

An Event Model for Trace-Based Performance Analysis of MPI Partitioned Point-to-Point Communication

Isabel Thärigen
IT Center, RWTH Aachen University
Aachen, Germany
thaerigen@itc.rwth-aachen.de

Marc-André Hermanns
IT Center, RWTH Aachen University
Aachen, Germany
hermanns@itc.rwth-aachen.de

Markus Geimer
Jülich Supercomputing Centre,
Forschungszentrum Jülich GmbH
Jülich, Germany
m.geimer@fz-juelich.de

ABSTRACT

The MPI 4.0 standard introduced the concept of partitioned point-to-point communication. One facet that may help in encouraging application developers to use this new concept in their programs is the availability of proper tool support in a timely manner. In this paper, we therefore propose nine new events extending the OTF2 event model to accurately represent the runtime behavior of partitioned point-to-point communication. We then demonstrate the suitability of these extensions with three different use cases in the context of performance analysis. In particular, we showcase a prototype implementation of an extended waitstate analysis in the Scalasca trace analyzer, and discuss further potential use cases in the realm of trace visualization and simulation.

CCS CONCEPTS

• **General and reference** → **Performance; Measurement; Software and its engineering** → *Software maintenance tools*; • **Computing methodologies** → Simulation environments.

KEYWORDS

MPI, performance analysis, partitioned communication

ACM Reference Format:

Isabel Thärigen, Marc-André Hermanns, and Markus Geimer. 2023. An Event Model for Trace-Based Performance Analysis of MPI Partitioned Point-to-Point Communication. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023)*, November 12–17, 2023, Denver, CO, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3624062.3624205>

1 INTRODUCTION

In high-performance computing (HPC), the de-facto standard for writing parallel scientific applications running on distributed memory supercomputers is the Message Passing Interface (MPI) [5]. Since its initial publication in 1994 introducing point-to-point and collective communication alongside basic abstractions and management functions, many new functionalities have been added with every major revision of the standard. Well known examples of such extensions are parallel I/O in MPI 2.0, non-blocking collective communication in MPI 3.0, and large count arguments in MPI 4.0.

However, while a reference implementation for each new feature should exist before it is voted into the standard—thus making the concepts available to both vendors of MPI libraries and the HPC user community in a timely fashion—adoption of new MPI functionalities by application developers is often lagging behind.

The reasons for such a lack of adoption can be manifold. First, domain scientists and HPC application developers are often reluctant to introduce disruptive changes into their code bases, as this requires to revalidate the correctness of previous simulation results. And even if a code team would be willing to go through this process again, they might prefer not to touch legacy portions of the code that are less well-understood since the original developer has left the team. Second, it usually takes a while for MPI library implementations to optimize new functionality, such that early adopters may see no benefit—or worse, even a severe degradation—from using it, thus reverting back to their previous production code and thenceforth considering this functionality as unsuitable for their purposes. Finally, application developers may also be discouraged by the lack of proper tool support for new MPI features, which can become an important factor in using them efficiently in their codes. In this paper, we specifically focus on this tool-support aspect for a novel communication mechanism introduced with MPI 4.0, namely partitioned point-to-point communication (for brevity, in the remainder generally referred to as partitioned communication).

Partitioned communication is an extension of persistent point-to-point communication, which is an established concept in MPI communication. As such, it enables the programmer to set up a non-blocking communication port for a fixed argument list (i.e., rank, communicator, tag, buffer) once in advance, and then reuse the request handle returned by the setup call many times, for instance, for repeated data transfers within an iteration loop. Partitioned communication additionally allows to split the send/receive buffers into equal-sized chunks and to notify the readiness or query the successful arrival of each buffer chunk individually. The number of partitions—and with it the individual partition size—may vary between sender and receiver, as long as the overall message size is the same on both for a particular message. The partitioning of the communication buffers enables more fine-grained dependencies between code that manipulates or uses their contents. It can also reduce messaging overheads in applications, as parts of the communication may already be handled before the whole communication buffer is assembled. While this enables more flexibility on parts of the application and MPI library, it also poses new challenges on MPI-focused programming tools—in particular those tracking dependencies between processes that may lead to waiting time during the execution.

SC-W 2023, November 12–17, 2023, Denver, CO, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023)*, November 12–17, 2023, Denver, CO, USA, <https://doi.org/10.1145/3624062.3624205>.

In this paper, we examine how MPI partitioned point-to-point communication can be modeled in event traces on an abstract level, and present an event model that is suitable for various use cases in the context of performance analysis. In particular, the main contributions of this paper are:

- We introduce an *event model extension* for the Open Trace Format 2 (OTF2) [4] to accurately represent the runtime behavior of MPI partitioned point-to-point communication.
- We outline our *prototypical implementation* of an enhanced wait-state detection in the Scalasca trace analyzer [6, 18] based on this enhanced event model.
- We provide a *theoretical discussion* of how our event model extension could benefit further use cases, showcasing its suitability for various usage scenarios in the context of performance analysis.

The remainder of this paper is structured as follows: Section 2 provides the necessary background information that forms the basis of our work. Section 3 then introduces our event model extensions in detail, before we demonstrate how this event data can be used in different use-case scenarios in Section 4. Next, we review related work in Section 5. Finally, Section 6 concludes this paper and provides an outlook on future work.

2 BACKGROUND

In this section, we briefly introduce the necessary background information that provides the context of our work, in particular, the key concepts of partitioned point-to-point communication, trace-based performance analysis, and the current OTF2 event model.

2.1 MPI Partitioned Point-to-Point Communication

As part of the MPI 4.0 standard [5], the concept of partitioned point-to-point communication has been introduced. Based on a proposal of Grant et al. [7, 8], it was adopted by the MPI standard to particularly support multithreaded and task-based programming in MPI, though it can also be beneficial in other scenarios. Unlike non-partitioned communication calls that always operate on the entire memory buffers provided, the key idea of partitioned communication is to allow a more fine-grained control by subdividing buffers into equal-length partitions. For each partition, the user can then individually specify when it is ready to be sent or test whether it has been received, respectively. In the context of multithreading or tasking, separate threads/tasks could then be responsible for preparing one or more send partitions of the buffer. The MPI library can then internally decide to send any readily prepared partition without having to wait for all partitions (i.e., the full buffer) to be ready. Likewise, on the receiver side, any successfully arrived partition can already be processed once it is available, with different threads/tasks being responsible for separate partitions.

