MPI PERUSE

An MPI Extension for Revealing Unexposed Implementation Information

Version 2.0

Abstract

This document describes an interface extension to the MPI-2 standard. The interface exposes useful information about an MPI implementation's internal state. The interface is appropriate for those who require an increased understanding of MPI internals such as those developing parallel development application tools.

Point of Contact

Terry Jones trj@llnl.gov 925.423.9834

Major Contributors

The following contributors have/are participating in telecons, meetings (e.g. SC BOFs), etc.

Brian Barrett, Indiana University
David Bernholdt, Oak Ridge National Laboratory
Ron Brightwell, Sandia National Laboratory
Lars Ailo Bongo, University of Tromso, Norway
George Bosilca, University of Tennessee, Knoxville
Ana Cortes, Universitat Autonoma de Barcelona
Toni Cortes, CEPBA
Jim Coyle, Iowa State University
Bronis R. de Supinski, Lawrence Livermore National Laboratory
Rossen Dimitrov, MPI Software Technology Inc.
Sevki Erdogon, University of Hawaii
Hans-Christian Hoppe, Pallas/Intel
Graham Fagg, University of Tennessee, Knoxville
Ferdinand Geier, Cluster Competence Center
Judit Gimenez, CEPBA
Rich Graham, Los Alamos National LaboratoryWilliam Gropp, Argonne National Laboratory
David Gunter, Los Alamos National Laboratory
Steve Healey, Intel
Curtis Janssen, Sandia National Laboratory
Karen Karavanic, Portland State University
Rainer Keller, HPC Center, Stuttgart
Bernie King-Smith, IBM
Terry Jones, Lawrence Livermore National Laboratory
Darren Kerbyson, Los Alamos National Laboratory
Jesus Labarta, CEPBA
Brian LePore, Powrsurg.com
Andrew Lumsdaine, University of Indiana
Chee Wai Lee, University of Illinois, Urbana Champaign
Rusty Lusk, Argonne National Laboratory
Dave Merril, Unisys
Bernd Mohr, FZJ
Kathryn Mohror, Portland State University
Matthis Mueller, HPC Center, Stuttgart
Beth Noble, IBM
Bob Numrich, University of Minnesota
Patrick Ohly, Pallas/Intel
Dhabaleswar Panda, Ohio State University
Kurt Pinnow, IBM
Kumaran Rajaram, MPI Software Technology Inc.
Hubert Ritzdorf, NEC Research Labs, Europe
Phillip Roth, University of Wisconsin
Martin Schulz, Lawrence Livermore National Laboratory
Miquel Senar, Universitat Autonoma de Barcelona
Tony Skjellum, MPI Software Technology Inc.
Jeff Squyres, Indiana University
Richard Treumann, IBM
Tim Woodall, Los Alamos National Laboratory

Acknowledgements

We appreciate support shown by the following:

Karl Feind, SGI
MaryDell Nochumson, Los Alamos National Laboratory
Susan Post, Los Alamos National Laboratory
Jeffrey Vetter, Oak Ridge National Laboratory

Revision Control

Revision	Date	Changes	Changed By	Affected Sections
1.0	4/22/2002	Initial version	R. Dimitrov	
1.1	5/7/2002	Added comments from R. Brightwell, B. de Supinski, C. Janssen, T. Jones, S. Post	T. Jones	
1.2	5/20/2002	*Reflects comments provided in 1.1 and discussion at LLNL on May 13, 2002.* Clarification about scope of document Clarification about sample profiler Clarification about levels of support and scope Clarification on portability Adding an example for callbacks Changes in PERUSE API Added discussion on thread safety Added discussion on layered libraries Added discussion on conditional callbacks Adding querying PERUSE functions Changed Peruse_types.h Changed Peruse.h Changed examples (added one more)	R. Dimitrov	 1.2 1.2 2.2 2.4 2.5 3.2 3.3 3.4 3.5 2.6 Appendix A Appendix B Appendix C
1.3	6/4/2002	First version of document intended to be a standalone, open specification to the High Performance Computing community at large. (Earlier versions were specifically targeted at a deliverable in MSTI's ASCI contract with LANL, LLNL, and Sandia.)	T. Jones	Title page 1.1 Inserted new 1.2,1.3,4.2,4.3,4.4
1.4	7/20/2002	Adopted name MPI_Peruse	T. Jones	
1.5	8/22/2002	Made clarifications in the Background and Scope sections; Clarifications of objective; New section on design concepts; Added a new section (chapter) for clarification on the concepts and terms used in this document; Changed a function prototype Joined Appendix A and B Appendix C became Appendix B Added a new example to Appendix B	R. Dimitrov	1.2, 1.3 2.1 2.2 3 4.2.3 Appendix A Appendix C Appendix B
1.6	8/27/2002	Minor edits, clarifications to 1.3, 2.1, 2.2, 3.1, 3.2, 3.3, 3.4, 3.6, 3.7, 3.8	T. Jones	1.3, 2.x, 3.x
1.7	9/25/2002	Discussion on debug and production libraries; More detail on the callback design approach, which is now equivalent to the query approach; Clarification on levels of support and compliance; Clarification on user program portability; Adding sender based matching for message queues; Clarification on collective metrics; Explain how PERUSE can be used to provide global scope Additional clarification on the meaning of metrics and when measurements are taken; Fixes in the API, definitions, and types; Clarification on callbacks and the event based model; Editing and adding new examples	R. Dimitrov K. Rajaram	2.1 2.2 2.3 2.5 3.4, 3.5 3.6 3.8 4.1.1 4.2 4.5 Appendix B
1.8	3/14/2003	Removed the requirement for using the same constant values in all implementations Removed MPI_Datatype metrics; Removed API calls that work with MPI_Datatype; Removed metric groups and corresponding types Removed API calls for metric groups Eliminating the query mechanism; Emphasizing only user callbacks as the only mechanism for collection of metrics data; Removed the statistics types and API calls Renamed Peruse.h to peruse.h and all Peruse_xxx_t types to peruse_xxx_t Modified peruse.h;	R. Dimitrov	2.1 2.2 2.4 4.1 4.2 4.4 Appendix A Appendix B

