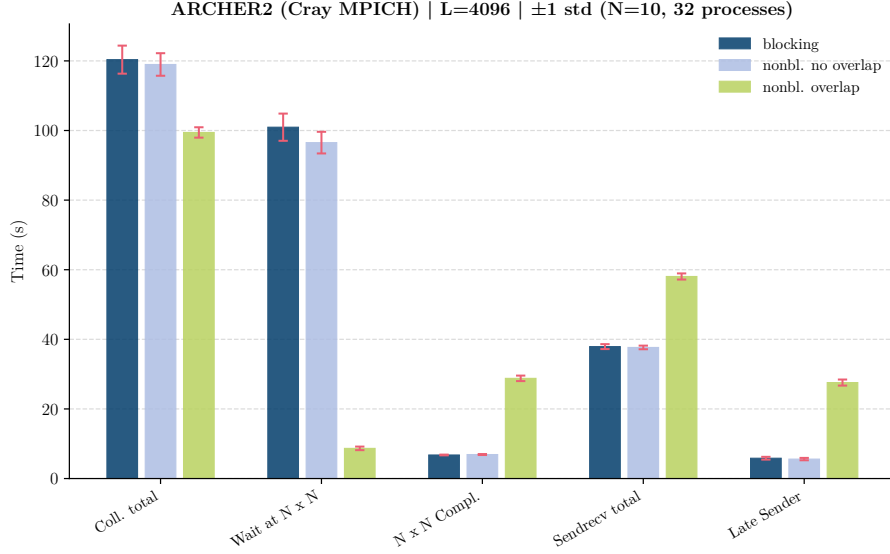


Figure 34: Wait-state metrics for the sandpile benchmark (allreduce variant) on ARCHER2, $L = 4096$.

To investigate the ARCHER2-specific behavior further, the traces were examined in Vampir. Figure 35 shows a zoomed-in view of a single iteration for all three variants. The upper panel shows the blocking variant, with time in MPI_Allreduce marked in red, and MPI_Sendrecv in dark blue. The middle panel shows the nonblocking immediate-wait variant, with the MPI_Iallreduce call marked in yellow and its completion in red. The lower panel shows the overlap variant. Here, odd and even processes exit the completion call at noticeably different times, causing some ranks to enter the subsequent MPI_Sendrecv call earlier than others. Since the point-to-point messages are exchanged with direct neighbors, this imbalance introduces *Late Sender* wait states in MPI_Sendrecv. The completion time is therefore responsible for the increase in *Late Sender* waiting time and offsets the benefit of the decreased overall time spent in the MPI_Wait call as the time is effectively shifted to the MPI_Sendrecv call.

The fact that the total time spent in MPI_Wait did not decrease, or in the ARCHER2 case was accompanied by an increase in completion time and *Late Sender* wait states, points to insufficient background progress by the MPI implementation. The observed behavior suggests that communication is deferred to the completion call rather than progressing asynchronously during the computation phase, negating the benefit of the nonblocking initiation. Prior work has shown that asynchronous progress in MPI collective operations varies substantially across implementations and that many MPI libraries do not leverage available overlap potential without dedicated optimization [33]. In particular, implementations have been shown to make little to no asynchronous progress for small buffer sizes [34], which is the case in this benchmark sending only a single integer per