MPI partitioned point-to-point communication is defined as an extension of persistent point-to-point communication. As such, a rank taking part in partitioned communication only specifies the communication parameters once in a dedicated initialization call (i.e., `MPI_Psend_init` and `MPI_Precv_init`, respectively). In addition to the send/receive buffer and message envelope parameters also used in non-partitioned communication (i.e., source/destination

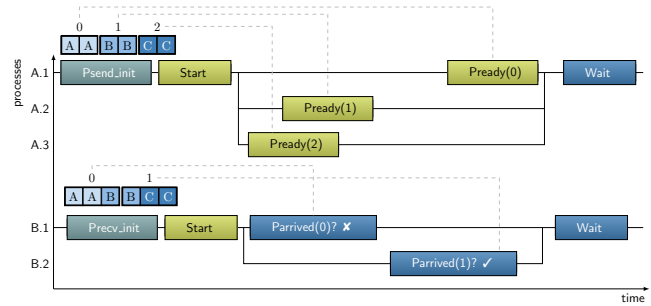


Figure 1: Timeline depicting a single partitioned point-to-point communication operation. Only relevant MPI functions are shown. Corresponding partitions are marked in parentheses where applicable.

rank, tag, and communicator), the caller also has to specify the number of send and receive partitions, respectively. These partition counts are explicitly allowed to differ on the sender and receiver side for the same message exchange, though the overall message size has to match. An `MPI_Precv_init` call only matches with an `MPI_Psend_init` using the same message envelope; wildcard receives are not allowed. For multiple initialization calls using the same envelope, their execution order defines the order in which the messages will eventually match.

As with other persistent communication, the overall send and receive operations for a partitioned communication between two ranks with matching init calls are initiated via either `MPI_Start` or `MPI_Startall`. Once a send operation has been started, the sending rank has to invoke `MPI_Pready` exactly once for each send partition when it is ready to be sent. There are also convenience functions to notify the readiness of a consecutive range or an arbitrary list of partitions, respectively. On the receiving rank, each partition can be tested individually for completion via `MPI_Parrived`, returning a boolean flag. If this flag indicates success, the corresponding partition of the buffer can already be used by the receiving rank, even if other partitions of the same buffer have not arrived yet. The earliest point an `MPI_Parrived` call can be observed to return successfully is when all `MPI_Pready` calls that it “depends on” have started. We say that an `MPI_Parrived` call depends on an `MPI_Pready` call if at least one buffer element is part of both partitions. Both send and receive operations always have to be completed with a call to `MPI_Wait`, a successful `MPI_Test`, or any of their variants, even if `MPI_Parrived` returned successfully for every partition. A user is not obliged to test for reception of all partitions, and any pending partitions will be considered arrived after successful completion of the receive operation. Note that the examples and discussions in the remainder of this paper only use `MPI_Start`, `MPI_Wait`, `MPI_Pready`, and `MPI_Parrived`, but equally apply to all other functions of the same category.

Figure 1 shows a timeline diagram of an exemplary call sequence for a single partitioned data exchange, using three send and two receive partitions for a buffer of six elements. Time progresses from left to right. The y-axis shows timelines for two ranks A and B, with three and two threads, respectively. Each colored box corresponds to one MPI function call; executions of computational user

functions in between are not shown for clarity. Note that the width of each box does not necessarily correspond to the actual duration spent in the call. Due to the different number of send and receive partitions, there is no exact match between send and receive partitions and the receive partitions depend on two send partitions each. Therefore, the MPI_Parrived call for receive partition 0 cannot return successfully, since it depends on the MPI_Pready calls for send partitions 0 and 1, from which the former starts only after the MPI_Parrived call ended. In contrast, the MPI_Parrived call for receive partition 1 can return successfully, because it depends on the MPI_Pready calls for send partitions 1 and 2, which both start early enough.

2.2 Trace-Based Performance Analysis

With the growing complexity of both HPC systems and distributed programs, performance analysis is getting more and more important to obtain efficient and scalable codes. To assist application developers with this task, a large variety of tools focussing on different performance aspects exists. Such tools make use of various techniques to obtain, aggregate, and evaluate performance data. In this work, we specifically focus on trace-based post-mortem analysis tools. This category of tools performs their analysis after the actual execution of the application has finished, based on event trace data collected during runtime. In particular, our work targets performance analysis tools based on the OTF2 trace format [4] as produced by Score-P [12, 16].

Score-P is a community-maintained instrumentation and measurement infrastructure to collect performance data for a number of different parallel application analysis tools, namely Scalasca [6, 18], Vampir [11], and TAU [15]. It supports the most prevalent parallel programming APIs like MPI, OpenMP, CUDA, and others. Depending on the API, it uses different techniques to collect the relevant performance data. For MPI, Score-P relies on a pre-instrumented (PMPI) wrapper library that allows to capture all communication details, including function call arguments. User code regions can be measured either via instrumentation or sampling. As output, Score-P can produce both aggregated runtime summary information in profiles as well as detailed event traces.

An *event trace* provides a record of the dynamic application execution behavior over time. It consists of a chronologically ordered sequence of *events*, which model changes in the execution state that are relevant for the intended purpose, in our context performance analysis. Examples of common events are sending or receiving a message, or entering or leaving a specific *code region*. Code regions intuitively model function calls, but may also refer to smaller code fragments or entire program phases marked manually by the user. Each event stores its type and a timestamp, as well as additional event-specific attributes such as the number of bytes transferred by a send operation. To avoid storing redundant information with individual events, a trace may also include *definition records*. Definitions store time-independent, static information that can then be referenced by many events using a unique identifier. The concrete definitions and events recorded to model the function calls and communication patterns of an application, their semantics, and interrelations are defined by the trace format and its underlying *event model*. In the following subsection, we therefore outline those

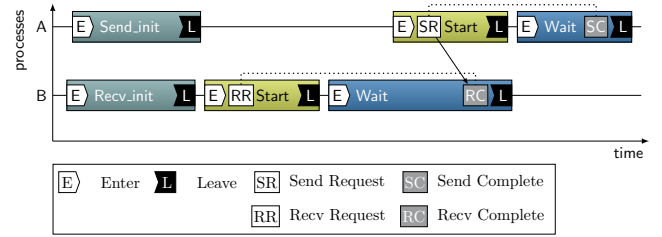


Figure 2: Event model for persistent (non-blocking) communication. Arrows depict communication relations between sender and receiver, whereas dotted lines indicate events belonging to the same communication operation.

parts of the current OTF2 event model that are relevant for this paper.

2.3 OTF2 Event Model

The OTF2 trace format [4] uses both definition and event records to store the trace data. Common definitions are, for example, code regions specified by source file name and begin/end line numbers, or communicators and their associated groups. Definitions are tracked per process, unified across all processes at the end of measurement, and stored in a single *global definitions file*, with mapping tables from process-local to global identifiers stored in additional *local definition files*. Events are collected per execution instance called *location* (i.e., separately for each CPU thread or GPU stream), and stored in so-called *local trace files*. In the following, we will only describe the OTF2 events used to model non-blocking point-to-point communication, which are also used for persistent point-to-point communication. Further event types of OTF2 that are unrelated in this context are not shown or discussed for brevity.