		Modified examples;		
1.9		Update contributors and add acknowledgements Adding new definitions Clarifying portability issues Clarifying the distinction between message matching and completion of requests Added "shipped bytes" for MPI I/O to account for non-local I/O operations Adding a new error code PERUSE_ERR_MPI_OBJECT Adding message envelope information to the callback parameters Added requirements for user callbacks Changed the meaning of the user callback return codes Clarification on return codes and MPI error classes Changed the thread-safety semantics Modified Appendix A and B Adding Appendix C and D for listing the PERUSE API functions and constants	R. Dimitrov T. Jones	1.4 2.4 3.4 3.7 4.2 4.1.2 4.1.2 4.1.2 4.2 4.3 Appendix A, B Appendix C, D
1.10		Reflected the event model Added clarification for unexpected/early arrival messages Added a new section that focuses on PERUSE events Moved sections to Appendix E Adding Appendix E with collective, one-sided, and file I/O events not defined in this version for future consideration Updated peruse.h and al examples Updated function and constant lists	R. Dimitrov T. Jones	1.3, 2.1 3.5 4 3.6, 3.7, and parts of 4.1.1 Appendix E Appendix A, B Appendix C, D
1.11	10/1/2004	Clarification on the message matching and completion Listing constraints on callback's code more directly and adding clarification on callback return codes Clarification on when PERUSE_Init can be called Clarification on error condition when MPI object is freed Removed old text left from the statistics model Added section to treat relationships between MPI and PERUSE handles	R. Dimitrov T. Jones	3.4 5.1.2 5.2.1 5.2.14 5.4 5.6
1.12	2/6/2006	Clarified subset of MPI covered by PERUSE. Additions and corrections to Definitions, Abbreviations and Acronyms Clarified scope limited to point2point messaging Added pseudo-code example Clarifications about events, added simple event diagram Clarified detailed message event diagram Added clarification for intended users Added clarification on activation windows Added clarification on events from collective calls Added clarification about PERUSE use with multiple tools Added PERUSE_Lock and PERUSE_Unlock Added PERUSE_PER_TAG, PERUSE_PER_SOURCE Added K. Mohror proposal to track MPI objects Added K. Mohror proposal on dynamic process creation Added K. Mohror proposal on remote memory access Added K. Mohror request for more info on MPI-IO Added request for more info on Control packets	T. Jones K. Mohror.	1.1, 1.3 1.4 2.1, 5.2.x, 8., 9., 10., 12.x.x 2.1 4. 4.3.2 5. 5.1.1 5.1.2 5.2.1 5.2.16, 5.2.17, 5.3 11. 12.412.5 12.6 12.7 12.8
1.13	2/25/2006	Corrected detailed message event diagram Removed spurious references to *win* and *file* Corrected return codes for PERUSE user callbacks Added PERUSE_PER_PEER Clarifications on PERUSE_Lock() Clarifications of semantics in multi-threaded mode Removed references to other language bindings Add new PERUSE lock error codes	T. Jones	4.3.2 5.1, 5.1.1, 5.1.2 5.1.2 5.2.6 5.2.16 5.3 6.2 8.0, 11.0
2.0	3/17/2006	Minor grammatical corrections Removed spurious references to *win* and *file*	T. Jones	1.3,2.2,5,5.1.2,5.6 5.1, Appendix A

Table of Contents

1. Main Concepts and Terms

1.1 Purpose

This document presents the design for an MPI performance revealing extensions interface (PERUSE). The extensions are intended to provide greater insight into the interactions between application software, system software, and message-passing middleware that take place in a parallel environment typical for supercomputer applications. In particular, the interface is designed to operate with a subset of version 2 of the Message Passing Interface (MPI-2) [1, 2].