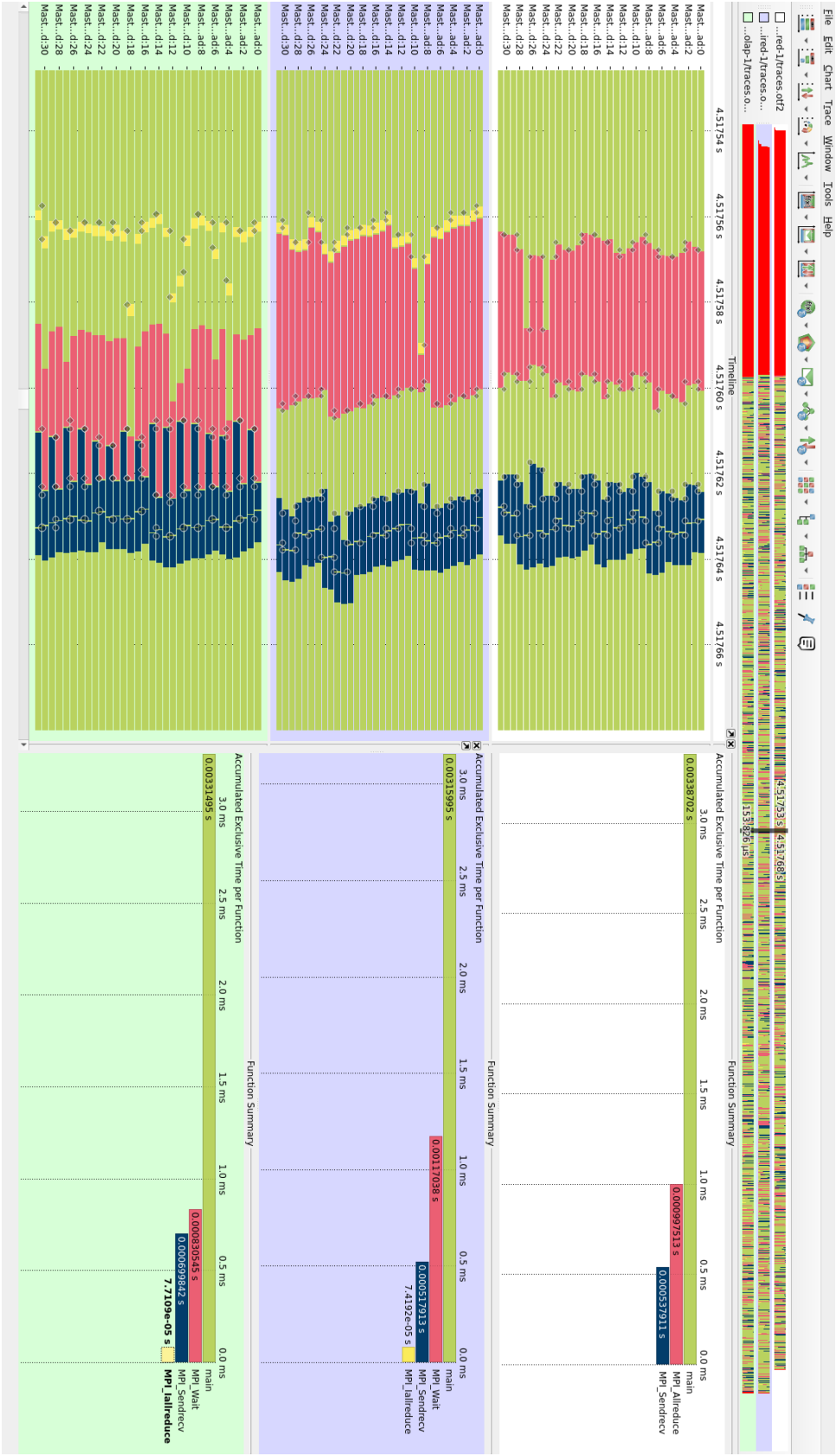


Figure 35: Vampir trace view of a single sample iteration (allreduce variant) on ARCHER2, L = 4096, showing the blocking (top), nonblocking immediate-wait (middle), and nonblocking overlap (bottom) variants

process. As Score-P does not record the internal point-to-point messages the MPI library uses to execute a collective operation, this behavior remains opaque. What can be inferred is that the time spent in the completion call likely comprises both the actual data transfer and waiting induced by the internal point-to-point exchange through which the MPI library implements the collective. This leads the application developer to investigate whether the MPI implementation can be nudged towards asynchronous progress. Possible starting points include implementation-specific options that influence collective algorithms, inserting `MPI_Test` calls during the computation phase to guarantee progress, or trying alternative MPI implementations.

The Scalasca analysis nonetheless shows a clear reduction in waiting time for the overlap variant. It therefore achieves its intended effect at the application level, since wait states reflect a portion of time that the application developer can potentially reduce. On ARCHER2, this reduction is counteracted by an increase in *Late Sender* wait states. The analysis results suggest that inserting additional computation between `MPI_Wait` and the subsequent `MPI_Sendrecv` could help mitigate this. Overall, the analysis demonstrates that wait states in nonblocking collectives do appear in practice and that the extended Scalasca wait-state detection is able to identify them.

6 Conclusion and Future Work

This thesis extended Scalasca’s replay-based wait-state analysis to nonblocking collective MPI operations. As a foundation, wait states for nonblocking collectives were defined analogously to their blocking counterparts, and additionally, wait-state classes specific to the rendezvous protocol were introduced. The central challenge in extending the analysis lies in exchanging the timestamps required for wait-state calculation. While the MPI standard enforces a consistent ordering of collective initiation calls across all ranks on the same communicator, completion calls may occur in arbitrary order and can fall inside local test calls that return immediately. Since replay timing differs from the original execution, there is no guarantee that an analysis operation initiated at the request event would be ready to complete at a corresponding completion event occurring in a test region. This rules out a single-pass approach and motivated the two-pass design implemented in this work. A dedicated forward pass exchanges the required timestamps by initiating nonblocking collectives at the `Collective Request` event and completing them no later than when all trace events have been processed, making the results available for the subsequent wait-state detection pass. In addition, most-severe instance tracking in multi-completion calls was extended to incorporate nonblocking collective wait states, allowing them to be compared against point-to-point wait states within the same completion region. The approaches for synchronization point insertion and for the delay and critical-path analysis that builds on them were outlined, though their implementation is left as future work.

Functional correctness was verified against a dedicated test suite covering all wait-state classes, including scenarios specific to nonblocking collectives. In all cases, the prototype implementation correctly identified the expected wait states, while the current Scalasca main branch missed them. Finally, the practical benefit of the extended analysis was demonstrated using a sandpile benchmark run on two HPC systems. The results show that the extended wait-state detection is able to identify wait states in nonblocking collectives in practice and that the overlap variant of the benchmark achieves a substantial reduction in *Wait at $N \times N$* waiting time compared to the blocking variant, even where the overall runtime benefit is limited by insufficient asynchronous progress in the MPI implementation.

Several directions remain open for future investigation. The most immediate next step is the implementation of synchronization point insertion for nonblocking collectives, followed by the delay and critical-path analysis that builds on them. As discussed in Section 4.3, the main challenge for the delay and critical-path analysis lies in exchanging the required data during the backward replay pass without deadlocking with the blocking MPI calls already present in the analysis. The outlined point-to-point exchange is a promising direction, and would additionally allow identical handling of blocking and nonblocking collectives in the delay analysis, simplifying the overall design. However, verifying that the approach is indeed deadlock-free in all cases will require careful analysis and practical testing.