Figure 2 shows a timeline diagram of a simple example using a single persistent non-blocking message transfer between two ranks, including the corresponding events as defined by the current OTF2 event model. First, the execution of a code region such as an MPI function call is represented by an Enter/Leave event pair, recorded at the start and end of the corresponding code fragment, respectively. These two events only carry a timestamp, as well as a reference to the definition record storing the details about the code region. Non-blocking communication is modeled with the help of four additional events in between, one in each of the MPI_Start calls and each completion call (MPI_Wait, successful MPI_Test, etc.).

On the sender side (i.e., on process A), a Send Request event is recorded directly after the Enter event of the function initiating the data transfer, here the MPI_Start call that triggers the transfer set up by a previous invocation of MPI_Send_init. Apart from the timestamp, this Send Request event also stores the corresponding send parameters, such as the destination rank, the tag, a handle to the communicator definition, and the number of bytes transferred. In addition, it also stores a locally unique identifier corresponding to the MPI request handle. The second event that is recorded for a non-blocking send operation is a Send Complete event at the end of the completion call, here MPI_Wait. This event only stores the timestamp and the respective request identifier, which allows to

match all events processing the same MPI request and therefore belonging to the same point-to-point operation.

On the receiver side (i.e., on process B), similar `Recv Request` and `Recv Complete` events are recorded in the equivalent function calls of the receive operation. In contrast to the corresponding send events, however, the receive parameters (i.e., source rank, tag, communicator, and bytes received) are stored in the `Recv Complete` event, as the initiating call might have used wildcards for the tag and/or source rank, and the final matching of messages is therefore only known in the completion call.

For both non-blocking sends and receives, `Request Tested` events can be recorded for unsuccessful calls to `MPI_Test` (not shown in the example in Figure 2). These events are optional and only carry a timestamp as well as the corresponding request identifier.

3 EXTENDED EVENT MODEL

The existing set of OTF2 events for non-blocking point-to-point communication described in the previous section are not sufficient to accurately model the fine-grained dependencies between send and receive partitions in MPI partitioned communication. In particular, the relaxed semantics of partially completed operations can not be represented. In this section, we therefore present our proposed extensions to the OTF2 event model to address this limitation. First, we briefly describe the desired properties of our event model extension, before delving into its details, design decisions, and considered alternatives.

3.1 Desired Properties

First, the extensions to the event model should make it easy to understand the underlying communication patterns when analyzing the event trace data. Furthermore, information regarding partitioned communication should be stored in specific events instead of attaching additional attributes to existing event records. This allows analysis tools to easily recognize and distinguish partitioned communication from other non-blocking communications.

Second, the model should be complete, ideally supporting any kind of communication analysis. As it is unfeasible to anticipate all future types of analysis using this model, it therefore has to strike a good balance between mandatory and optional elements. Thus, we focus on capturing the key aspects of partitioned communication that still cover a wide range of use cases.

Third, we aim for a compact modeling of partitioned point-to-point communication, reducing redundancy as much as possible. In certain use cases, however, it might be beneficial to (optionally) allow for some redundant data.

Finally, the extended event model should be conceptionally consistent with the existing event model in OTF2, as much as this is possible.

3.2 Proposed Events

Our extended event model covers the key aspects of the communication in order to support many different use cases. For point-to-point communication in general, this includes providing the relevant send/receive parameters, as well as sufficient data to allow for reconstructing the matching of messages. Moreover, it offers

information about the start and completion of the send and receive operations. For partitioned communication, the partial readiness of send operations also needs to be tracked, and partial and total completion of receive operations need to be distinguished.

In the following subsections, we describe our proposed new events and how they address these aspects, and discuss design alternatives that have been considered. Note that in all cases, these events are surrounded by the `Enter` and `Leave` events written at the start and end of the respective MPI partitioned communication function call. Table 1 gives an overview of all proposed events, their attributes, and location in the event stream. Figure 4 shows the same communication scenario as Figure 1 augmented by the respective events.

3.2.1 Partial readiness and completion. To model the readiness and successful reception of individual send/receive buffer partitions, we introduce additional events. Matching to the corresponding MPI calls where they are generated, we name them `Pready` (`Pr`) and `Parrived` (`Pa`), respectively. Besides the mandatory timestamp, these events need to carry the partition number as well as the request identifier. Both can be directly derived from the function call arguments, and allow to correctly correlate these events with the corresponding communication operation and partition. To indicate the earliest point in time a partition may have been marked ready by the MPI library, the `Pready` event is generated at the begin of an `MPI_Pready` call. Similarly, the `Parrived` event marks the time the completion of a partition is observed by the user application and is generated at the end of a successful `MPI_Parrived` call.

3.2.2 Unsuccessful partial completion. Unsuccessful invocations of `MPI_Parrived` are not directly relevant for the completion semantics or message matching, yet may provide additional information about application behavior. When called very often (e.g., when using a polling scheme), such calls may produce a significant amount of event data, and thus produce unnecessary clutter in a trace file and enlarge it without much benefit for many use cases. However, for some use cases, such as for simulations, it might be useful to record such unsuccessful calls. Therefore, an optional `Ptested` (`Pt`) event is part of our extended event model, similar to the existing `Request Tested` event for persistent and other non-blocking communication. Besides the mandatory timestamp, this `Ptested` event includes the tested partition number as well as the request identifier.

3.2.3 Transfer start and (full) completion. As with persistent communication, the overall start and completion of a message transfer is tracked with separate events. In the existing OTF2 event model, this is achieved by the respective `Request` and `Complete` events for both send and receive operations, which also store the corresponding send and receive parameters. On the receiver side this is necessary because the possibility of using wildcard receives means that the receive parameters might change with each transfer. For symmetry reasons, and since these events are also used to model non-persistent non-blocking communication, the send parameters are also stored at the `Send Request` event.

As partitioned communication disallows the use of wildcard receives, send and receive parameters will not change across multiple transfers and are known in advance. To avoid redundancy

by repeatedly storing the same data for every repetition of the message transfer, it is sufficient to track them only once. A suitable set of events to capture this information is introduced in the next subsection, so the respective Request and Complete events for partitioned communication only need to store the request ID besides their timestamp. Therefore we opted to introduce a new set of events to capture the start and completion of partitioned transfers. These events are named in a similar way as their non-partitioned counterparts, that is, Psend Request (`PSR`), Psend Complete (`PSC`), Precv Request (`PRR`), and Precv Complete (`PRC`). Like with the existing events, the Request events are written at the begin of the respective MPI function call, and the Complete events at the end of the completion call (i.e., `MPI_Wait`, successful `MPI_Test`, etc.).

One could argue that on the sender side, the actual transfer does not start until the first `MPI_Pready` call—which is already represented by an event—and that the Psend Request event is therefore unnecessary. However, there is no such correlation on the receiver side, where the `MPI_Parrived` calls are neither required nor indicating the actual start of the receive operation. Thus, eliminating the Psend Request event would lead to asymmetrical event sequences and an inconsistency with the existing event model for non-partitioned communication. Furthermore, being able to identify the temporal distance between request generation and the first Pready event may be relevant for some performance tools.

3.2.4 Transfer initialization. So far our event model extensions provide information regarding the start and completion of the send and receive operations, both for partial and total semantics. While the data stored in the corresponding Request and Complete events for persistent and “regular” non-blocking communication is sufficient to reconstruct the correct message matching, this is not the case for partitioned point-to-point communication. According to the MPI standard, non-partitioned communications match based on the message envelope. In the corner case of identical envelopes, message transfers match based on the order in which they were initiated (i.e., the event or timestamps order). However, partitioned communications match in this case based on the order of their *initialization*.

Figure 3 shows an example where a matching of the send and receive operations does not coincide with a matching based on initialization. Here, non-partitioned persistent communication would match the `MPI_Start` of send operation 2 with the `MPI_Start` of receive operation 1 and vice versa (indicated by the dashed red lines). However, the initialization calls dictate a matching of send operation 1 with receive operation 1, and send operation 2 with receive operation 2 (solid green lines).

As a result, two more events generated by the initialization calls are required, which we call Psend Init (`PSI`) and Precv Init (`PRI`). As mentioned above, wildcard receives are not allowed for partitioned communication, so these two events also give us the opportunity to reduce redundancy by including the send and receive parameters in those events rather than in the Psend Request and Precv Complete events. In summary, the Psend Init and Precv Init events carry a timestamp, the destination/source rank, tag, communicator, number of bytes sent/received, number of partitions, as well as the request identifier.

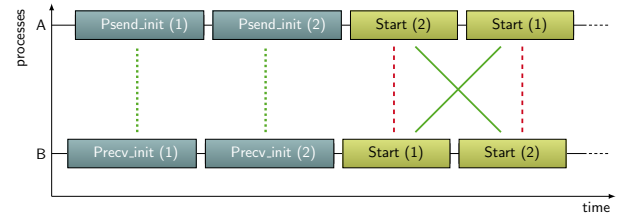


Figure 3: Different matching depending on whether the matching is done according to the initialization calls or the start calls. Green (solid and dotted) indicates the matching for partitioned communication, red (dashed) indicates the matching according to non-partitioned persistent communication.

4 USE CASES

In this section, we demonstrate how to use the proposed event model extensions for partitioned communication in three different scenarios in the context of performance analysis. To this end, we have implemented support for reading and writing our new event types in the OTF2 library, and extended the Score-P measurement system to generate those events from its PMPI wrappers of the corresponding MPI routines as described before. In the following, we present details of our prototypical implementation of an enhanced wait-state search for partitioned communication in the Scalasca trace analyzer, and outline—on a conceptual level—how trace visualization and simulation can make use of the new events.

4.1 Automated Trace Analysis

The Scalasca Trace Tools [6, 18] are a collection of trace-based post-mortem performance analysis tools supporting OTF2 trace files as generated by Score-P as their main input format. The core component—the Scalasca trace analyzer—is able to identify wait states in communication and synchronization operations, as well as communication or load imbalances as their root causes. The analysis specifically targets large-scale parallel applications and is based on a so-called *parallel replay*. That is, the analyzer is started using the same number of processes and threads as the target application, with each analysis process/thread processing the corresponding local trace file. The event streams are traversed in parallel in timestamp order. Upon encountering specific communication events, information required for the analysis is exchanged along the same communication paths recorded for the application, using operations of similar type (e.g., point-to-point communication for a point-to-point message transfer). Using the same parameters as in the original execution ensures that the replay operations also match in the same way.

Since the Scalasca analyzer keeps the entire trace data in memory, the event streams can be processed in both forward and backward direction. In the latter case, the roles of senders and receivers are reversed. In total, the analysis algorithm performs five such traversals in alternating directions. In the following we only focus on the detection of *Late Sender* wait states for partitioned communication, which is performed during the first forward replay. Covering the later replay stages is left for future work.

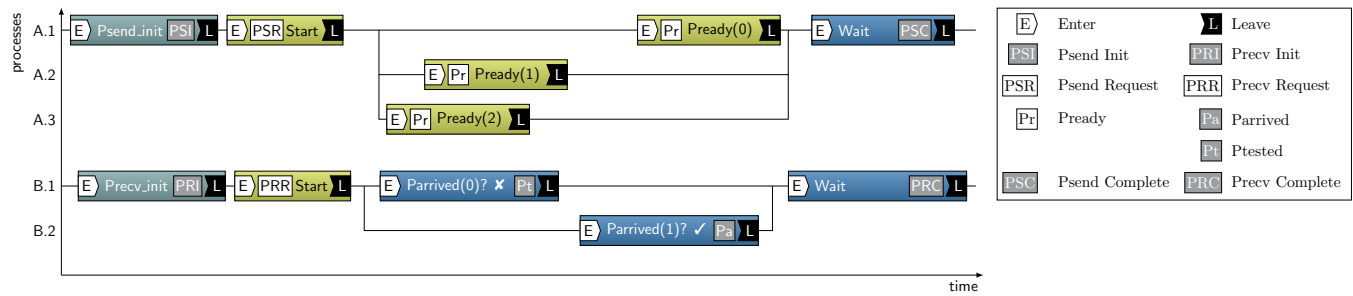


Figure 4: Extended event model for a partitioned communication example. Each MPI call writes an Enter and a Leave event. For the start of a partitioned send or receive, a Psend Request or Precv Request event is written, respectively. In the completion call, a Psend Complete, or Precv Complete event is written, respectively. Each MPI_Pready call introduces a corresponding Pready event. MPI_Parrived calls either write a Parrived event if they are successful, or a Ptested event otherwise. A Psend Init and Precv Init event is written in the corresponding initialization calls.

Event	Function	Position	Attributes
PsendInit	MPI_Psend_init	End	dest, comm, tag, bytes sent, #send partitions, reqID
PrecvInit	MPI_Precv_init	End	src, comm, tag, bytes recvd, #recv partitions, reqID
Pready	MPI_Pready	Begin	partition, reqID
Parrived	Successful MPI_Parrived	End	partition, reqID
Ptested	Unsuccessful MPI_Parrived	End	partition, reqID
PSendRequest	MPI_Start	Begin	reqID
PRecvRequest	MPI_Start	Begin	reqID
PSendComplete	MPI_Wait/successful MPI_Test	End	reqID
PRecvComplete	MPI_Wait/successful MPI_Test	End	reqID

Table 1: Overview of the proposed new events, their position in the event stream, and attributes.