A further direction concerns the OTF2 event model itself. Currently, Score-P does not record the internal point-to-point messages that an MPI library uses to implement a collective operation. As a consequence, several classes of wait states that arise from specific collective algorithms cannot be detected from the recorded event data alone. These include intermediate wait states within the underlying point-to-point exchange pattern of a collective operation (e.g., at inner nodes of a tree-based algorithm). Such wait states cannot be identified without knowledge of how the collective is decomposed into point-to-point messages. Undetectable patterns also include wait states tied to the rendezvous protocol, such as the *Early Broadcast* and *Late Reduce* wait states discussed in Section 2.2.1. In principle, these could be approximated using a full-overlap heuristic across all processes. In practice, however, different processes may transfer different amounts of data in the underlying point-to-point exchanges, meaning that even within a single collective operation only some of the internal transfers may use a rendezvous protocol. Which messages cross that threshold depends on the MPI implementation's eager limit, an implementation-specific value. The analysis therefore cannot reliably classify internal transfers as eager or rendezvous without access to both the actual internal transfers and the corresponding threshold. That such internal transfers are in principle observable has been demonstrated by Bosilca et al. [35], whose runtime monitoring library hooks into OpenMPI's internal message layer and records the individual point-to-point exchanges underlying collective operations. Extending Score-P to capture these transfers as trace events would make the corresponding wait states detectable in a post-mortem analysis and give deeper insight into the communication behavior of collective operations. This is particularly relevant in cases where the observed execution deviates from expectations, as seen in some of the functional correctness tests and the sandpile results. Access to the internal message transfers would make such behavior directly observable, rather than leaving it as a black box that can only be inferred indirectly from the analysis report. However, surfacing this information usefully to the user is a further challenge. A wait state attributable to the internal communication algorithm of the MPI library must be distinguished from one arising at a synchronization point defined by the MPI standard. Access to the internal transfers would additionally reveal whether the MPI library actually progresses the operation asynchronously, or whether progress is deferred until the completion call. A further practical obstacle is portability, since the library of Bosilca et al. relies on an instrumentation component internal to OpenMPI that has no standardized counterpart in all implementations. However, the Score-P and Scalasca tool suite is designed to work across MPI implementations. Capturing intra-collective message transfers in a fully portable way would therefore either require a standardized interface or a separate implementation-specific instrumentation layer for each supported MPI library.

References

- [1] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 5.0*. June 2025. URL: <https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf>.
- [2] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. “Scalable Parallel Trace-Based Performance Analysis”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 303–312. DOI: 10.1007/11846802_43.
- [3] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. “A scalable tool architecture for diagnosing wait states in massively parallel applications”. In: *Parallel Computing* 35.7 (2009), pp. 375–388. DOI: 10.1016/j.parco.2009.02.003.
- [4] C. Feld et al. “The Score-P Performance Tools Ecosystem”. In: *Frontiers in High Performance Computing 3 - 2025* (2026). DOI: 10.3389/fhpcp.2025.1709051.
- [5] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. Nagel, and F. Wolf. “Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries”. In: vol. 22. Jan. 2012, pp. 481–490. DOI: 10.3233/978-1-61499-041-3-481.
- [6] D. Böhme, M. Geimer, F. Wolf, and L. Arnold. “Identifying the Root Causes of Wait States in Large-Scale Parallel Applications”. In: *2010 39th International Conference on Parallel Processing*. 2010, pp. 90–100. DOI: 10.1109/ICPP.2010.18.
- [7] J. E. Kelley and M. R. Walker. “Critical-path planning and scheduling”. In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM ’59 (Eastern). Boston, Massachusetts: Association for Computing Machinery, 1959, pp. 160–173. ISBN: 9781450378680. DOI: 10.1145/1460299.1460318.
- [8] D. Böhme, F. Wolf, B. R. de Supinski, M. Schulz, and M. Geimer. “Scalable Critical-Path Based Performance Analysis”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 2012, pp. 1330–1340. DOI: 10.1109/IPDPS.2012.120.
- [9] M. Gerndt. “Performance Analysis Tools”. In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Boston, MA: Springer US, 2011, pp. 1515–1522. DOI: 10.1007/978-0-387-09766-4_267.
- [10] A. Munera, S. Royuela, G. Llort, E. Mercadal, F. Wartel, and E. Quiñones. “Experiences on the Characterization of Parallel Applications in Embedded Systems with Extrae/Paraver”. In: Aug. 2020, pp. 1–11. DOI: 10.1145/3404397.3404440.

- [11] L. Ad hianto et al. “HPCToolkit: Tools for performance analysis of optimized parallel programs”. In: *Concurrency and Computation: Practice and Experience* 22 (Dec. 2009). DOI: 10.1002/cpe.1553.
- [12] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. “VAMPIR: Visualization and analysis of MPI resources”. In: *Supercomputer XII.1* (1996), pp. 69–80. URL: <http://hdl.handle.net/2128/11887>.
- [13] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. “Using automated performance modeling to find scalability bugs in complex codes”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’13. Denver, Colorado: Association for Computing Machinery, 2013. DOI: 10.1145/2503210.2503277.
- [14] S. S. Shende and A. D. Malony. “The Tau Parallel Performance System”. In: *Int. J. High Perform. Comput. Appl.* 20.2 (2006), pp. 287–311. DOI: 10.1177/1094342006064482.
- [15] R. Bell, A. D. Malony, and S. Shende. “ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis”. In: *Euro-Par 2003 Parallel Processing*. Ed. by H. Kosch, L. Böszörményi, and H. Hellwagner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 17–26. DOI: 10.1007/978-3-540-45209-6_7.
- [16] K. Huck and A. Malony. “PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing”. In: *SC ’05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 2005, pp. 41–41. DOI: 10.1109/SC.2005.55.
- [17] D. Computadors, V. Pillet, J. Labarta, T. Cortes, and S. Girona. “PARAVER: A tool to visualize and analyze parallel code”. In: *WoTUG-18* 44 (Mar. 1995).
- [18] S. Girona, J. Labarta, and R. M. Badia. “Validation of Dimemas Communication Model for MPI Collective Operations”. In: *Proceedings of the 7th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 39–46. DOI: 10.1007/3-540-45255-9_9.
- [19] R. Asbury and M. Wrinn. “MPI tuning with Intel© Trace Analyzer and Intel Trace Collector.” In: *CLUSTER*. 2004, p. 4.
- [20] NVIDIA. *NVIDIA Nsight Developer Tools*. 2026. URL: <https://developer.nvidia.com/tools-overview> (visited on 07/01/2026).
- [21] AMD. *ROCprofiler SDK*. 2026. URL: <https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/latest> (visited on 07/01/2026).