4.1.1 Waiting Time Definition. Waiting time in MPI operations occurs when one rank arrives late at a common synchronization point, such as a message transfer, causing other ranks to idle while waiting for the latecomer. For point-to-point communication, the Scalasca analyzer detects two types of wait states: *Late Sender* and *Late Receiver*. In the *Late Sender* case, a send operation starts later than the corresponding receive operation, causing the receiving rank to sit idle in a blocking call (e.g., MPI_Recv or MPI_Wait in case of non-blocking communication). The waiting time then corresponds to the timespan between entering the blocking call on the receiver side and the earliest point at which it could have completed, that is, the start of the corresponding send operation. Figure 5 shows an example of Late Sender waiting time in persistent communication, indicated by the red part of the MPI_Wait call. *Late Receiver* wait states are defined analogously, where a receive operation is started too late, causing the sending rank to stall (e.g., due to the use of a rendezvous protocol for large messages). Here, the waiting time is defined as the difference between the begin of the call in which the send operation is completed and the start of the receive operation, as the latter is the first point at which the send completion can take place.

In partitioned point-to-point communication, the earliest point at which the whole buffer can be received is when the last MPI_Pready call is started. Therefore, we define Late Sender waiting time as the time between the start of the MPI_Wait call of the receive operation and the start of the last MPI_Pready call. A timeline visualization

of this situation is shown in Figure 6. Late Receiver wait states in partitioned communication are defined in exactly the same way as for persistent communication, as the earliest point at which the send operation can complete is when the receive operation is started. The events of the proposed extension include all the information that is also present in the events for persistent communication. As a result, the detection of Late Receiver wait states in partitioned communication is trivial and only Late Sender waiting time will be considered in the following.

4.1.2 Waiting Time Detection. For non-partitioned persistent point-to-point communication, the parallel replay in Scalasca’s trace analyzer detects Late Sender waiting time in the MPI_Wait call of the receive operation. For this it needs the Enter event timestamp of the sender’s MPI_Start call, as well as the Enter timestamp of its own MPI_Wait region. As each analysis process/thread can only access its own process- or thread-local trace, respectively, it can access the Enter timestamp of the MPI_Wait call directly, but has to receive the sender’s Enter event timestamp via MPI communication. This communication is triggered when encountering a Send Request event by calling MPI_Isend with the same parameters as the original persistent communication, sending the corresponding Enter timestamp. The completion of the send operation is ensured later on. Similarly, if a replay process encounters a Recv Complete event, a message is received using the same source rank, communicator, and

tag as in the original communication. Once completed, both timestamps are available at the receiving process and the waiting time can be calculated as their difference. This replay communication is visualized with (turquoise) lines in Figure 5.

The basic idea of our analysis for partitioned communication is visualized by the (turquoise) lines and annotations in Figure 6. First, we need to ensure a proper matching of messages. As mentioned in Section 3, partitioned communications with identical envelope match according to the order of initialization calls and not their start calls. Thus, a replay as outlined above might match messages in the wrong order. As a result, we decided to also use partitioned communication for the replay, though with only a single partition. That is, to ensure the correct matching, we replay an MPI_Psend_init and MPI_Precv_init call when encountering the Psend Init and Precv Init events, respectively, keeping the source/destination rank, tag, and communicator the same as in the original communication.

Second, the transfer of the analysis data has to be initiated. This is achieved by calling MPI_Start on encountering a Psend Request event. To calculate the Late Sender waiting time in partitioned point-to-point communication, the Enter timestamp of the last MPI_Pready event is required. However, this information is not readily available at the Psend Request event during a forward replay of the trace, and might also not be easily accessible as the call might happen on another thread. Thus, the replay of the send operation has to be deferred to a later point in time. In our current implementation, a replay process/thread encountering a Pready event determines the corresponding Enter event timestamp and stores it in a shared, lock-protected data structure. Afterwards, we check if this data structure already contains Enter timestamps for all other Pready events belonging to this communication. If this is the case, the maximum is calculated, stored in the send buffer, and MPI_Pready is called for the single partition of the corresponding request.

Note that it is not possible to execute these steps in the Psend Complete event, since this would delay the start of a non-blocking communication too far—potentially after a different blocking call, which might cause a deadlock during the replay. Also, the maximum timestamp is not necessarily the timestamp of the Pready event triggering the readiness of the send buffer, since a multithreaded replay might encounter the Pready events in a different time order than they were written. Moreover, it might be more efficient to directly track the maximum timestamp in a shared variable, but we leave such optimizations of our prototype implementation as future work.

Third, the data transfer has to be completed. For persistent communication, send operations are regularly tested for completion. However, for partitioned communication we have to ensure that a send operation is completed before starting the next, so this approach does not work. Unfortunately, replaying an MPI_Wait when encountering the Psend Complete event might also lead to a deadlock. As this event could have been generated by an MPI_Test call completing a message transfer sent in eager mode early, finalizing the send operation with a potentially blocking MPI_Wait during the replay may unintentionally stall the execution and prevent the sending of another message that is expected to be received by the receiver before the Precv Request event. Instead we replay

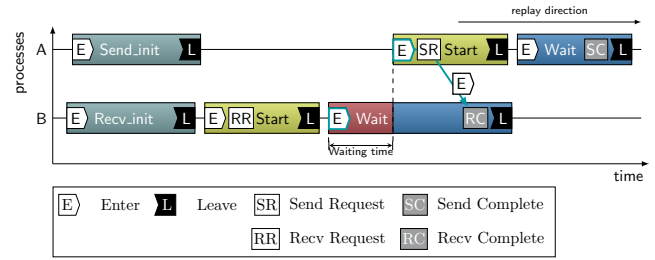


Figure 5: Late Sender waiting time in persistent point-to-point communication. The detection is visualized via (turquoise) arrows.

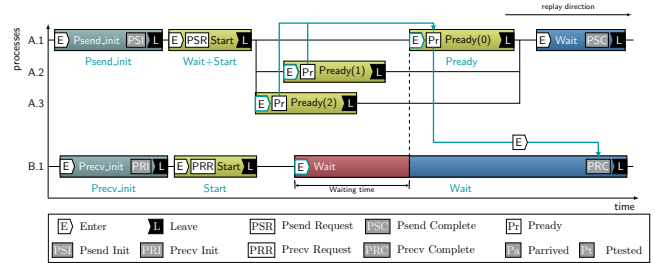


Figure 6: Late Sender waiting time in partitioned point-to-point communication. The events communicated during the wait-state detection and their communication paths are visualized via (turquoise) arrows. Descriptions below regions indicate the MPI calls used during the replay.

the MPI_Wait call in the next Psend Request event, right before the call to MPI_Start as described above. At this point we know that the original application has already ensured completion of the previous communication, so it is safe to complete the operation without risking a deadlock. And since MPI_Wait calls on inactive requests complete directly (i.e., are essentially no-ops), this is also not an issue for the very first Psend Request event encountered. This leaves the completion of the final communication operation, which we perform at the end of the replay.