- [22] D. Becker, R. Rabenseifner, and F. Wolf. “Timestamp Synchronization for Event Traces of Large-Scale Message-Passing Applications”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by F. Cappello, T. Herault, and J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 315–325. DOI: 10.1007/978-3-540-75416-9_43.
- [23] D. Becker, R. Rabenseifner, F. Wolf, and J. C. Linford. “Scalable timestamp synchronization for event traces of message-passing applications”. In: *Parallel Comput.* 35.12 (Dec. 2009), pp. 595–607. DOI: 10.1016/j.parco.2008.12.012.
- [24] M. Geimer, P. Saviankou, A. Strube, Z. Szebenyi, F. Wolf, and B. J. N. Wylie. “Further Improving the Scalability of the Scalasca Toolset”. In: *Applied Parallel and Scientific Computing*. Ed. by K. Jónasson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 463–473. DOI: 10.1007/978-3-642-28145-7_45.
- [25] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, and B. J. N. Wylie. “A Parallel Trace-Data Interface for Scalable Performance Analysis”. In: *Applied Parallel Computing. State of the Art in Scientific Computing*. Ed. by B. Kågström, E. Elmroth, J. Dongarra, and J. Waśniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 398–408. DOI: 10.1007/978-3-540-75755-9_49.
- [26] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr. “Cube v4: From Performance Report Explorer to Performance Analysis Tool”. In: *Procedia Computer Science* 51 (Dec. 2015), pp. 1343–1352. DOI: 10.1016/j.procs.2015.05.320.
- [27] D. Krings. “Nonparametric Density Approximation of Streaming Data”. Campus Jülich. MA thesis. FH Aachen University of Applied Sciences, 2010.
- [28] D. Böhme, M. Geimer, L. Arnold, F. Voigtländer, and F. Wolf. “Identifying the Root Causes of Wait States in Large-Scale Parallel Applications”. In: *ACM Transactions on Parallel Computing* 3 (July 2016), pp. 1–24. DOI: 10.1145/2934661.
- [29] C. Feld et al. *Score-P: Scalable performance measurement infrastructure for parallel codes*. Version 9.4. Dec. 2025. DOI: 10.5281/zenodo.17964650.
- [30] P. Bak, C. Tang, and K. Wiesenfeld. “Self-Organized Criticality”. In: *Physical Review A* 38 (July 1988), pp. 364–374. DOI: 10.1103/PhysRevA.38.364.
- [31] G. Beckett et al. *ARCHER2 Service Description*. Dec. 2024. DOI: 10.5281/zenodo.14507040.
- [32] D. Alvarez. “JUWELS Cluster and Booster: Exascale Pathfinder with Modular Supercomputing Architecture at Juelich Supercomputing Centre”. In: *Journal of large-scale research facilities JLSRF* 7 (Oct. 2021). DOI: 10.17815/jlsrf-7-183.

- [33] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, and D. K. (DK) Panda. “Efficient design for MPI asynchronous progress without dedicated resources”. In: *Parallel Computing* 85 (2019), pp. 13–26. DOI: 10.1016/j.parco.2019.03.003.
- [34] A. Medvedev. “Towards Benchmarking the Asynchronous Progress of Non-Blocking MPI Operations”. In: *International Conference on Parallel Computing (ParCo)*. Prague, Czech Republic, Apr. 2020. DOI: 10.3233/APC200067.
- [35] G. Bosilca, C. Foyer, E. Jeannot, G. Mercier, and G. Papauré. “Online Dynamic Monitoring of MPI Communications”. In: *Euro-Par 2017: Parallel Processing*. Ed. by F. F. Rivera, T. F. Pena, and J. C. Cabaleiro. Cham: Springer International Publishing, 2017, pp. 49–62. DOI: 10.1007/978-3-319-64203-1_4.