Finally, the replay on the receiver’s side is much simpler. Here, we do not have to collect any timestamps from the trace beforehand. We can therefore replay the MPI_Precv_init, MPI_Start, and MPI_Wait calls while processing the Precv Init, Precv Request, and Precv Complete events, respectively. Note that it is safe to replay the wait at the Precv Complete event without risking a deadlock since the receive operation always behaves synchronously. After the replayed MPI_Wait returns, we can obtain the wait Enter timestamp from our process- or thread-local trace and calculate the waiting time by subtracting it from the received maximum MPI_Pready Enter timestamp. If the result is greater than zero, we have detected a Late Sender wait state.

4.1.3 *Prototype Validation.* To verify the correctness of our prototypical implementation of an enhanced wait-state search, we used a simple test code with three different variants. Figure 7 visualizes

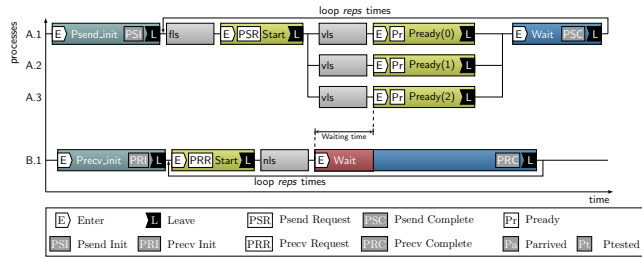


Figure 7: Test case for partitioned Late Sender waiting time. The length of the delays fls , vls , and nls determines the amount of waiting time that occurs. Only ever one of the delays appears in the execution, yielding three different code variants: NLS for nls , FLS for fls , and VLS for vls .

the basic idea. The test performs a simple partitioned point-to-point communication between two processes that is started and stopped $reps$ times, with a configurable number of send and receive partitions. By default, one partition is assigned to each thread, however, this can be changed via an overdecomposition factor d that determines the number of partitions per thread. The code can be executed in both single- and multithreaded mode. In the single-threaded case, the `MPI_Pready` calls are executed in ascending order of their partition.

The code regions fls , vls , and nls represent potential delays in the execution due to computation. In our test code, we mimic this delay by calling `sleep()`. We assume that exactly one of the delays appears, leading to three different code variants causing different Late Sender waiting times:

NLS is short for ‘No Late Sender’. Here, only code region nls appears, and all `MPI_Pready` calls start before the `MPI_Wait` call on the receiver side. Thus there is no Late Sender waiting time.

FLS is short for ‘Fixed Late Sender’. In this case, only code region fls appears. This leads to a Late Sender waiting time for each loop repetition that is approximately equal to the duration of fls .

VLS is short for ‘Varied Late Sender’, where only code region vls appears. For a multithreaded execution, the waiting time per repetition is then approximately equal to the duration of vls . In the single-threaded case, this waiting time scales with the number of send partitions.

Note that the widths of the regions depicted in Figure 7 are not proportional to the actual execution times, that is, in reality the durations of fls , vls , and nls are magnitudes larger than the time spent in the non-blocking MPI communication calls.

We ran three different measurement series for each of these code variants, in each varying one of the following three factors: number of send partitions, number of repetitions, and overdecomposition factor. Additionally, each measurement was run in both single- and multithreaded mode. By default (i.e., when the corresponding setting was not varied), the number of send partitions was set to 2, and the repetitions and overdecomposition factor was set to 1. All measurements were executed on the CLAIX-2018 cluster of RWTH Aachen University using GCC 11.3.0 and MPICH 4.2a1.

As expected, our analysis prototype did not report any Late Sender waiting time for the NLS measurements. For measurements of the FLS variant, we used a 1 second sleep for fls in the repetition scaling experiment, and 10 seconds in all others. All reported waiting times corresponded to the expected values of $reps$ and $10 \cdot reps$ seconds, respectively, with a deviation of less than 5%, in most cases even less than 1%. For the VLS variant, we fixed the duration of vls to 1 second when scaling the number of send partitions, and to 0.1 seconds in all others. Again, the reported waiting times scaled as expected with the number of send partitions.

4.2 Trace Visualization

Instead of automatically analyzing event traces, trace data can also be visualized to facilitate a manual analysis. A well-known tool in this category is Vampir [11], which also operates on OTF2 trace files. Vampir displays traces in a timeline view similar to the various diagrams used in this paper. Each location (e.g., process, thread) has its own timeline, with code regions depicted as colored rectangles. Communication is visualized by connecting the corresponding regions on the participating locations, for example, using a line from the start of a send operation to the end of a receive completion in case of point-to-point communication. The communication details such as the tag, communicator, and bytes transferred, can be retrieved by selecting such a message line. Besides the main timeline view, Vampir also provides a variety of additional specialized timeline and statistics views, allowing for an in-depth analysis of the execution behavior.

Obviously, such a timeline visualization is also desirable for partitioned point-to-point communication. However, the partitioned semantics make it less obvious from where to where the communication lines should be drawn. The next two subsections detail how such a visualization could look like, and how it can be derived from our extended event model.

4.2.1 Visualization. For persistent communication, the begin of the `MPI_Start` region on the sender side is connected to the end of the matching `MPI_Wait` (or successful `MPI_Test`) region on the receiver side. The number of bytes transferred corresponds to the size of the send buffer. A similar visualization would also be possible for partitioned communication, while taking into account that the matching is determined by the initialization calls. However, such a simplified visualization completely ignores the partitioned semantics, which might also be of interest to a user. Therefore, we propose an alternative sub-partition visualization that explicitly visualizes which `MPI_Pready` calls communicate with which completion calls (i.e., a successful `MPI_Parrived`, `MPI_Test`, or `MPI_Wait` call), indicating that a partition of the buffer can be reused.

A first approach might try to keep either the number of outgoing edges per `MPI_Pready` region or incoming edges per completion call less or equal to one. However, both approaches make it impossible to assign further information (e.g., the number of bytes transferred) in a sensible manner, as either the value per communication line or the sum over all communication lines for one partitioned exchange amounts to a wrong value. Instead, we propose to connect the completion call for each receive partition with the `MPI_Pready` call that it depends on. As explained in Section 2.1, a call depends on another if the corresponding partitions contain at least one buffer

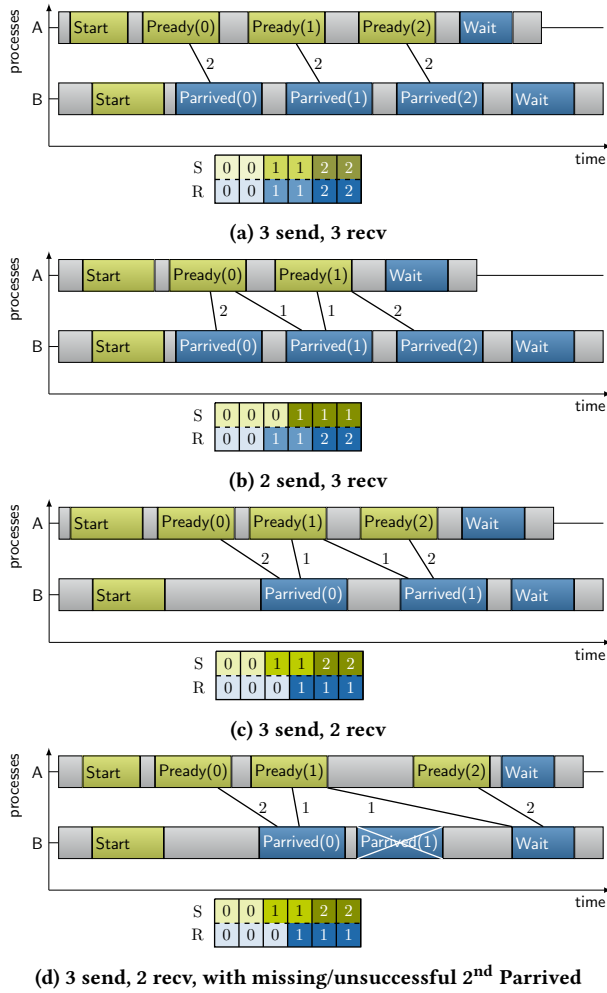


Figure 8: Proposed visualization for partitioned communication. The timeline views only depict successful `MPI_Parrived` calls, except for case (d), where the unsuccessful (or missing) `MPI_Parrived` call region is crossed out. The buffers for all examples have a size of 6 bytes and are depicted below each timeline. The send and receive partitioning of the buffers is visualized via different colors. Lines connect each `MPI_Pready` region and the corresponding receiving region(s). Numbers next to those lines correspond to the transferred message size in bytes assigned to the corresponding transfer.

element that is part of both partitions. If a completion call depends on multiple `MPI_Pready` calls, a line will be drawn for each of those. The number of transferred bytes for a connection corresponds to the number of overlapping buffer elements for the particular send and receive partition, times the size of the buffer data type in bytes. If a trace contains multiple successful `MPI_Parrived` calls for the same partition, the first one counts as the one completing the receive of this partition.

Figure 8 shows such a visualization for different cases of an example communication using a buffer of six byte-sized elements.

Unless noted otherwise, the depicted `Parrived` calls are successful. In Figure 8a, we have a matching number of send and receive partitions, that is, each receive partition is dependent on exactly one send partition. Thus, we get three connections, each transferring two bytes. In Figure 8b, we only have two send but three receive partitions. As a result, the second receive partition depends on both send partitions and is assigned two incoming connections with 1 byte each. Figure 8c shows the inverse case with three send and two receive partitions, so that the `MPI_Pready` call for partition 2 has two outgoing connections with 1 byte each. Finally, Figure 8d shows what happens when no successful `MPI_Parrived` occurs for a partition, either because a `MPI_Parrived` call is missing or returns unsuccessfully. Here, the second receive partition gets completed in the `MPI_Wait` call, so the corresponding connections will be drawn to this region instead.

4.2.2 Derivation From The Event Model. Now we show that it is feasible to derive the proposed visualization from our extended event model. For the simplified visualization that ignores the partitioned semantics, we need events in the start of each send and completion of each receive operation, as well as information about the matching. The latter can be determined from the `Psend Init` and `Prcv Init` events, which provide all communication parameters and the timestamps of the initialization calls. The actual connection can then be drawn between the `Psend Request` and `Prcv Complete` events, whose matching can be derived from the initialization calls using the same request identifier. The number of bytes transferred corresponds to the number of bytes sent, stored in the `Psend Init` event.

The proposed sub-partition visualization additionally needs information about the position of the `MPI_Pready` and completion calls, given by the `Pready`, `Parrived`, and/or `Prcv Complete` events. The matching can again be determined via the initialization calls with the same request identifier. To determine which send and receive partitions are dependent on each other, we only need the number of send and receive partitions, which are also available from the initialization events. For a send partition $i \in \{0, \dots, s-1\}$, the corresponding receive partitions $j_1, \dots, j_n \in \{0, \dots, r-1\}$ can then be calculated as:

$$J(i) = \{j_1, \dots, j_n\} = \left\{ \lfloor \frac{r}{s} \cdot i \rfloor, \lfloor \frac{r}{s} \cdot i \rfloor + 1, \dots, \lceil \frac{r}{s} \cdot (i+1) \rceil - 1 \right\}$$

Likewise, the send partitions $i_1, \dots, i_n \in \{0, \dots, s-1\}$ of a receive partition $j \in \{0, \dots, r-1\}$ can be calculated as:

$$I(j) = \{i_1, \dots, i_n\} = \left\{ \lfloor \frac{s}{r} \cdot j \rfloor, \lfloor \frac{s}{r} \cdot j \rfloor + 1, \dots, \lceil \frac{s}{r} \cdot (j+1) \rceil - 1 \right\}$$

To calculate the number of bytes transferred, we additionally need the total size of the buffer, available via the “bytes sent” attribute of the `Psend Init` event. In summary, our event model extensions provide all information needed to derive both the simplified and the sub-partition visualizations for a given event trace.

4.3 Simulation

In the field of trace-based performance simulation it is paramount for tools, such as the Scalasca trace simulator [9], to base the simulation on as accurate information as possible to obtain good simulation results. The Scalasca trace simulator, which is part of the Scalasca Trace Tools, uses the same replay-based infrastructure as

the Scalasca trace analyzer discussed in Section 4.1. Therefore, the necessary callbacks can be registered for the mandatory events of our proposed event model. However, the simulator also allows for manipulation of the execution times of specific regions as part of so-called *hypotheses*. Hypotheses may include function foo being faster by a factor of x or function bar having a better balance of execution time across multiple processes. Such hypothetical changes influence when the communication is taking place in the replay of the application behavior. To increase prediction accuracy, its simulation of general non-blocking communication [2] takes Request Tested events into account, which mark unsuccessful MPI_Test calls. Analogously, our proposed event model therefore contains the *optional* event Ptested to mark unsuccessful tests for the arrival of a certain partition. Retaining such information also for partitions of communication buffers can enable the simulator to better identify earlier partition completion potential by a receiving process and consider this in its simulation results. Here, the user will have to weigh for each application whether the additional information is worth its potential overhead. Note that an actual implementation in the Scalasca trace simulator using the proposed event model was out of scope for this paper and has to be left to future work.