List of Figures

1	Mapping of MPI operation stages to procedure calls	4
2	Point-to-point communication	7
3	Wait states in blocking point-to-point communication	10
4	Wait states in nonblocking point-to-point communication	10
5	<i>Late Broadcast</i> wait states in nonblocking collective communication	11
6	Tree-based broadcast communication	12
7	<i>Late Broadcast</i> and <i>Early Broadcast</i> wait states in nonblocking collective communication	13
8	Wait states in blocking and nonblocking barrier communication	13
9	Wait states in blocking and nonblocking reduce operations	14
10	<i>Early Scan</i> wait state in nonblocking collective operation	14
11	Wait states in multi-completion call	15
12	Direct and indirect wait states resulting from a delay	16
13	Score-P usage workflow	19
14	OTF2 events in blocking collectives	20
15	OTF2 events in nonblocking collectives	21
16	Overview of the SCOUT analysis	26
17	Detection of <i>Late Receiver</i> pattern	30
18	Callbacks involved in Wait at Barrier wait-state detection	33
19	Analysis of <i>Late Broadcast</i> wait state in blocking operation	35
20	Most-severe instance tracking in multi-completion regions	38
21	Delay analysis for collective operations	40
22	Nonblocking barriers on distinct communicators	46
23	Event sequence for the program in Listing 4	47
24	Most-severe instance tracking in multi-wait region	55
25	Synchronization points in a nonblocking barrier	59
26	Data exchange for a nonblocking barrier during delay analysis	61
27	Timeline view for <i>Wait at Barrier</i> test case (Listing 2)	68
28	Timeline view for <i>Late Broadcast</i> test case (Listing 3)	69
29	Timeline view for <i>Early Reduce</i> test case (Listing 5)	69
30	Timeline view for <i>Early Scan</i> test case (Listing 6)	70
31	Timeline view for <i>Wait at $N \times N$</i> test case (Listing 7)	71
32	Timeline view for most-severe instance tracking test case (Listing 8)	72
33	Wait-state metrics the sandpile benchmark (allreduce variant) on JUWELS, $L = 4096$	76

34	Wait-state metrics for sandpile benchmark (allreduce variant) on ARCHER2, $L = 4096$	77
35	Vampir trace view of a single sandpile iteration (allreduce variant) on ARCHER2, $L = 4096$	78

List of Tables

1 MPI Collective Operations	7
2 Formal wait-state definitions for nonblocking collectives	23
3 Elapsed time for sandpile benchmark (allreduce variant).....	75
4 Scalasca wait-state metrics and elapsed time for sandpile benchmark (reduce variant)	93
5 Scalasca wait-state metrics for sandpile benchmark (allreduce variant)	96

Appendix

```
1 #include <mpi.h>
2
3 int main(int argc, char** argv)
4 {
5     MPI_Init(&argc, &argv);
6     int rank;
7     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8     int bvalue;
9     MPI_Request breq;
10    int flag = 0;
11
12    if (rank == 0) {
13        bvalue = 42;
14        int svalue = 43;
15        MPI_Ibcast(&bvalue, 1, MPI_INT, 0, MPI_COMM_WORLD, &breq);
16        MPI_Test(&breq, &flag, MPI_STATUS_IGNORE);
17        MPI_Send(&svalue, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
18        if (!flag) {
19            MPI_Wait(&breq, MPI_STATUS_IGNORE);
20        }
21    }
22    else if (rank == 1) {
23        int rvalue;
24        MPI_Request rreq;
25        MPI_Irecv(&rvalue, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &rreq);
26        MPI_Test(&rreq, &flag, MPI_STATUS_IGNORE);
27        MPI_Ibcast(&bvalue, 1, MPI_INT, 0, MPI_COMM_WORLD, &breq);
28        MPI_Wait(&breq, MPI_STATUS_IGNORE);
29        if (!flag) {
30            MPI_Wait(&rreq, MPI_STATUS_IGNORE);
31        }
32    }
33    else {
34        MPI_Ibcast(&bvalue, 1, MPI_INT, 0, MPI_COMM_WORLD, &breq);
35        MPI_Wait(&breq, MPI_STATUS_IGNORE);
36    }
37
38    MPI_Finalize();
39    return 0;
40 }
```

Listing 4: *MPI program with tests.*

```
1 void
2 Test_ireduce_early_reduce(MPI_Comm comm, int size, int rank)
3 {
4     const int root = MIN(size, REQUIRED_MPI_RANKS) - 1;
5     int* sendbuf = (int*)malloc(1 * sizeof(int));
6     sendbuf[0] = 1;
7     int* recvbuf = NULL;
8     if (rank == root)
9     {
10         recvbuf = (int*)malloc(1 * sizeof(int));
11         memset(recvbuf, 0xff, 1 * sizeof(int));
12     }
13     double delay = 0.1 * (MIN(size, REQUIRED_MPI_RANKS) - (rank % REQUIRED_MPI_RANKS));
14     comp2(delay/2);
15     MPI_Request request;
16     if (rank == root)
17     {
18         MPI_Ireduce(sendbuf, recvbuf, 1, MPI_INT, MPI_SUM, root, comm, &request);
19     }
20     else
21     {
22         MPI_Ireduce(sendbuf, NULL, 1, MPI_INT, MPI_SUM, root, comm, &request);
23     }
24     comp3(delay/2);
25     MPI_Wait(&request, MPI_STATUS_IGNORE);
26     free(recvbuf);
27     free(sendbuf);
28 }
```

Listing 5: Test case for Early Reduce wait state on root.

```
1 void
2 Test_scan_early_scan_some(MPI_Comm comm, int size, int rank)
3 {
4     int* sendbuf = (int*)malloc(1 * sizeof(int));
5     sendbuf[0] = 1;
6     int* recvbuf = (int*)malloc(1 * sizeof(int));
7     memset(recvbuf, 0xff, 1 * sizeof(int));
8     double delay;
9     if ((rank % REQUIRED_MPI_RANKS) < 3)
10     {
11         delay = 0.3 + 0.1 * (rank % REQUIRED_MPI_RANKS);
12     }
13     else
14     {
15         delay = 0.1 * (MIN(size, REQUIRED_MPI_RANKS)
16             - (rank % REQUIRED_MPI_RANKS));
17     }
18     comp2(delay);
19     MPI_Request request;
20     MPI_Iscan(sendbuf, recvbuf, 1, MPI_INT, MPI_SUM, comm, &request);
21     if (rank % 2 == 0)
22     {
23         comp3(0.05 * MIN(size, REQUIRED_MPI_RANKS));
24     }
25     MPI_Wait(&request, MPI_STATUS_IGNORE);
26     free(recvbuf);
27     free(sendbuf);
28 }
```