5 RELATED WORK

Sampling performance profilers, such as HPCToolkit [1], do not specifically intercept communication calls, but rather provide a statistical overview of where time is spent in the application (and the libraries used, including MPI). Therefore specific state changes (events) are neither observed nor recorded. Thus, an event model as presented here is not applicable for their usage scenarios. Instrumenting performance profilers, such as provided by Score-P [4] and TAU [15], may observe the desired events, but aggregate such information at runtime. As such, the parts of the proposed event model specifically geared to identify cross-process dependencies are also not applicable for them.

Next to the trace-based tools discussed in Section 4, further tools exist that focus on HPC applications using MPI. Extrae, the measurement infrastructure for Paraver [13], does not use a specific, pre-defined event model such as OTF2, but rather defines trace records without fixed semantics. While this technique has also proven itself to be powerful in practice, the event model presented here could form the basis for performance analysts using Paraver to define such records for their analyses. Dimemas is a performance simulator based on Extrae event traces. As with Paraver, its underlying event set can be freely determined by the performance analyst and therefore has similar implications as Paraver. Intel Trace Analyzer and Collector (ITAC) [10] is a measurement and analysis framework similar to a combination of Score-P and Vampir and in parts Scalasca. Its main focus lies on identifying hot code regions and providing timeline visualizations of MPI process interaction. It works on event traces in the STF format, but offers capabilities to convert OTF2 traces to this format. ScalaTrace [14] provides scalable trace collection for MPI applications by using on-the-fly compression techniques during measurement reducing redundant information. While it also uses an event model for its internal representation, it no longer seems to be maintained. EZTrace [17] provides easy access to tracing capabilities in an HPC environment

and can be used in combination with the ViTE trace visualizer [3]. EZTrace intercepts MPI based on the PMPI interface and writes OTF2 trace files.

To the best of our knowledge, none of the abovementioned trace-based tools currently supports MPI partitioned point-to-point communication.

6 CONCLUSION AND OUTLOOK

As part of the MPI 4.0 standard, the concept of partitioned point-to-point communication has been introduced. Adding proper support for this concept to analysis tools can be considered as one cornerstone in encouraging developers to make use of this new feature. To this end, this paper introduced an extension to the OTF2 event model, encompassing nine new events to model partitioned communication. In addition, we demonstrated that these extensions are suitable to cover three different use cases. We described our prototypical implementation of an enhanced wait-state search in Scalasca, proposed a customized visualization scheme for trace visualizers such as Vampir, and showed how the new events can be handled in simulation tools.

As part of our future work, we plan to upstream our extensions to the various software packages, that is, OTF2, Score-P, and Scalasca. For the latter, we also need to investigate how to handle partitioned communication in the delay and critical-path analysis steps, and evaluate the outlined improvements for the wait-state detection algorithm. We also plan to examine better ways to deal with unsuccessful test calls, such as MPI_Test or MPI_Parrived, which may cause excessive tracing overheads in some cases. For example, the generation of all events related to such calls—including Enter and Leave—could be skipped when not needed for the intended use case. Instead, only the total number of unsuccessful tests could be recorded with the request completion event. This, in fact, applies to both partitioned and non-partitioned communications.

REFERENCES

- [1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701. <https://doi.org/10.1002/cpe.1553>
- [2] David Böhme, Marc-André Hermans, Markus Geimer, and Felix Wolf. 2010. Performance Simulation of Non-blocking Communication in Message-Passing Applications. In *Euro-Par 2009 – Parallel Processing Workshops*, Hai-Xiang Lin, Michael Alexander, Martti Forsell, Andreas Knüpfer, Radu Prodan, Leonel Sousa, and Achim Streit (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 208–217.
- [3] Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and François Trahay. 2012. An Open-Source Tool-Chain for Performance Analysis. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–48.
- [4] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang Nagel, and Felix Wolf. 2012. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing (Advances in Parallel Computing, Vol. 22)*. IOS Press, Amsterdam, 481 – 490. <https://doi.org/10.3233/978-1-61499-041-3-481>
- [5] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard*.
- [6] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. 2009. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Comput.* 35, 7 (July 2009), 375–388. <https://doi.org/10.1016/j.parco.2009.02.003>
- [7] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. 2019. Finepoints: Partitioned Multithreaded MPI Communication. In *High Performance Computing*, Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan (Eds.). Springer, Cham, 330–350.

- [8] Ryan E. Grant, Anthony Skjellum, and Purushotham V. Bangalore. 2015. Lightweight threading with MPI using Persistent Communications Semantics.
- [9] Marc-Andre Hermanns, Markus Geimer, Felix Wolf, and Brian J. N. Wylie. 2009. Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 78–84. <https://doi.org/10.1109/PDP.2009.50>
- [10] Intel Corp. 2012. Intel Trace Analyzer and Collector. <http://software.intel.com/en-us/intel-trace-analyzer>
- [11] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. 2008. The Vampir Performance Analysis Tool-Set. In *Tools for High Performance Computing*, Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 139–155. https://doi.org/10.1007/978-3-540-68564-7_9
- [12] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleyunik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91. https://doi.org/10.1007/978-3-642-31476-6_7
- [13] Jesus Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. 1996. DiP: A parallel program development environment. In *Euro-Par'96 Parallel Processing (LNCS, Vol. 1124)*, Luc Bougé, Pierre Fraignaud, Anne Mignotte, and Yves Robert (Eds.). Springer, Berlin, Heidelberg, 665–674. <https://doi.org/10.1007/BFb0024763>
- [14] Frank Mueller, Xing Wu, Martin Schulz, Bronis R. de Supinski, and Todd Gamblin. 2010. ScalaTrace: Tracing, Analysis and Modeling of HPC Codes at Scale. In *Applied Parallel and Scientific Computing (PARA 2010) (LNCS, Vol. 7134)*, Kristján Jónasson (Ed.). Springer, Berlin, Heidelberg, 410–418. https://doi.org/10.1007/978-3-642-28145-7_40
- [15] Sameer S. Shende and Allen D. Malony. 2006. The Tau Parallel Performance System. *The International Journal of High Performance Computing Applications* 20, 2 (May 2006), 287–311. <https://doi.org/10.1177/1094342006064482>
- [16] Technische Universität Dresden and Forschungszentrum Jülich GmbH. 2019. *Score-P User Manual*.
- [17] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. 2011. EZTrace: A Generic Framework for Performance Analysis. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 618–619. <https://doi.org/10.1109/CCGrid.2011.83>
- [18] Ilya Zhukov, Christian Feld, Markus Geimer, Michael Knobloch, Bernd Mohr, and Pavel Saviankou. 2015. Scalasca v2: Back to the Future. In *Tools for High Performance Computing 2014*, Christoph Niethammer, José Gracia, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel (Eds.). Springer, Cham, 1–24. https://doi.org/10.1007/978-3-319-16012-2_1