Listing 6: Test case for Early Scan wait states on a subset of ranks.

```

void
Test_ialltoall_wait_at_nxn_some(MPI_Comm comm, int size, int rank)
{
    int* sendbuf = (int*)malloc(size * sizeof(int));
    for (int i = 0; i < size; ++i)
    {
        sendbuf[i] = i;
    }
    int* recvbuf = (int*)malloc(size * sizeof(int));
    memset(recvbuf, 0xff, size * sizeof(int));
    const double delay = 0.1 * (MIN(size, REQUIRED_MPI_RANKS) - (rank % REQUIRED_MPI_RANKS));
    comp2(delay);
    MPI_Request request;
    MPI_Ialltoall(sendbuf, 1, MPI_INT, recvbuf, 1, MPI_INT, comm, &request);
    if (rank % 2 == 0)
    {
        comp3(0.05 * MIN(size, REQUIRED_MPI_RANKS));
    }
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    free(recvbuf);
    free(sendbuf);
}

```

Listing 7: Test case for Wait at $N \times N$ wait states on a subset of ranks.

L	Variant	Collective call				Elapsed (s)
		Total all (s)	Root (s)	Early Reduce (s)	(%)	
512	blocking	4.13 (0.08)	0.556 (0.006)	0.095 (0.003)	17.09%	1.361 (0.014)
	nonbl. (no ovlp)	4.87 (0.21)	0.581 (0.023)	0.056 (0.022)	9.64%	1.460 (0.031)
	nonbl. (overlap)	5.42 (0.57)	0.538 (0.031)	0.034 (0.023)	6.40%	1.408 (0.035)
1024	blocking	4.87 (0.03)	0.633 (0.012)	0.160 (0.008)	25.26%	2.320 (0.007)
	nonbl. (no ovlp)	5.75 (0.32)	0.639 (0.043)	0.110 (0.039)	17.21%	2.429 (0.044)
	nonbl. (overlap)	5.93 (0.12)	0.606 (0.088)	0.033 (0.003)	5.47%	2.386 (0.089)
2048	blocking	8.32 (0.20)	0.874 (0.016)	0.373 (0.016)	42.67%	6.170 (0.010)
	nonbl. (no ovlp)	9.01 (0.60)	0.912 (0.086)	0.350 (0.080)	38.41%	6.308 (0.106)
	nonbl. (overlap)	8.63 (1.98)	1.003 (0.348)	0.128 (0.118)	12.79%	6.433 (0.355)
4096	blocking	17.16 (2.47)	1.913 (0.252)	1.412 (0.237)	73.80%	20.776 (0.281)
	nonbl. (no ovlp)	17.24 (2.64)	1.824 (0.309)	1.270 (0.306)	69.62%	20.752 (0.373)
	nonbl. (overlap)	13.01 (2.98)	1.887 (0.367)	0.315 (0.321)	16.70%	20.892 (0.420)

Table 4: Scalasca wait-state metrics and overall elapsed time for the sandpile benchmark (reduce variant) on ARCHER2 (Cray MPICH, 5 runs per configuration, 32 processes). Values are means with standard deviation in parentheses. Early Reduce percentage is relative to the time on the root process.

```
1 void
2 Test_late_broadcast_waitall(MPI_Comm comm, int size, int rank)
3 {
4     const int root = 0;
5
6     int* buffer = (int*)malloc(1 * sizeof(int));
7     buffer[0] = (rank == root) ? size : -1;
8
9     int* sendbuf = (int*)malloc(1 * sizeof(int));
10    sendbuf[0] = rank;
11
12    int* recvbuf = (int*)malloc(1 * sizeof(int));
13    memset(recvbuf, 0xff, 1 * sizeof(int));
14
15    switch (rank % size)
16    {
17        case 0:
18        {
19            MPI_Request reqarr[2];
20            MPI_Status statarr[2];
21
22            comp2(0.5);
23            MPI_Isend(sendbuf, 1, MPI_INT, rank + 1, 42, comm, &reqarr[0]);
24            comp3(0.1);
25            MPI_Ibcast(buffer, 1, MPI_INT, root, comm, &reqarr[1]);
26
27            MPI_Waitall(2, reqarr, statarr);
28            break;
29        }
30        case 1:
31        {
32            MPI_Request reqarr[3];
33            MPI_Status statarr[3];
34
35            comp2(0.1);
36
37            MPI_Irecv(recvbuf, 1, MPI_INT, rank - 1, 42, comm, &reqarr[0]);
38            comp3(0.1);
39            MPI_Issend(sendbuf, 1, MPI_INT, rank + 1, 42, comm, &reqarr[1]);
40            comp2(0.1);
41            MPI_Ibcast(buffer, 1, MPI_INT, root, comm, &reqarr[2]);
42
43            MPI_Waitall(3, reqarr, statarr);
44
45            break;
46        }
47        case 2:
48        {
49
50            MPI_Request reqarr[2];
51            MPI_Status statarr[2];
52
53            MPI_Irecv(recvbuf, 1, MPI_INT, rank - 1, 42, comm, &reqarr[0]);
54            comp2(0.2);
55            MPI_Ibcast(buffer, 1, MPI_INT, root, comm, &reqarr[1]);
56            comp3(0.2);
57            MPI_Waitall(2, reqarr, statarr);
58
59            break;
60        }
61        default:
62        {
63            MPI_Request request;
64            MPI_Ibcast(buffer, 1, MPI_INT, root, comm, &request);
65            MPI_Wait(&request, MPI_STATUS_IGNORE);
66            break;
67        }
68    }
69    free(sendbuf);
70    free(recvbuf);
71    free(buffer);
72 }
73
```

Listing 8: Test case for most-severe instance tracking, with Late Broadcast as the most-severe wait state in a multi-completion call.

```

1 for (step = 1; step <= maxstep; step++) {
2     // Swap halos up and down
3     MPI_Sendrecv(&oldh[LX][1], LY, MPI_INT, next, tag,
4                 &oldh[0][1], LY, MPI_INT, prev, tag,
5                 comm, &status);
6
7     MPI_Sendrecv(&oldh[1][1], LY, MPI_INT, prev, tag,
8                 &oldh[LX+1][1], LY, MPI_INT, next, tag,
9                 comm, &status);
10
11     localnchange = 0;
12
13     for (i=1; i<=LX; i++) {
14         for (j=1; j<=LY; j++) {
15             h = oldh[i][j];
16
17             // Sandpiles greater than MAXH tumble in all directions
18             if (h > MAXH) h = h - 4;
19
20             // Sandpile grows if neighbours tumble
21             if (oldh[i-1][j] > MAXH) h = h + 1;
22             if (oldh[i+1][j] > MAXH) h = h + 1;
23             if (oldh[i][j-1] > MAXH) h = h + 1;
24             if (oldh[i][j+1] > MAXH) h = h + 1;
25
26             if (h != oldh[i][j]) localnchange = localnchange + 1;
27             newh[i][j] = h;
28         }
29     }
30
31     // Compute the global changes on rank 0
32     // May seem strange to use a temporary variable but without it
33     // the above loop is not vectorised and runs 6 times slower!
34     tmp = localnchange;
35
36     #ifndef IREDUCE
37     MPI_Reduce(&tmp, &nchange, 1, MPI_INT, MPI_SUM, 0, comm);
38     #else
39     MPI_Ireduce(&tmp, &nchange, 1, MPI_INT, MPI_SUM, 0, comm, &request);
40     #ifndef OVERLAP
41     MPI_Wait(&request, &status);
42     #endif
43     #endif
44
45     // Copy back for next step
46     for (i=1; i<=LX; i++) {
47         for (j=1; j<=LY; j++) {
48             oldh[i][j] = newh[i][j];
49         }
50     }
51
52     #ifdef OVERLAP
53     MPI_Wait(&request, &status);
54     #endif
55
56     // Report progress of simulation
57     if (step % printfreq == 0) {
58         if (rank == 0) {
59             printf("sandpile: changes on step %d is %d\n", step, nchange);
60         }
61     }
62 }

```

Listing 9: Main loop of sandpile benchmark (reduce variant)

MPI L	Variant	Collective completion: MPI_Allreduce/MPI_Wait					MPI_Sendrecv		
		Total (s)	Wait NxN (s)	(%) NxN Compl	(%)	(%)	Total (s)	Late Sender (s)	(%)
OpenMPI	512 blocking	14.06 (0.15)	7.35 (0.18)	52.29%	3.05 (0.02)	21.69%	8.91 (0.08)	1.64 (0.03)	18.38%
	512 non-bl. (no ovlp)	19.74 (0.13)	7.83 (0.43)	39.69%	5.34 (0.37)	27.04%	8.87 (0.04)	1.79 (0.04)	20.14%
	512 non-bl. (overlap)	19.52 (0.16)	7.19 (0.61)	36.86%	5.87 (0.52)	30.08%	8.83 (0.03)	1.81 (0.04)	20.50%
	1024 blocking	17.79 (0.31)	10.98 (0.32)	61.73%	3.13 (0.04)	17.60%	11.55 (0.05)	1.78 (0.03)	15.44%
	1024 non-bl. (no ovlp)	23.67 (0.63)	11.59 (0.78)	48.98%	5.48 (1.24)	23.17%	11.49 (0.05)	2.18 (0.13)	18.98%
	1024 non-bl. (overlap)	21.56 (0.68)	7.10 (0.65)	32.94%	7.54 (0.32)	34.95%	11.39 (0.05)	2.21 (0.07)	19.45%
	2048 blocking	35.73 (1.31)	27.30 (1.34)	76.41%	4.16 (0.17)	11.65%	17.21 (0.24)	3.65 (0.34)	21.20%
	2048 non-bl. (no ovlp)	41.11 (1.35)	25.65 (1.33)	62.40%	7.28 (1.65)	17.71%	17.22 (0.24)	3.91 (0.31)	22.73%
	2048 non-bl. (overlap)	37.68 (1.98)	7.93 (0.37)	21.04%	9.56 (0.68)	25.37%	16.89 (0.07)	3.31 (0.11)	19.61%
ParaStationMPI	4096 blocking	79.22 (1.96)	69.80 (1.98)	88.11%	4.41 (0.19)	5.57%	25.12 (0.11)	3.88 (0.11)	15.44%
	4096 non-bl. (no ovlp)	89.40 (4.43)	72.19 (4.15)	80.75%	5.14 (1.17)	5.75%	25.15 (0.19)	4.11 (0.11)	16.34%
	4096 non-bl. (overlap)	81.09 (3.11)	8.59 (0.16)	10.59%	5.83 (0.17)	7.19%	25.79 (0.14)	4.88 (0.05)	18.91%
	512 blocking	16.68 (0.24)	7.78 (0.24)	46.63%	3.84 (0.16)	23.01%	10.11 (0.05)	1.87 (0.04)	18.47%
	512 non-bl. (no ovlp)	18.05 (0.33)	6.30 (0.29)	34.89%	4.05 (0.34)	22.41%	10.18 (0.05)	1.91 (0.04)	18.74%
	512 non-bl. (overlap)	18.53 (0.22)	5.42 (0.24)	29.23%	4.15 (0.33)	22.39%	10.14 (0.04)	1.93 (0.04)	19.01%
	1024 blocking	21.74 (0.36)	12.73 (0.35)	58.55%	4.39 (0.10)	20.19%	12.91 (0.02)	2.18 (0.06)	16.92%
	1024 non-bl. (no ovlp)	22.89 (0.23)	10.99 (0.25)	48.01%	4.85 (0.25)	21.17%	12.94 (0.02)	2.26 (0.04)	17.44%
	1024 non-bl. (overlap)	22.99 (0.32)	6.71 (0.28)	29.21%	5.54 (0.87)	24.08%	12.82 (0.03)	2.19 (0.08)	17.09%
Cray MPICH	2048 blocking	41.69 (1.06)	29.88 (1.28)	71.68%	5.94 (0.31)	14.25%	19.64 (0.33)	3.58 (0.24)	18.22%
	2048 non-bl. (no ovlp)	40.87 (0.73)	25.15 (0.92)	61.53%	6.78 (0.63)	16.60%	19.50 (0.18)	3.65 (0.16)	18.70%
	2048 non-bl. (overlap)	45.07 (1.12)	8.34 (0.41)	18.51%	7.47 (0.87)	16.57%	17.65 (0.08)	2.84 (0.23)	16.06%
	4096 blocking	89.14 (4.65)	75.76 (4.55)	84.99%	5.07 (0.39)	5.69%	25.95 (0.28)	3.61 (0.16)	13.90%
	4096 non-bl. (no ovlp)	85.72 (4.28)	67.67 (4.29)	78.94%	4.82 (0.58)	5.62%	25.92 (0.21)	3.48 (0.21)	13.42%
	4096 non-bl. (overlap)	85.33 (3.66)	7.67 (0.20)	8.99%	5.37 (1.36)	6.29%	22.35 (0.09)	2.23 (0.29)	9.96%
	512 blocking	24.78 (0.33)	6.32 (0.26)	25.52%	3.23 (0.06)	13.02%	18.10 (0.40)	1.50 (0.07)	8.27%
	512 non-bl. (no ovlp)	24.25 (0.25)	3.32 (0.19)	13.69%	3.06 (0.05)	12.62%	17.94 (0.08)	1.34 (0.06)	7.49%
	512 non-bl. (overlap)	23.64 (0.42)	2.15 (0.33)	9.10%	3.22 (0.08)	13.63%	18.06 (0.09)	1.55 (0.07)	8.60%
Cray MPICH	1024 blocking	28.40 (0.43)	10.07 (0.40)	35.46%	3.59 (0.06)	12.65%	21.09 (0.09)	1.74 (0.07)	8.25%
	1024 non-bl. (no ovlp)	28.31 (0.52)	6.94 (0.48)	24.50%	3.58 (0.05)	12.64%	21.20 (0.09)	1.66 (0.06)	7.82%
	1024 non-bl. (overlap)	26.77 (0.60)	1.73 (0.48)	6.45%	4.64 (0.07)	17.35%	22.07 (0.11)	2.79 (0.08)	12.64%
	2048 blocking	49.87 (1.30)	30.47 (1.22)	61.10%	5.45 (0.09)	10.93%	26.37 (0.16)	2.83 (0.12)	10.72%
	2048 non-bl. (no ovlp)	50.36 (1.36)	27.74 (1.31)	55.09%	5.62 (0.04)	11.16%	26.62 (0.10)	2.92 (0.06)	10.99%
	2048 non-bl. (overlap)	44.53 (0.85)	2.87 (0.94)	6.44%	11.42 (0.26)	25.64%	32.46 (0.34)	9.10 (0.29)	28.03%
	4096 blocking	120.36 (4.03)	100.95 (3.91)	83.88%	6.77 (0.11)	5.63%	37.91 (0.71)	5.84 (0.40)	15.41%
	4096 non-bl. (no ovlp)	118.97 (3.24)	96.52 (3.11)	81.13%	6.90 (0.14)	5.80%	37.67 (0.52)	5.59 (0.34)	14.85%
	4096 non-bl. (overlap)	99.44 (1.49)	8.67 (0.51)	8.72%	28.79 (0.79)	28.96%	58.06 (0.88)	27.58 (0.87)	47.49%

Table 5: Scalasca wait-state metrics for the sandpile benchmark (allreduce variant) on JUWELS (OpenMPI and ParaStationMPI) and ARCHER2 (Cray MPICH), 10 runs per configuration, 32 processes. Values are means with standard deviation in parentheses. Percentages for Wait NxN and NxN Completion are relative to the total collective completion call time; Late Sender percentage is relative to MPI_Sendrecv time.