1.2 Background

This specification is an outgrowth from a proposed interface designed by MPI Software Technology Incorporated (http://www.mpi-softtech.com) [3].

In addition, the current specification reflects the thinking and direction of multiple institutions with a long history of commitment and use of MPI including Lawrence Livermore National Laboratory, Los Alamos National Laboratory, Sandia National Laboratory, Pallas, and MPI Software Technology Incorporated. A large number of PERUSE features presented in this document are based on a requirements gathering phase carried out among MPI users in the three ASC labs.

1.3 Scope

This document presents the specification of PERUSE, a conceptual design, definition of PERUSE API with C bindings, an example PERUSE C include header file, and a set of examples for demonstrating the used of PERUSE.

After consideration, we have decided to release the first specification with a focus on MPI's point-to-point message passing. The hope is that we will learn from both MPI implementations and parallel tools, and that the more complicated aspects of MPI (including MPI-IO, collectives, MPI one sided, dynamic MPI, and so on) will profit from these experiences. We do anticipate that PERUSE will be valuable for the study of MPI-IO, Collectives, and MPI-1Sided usage and have included an initial strawman for how PERUSE might be extended in Appendix E (see section 12).

PERUSE is intended to facilitate the development of parallel program development tools such as profiler tools and debugger tools; it is not intended to provide asynchronous extensions to MPI for user level applications.

1.4 Definitions, Abbreviations, and Acronyms

API: Application programmer's interface.
ASC: Advanced Simulation and Computing Program. A United States Department of Energy program created for science-based Stockpile Stewardship.
Event callback: user-defined callback function registered with the MPI library
Event callback activation window: Period during which event callbacks will be called when the MPI event of interest occurs. Note that PERUSE activation windows can be overlapping; more than one PERUSE window may be active at any given time.
LANL: Los Alamos National Laboratory
LAPI: Low-level Application Programming Interface: an active-message-type API for optimal communication through the IBM SP switch. Provides reliable, unordered communication between all processes in the MPI world.
LLNL: Lawrence Livermore National Laboratory
MPI: Message Passing Interface.
MPI-2: Extensions to the MPI standard.
MPI I/O: An MPI extension allowing for the manipulation of files on different file systems.
MSTI: MPI Software Technology Incorporated

PERUSE: MPI Performance examination and revealing unexposed state extension specification – the specified API.
PERUSE Event: Internal MPI processing events of interest to PERUSE.
PERUSE Implementation: interfaces, utilities, and mechanisms provided by an MPI implementation in order to support PERUSE.
PERUSE User: software that uses the PERUSE interface.
PERUSE Specification: the draft document that defines PERUSE (this document).
PMPI: Profiling interface for MPI specified by the MPI standard.
Portals: Low-level API providing reliable and ordered communication for various interconnects and machines including Myrinet/Cplant and Red Storm.
Sandia: Sandia National Laboratories
SMP: Symmetric multiprocessor.

2. General Design Considerations

2.1 Design objectives of PERUSE

The main objective of PERUSE is to provide MPI application and performance tool developers with the capability to obtain low-level performance data unavailable through the standard MPI profiling interface in a non-intrusive manner. The PERUSE design provides an interface that suggests low processing overhead by allowing the user to collect data only for the MPI events that are of interest during the periods of program's execution of user's choice. This fine grain level of control minimizes unnecessary processing not relevant to the profiler's goals. The fundamental design concept of PERUSE is the use of user callbacks for registering MPI processing events of interest. These events are related to the MPI internal processing of user requests for point-to-point message passing. (As explained earlier in section 1.3, although this original version of PERUSE focuses on message-passing it is anticipated that future versions of PERUSE will likely include events specific to file I/O, one sided communication, collectives, and so on.) The event callback design leaves data collection, metrics definition, and statistics processing to users, thus further reducing the processing overhead in the MPI library and the same time simplifying the implementation of PERUSE.

It is not an objective of PERUSE to create an abstract model of MPI implementations or to force MPI libraries to comply with such a model and subsequently implement PERUSE against this model. Such an abstract model will not be useful in meeting the main objectives of PERUSE and may actually be counter-productive in terms of achieving these objectives. It is expected that certain MPI events defined in this specification (also referred to as *PERUSE events* hereinafter) will not be applicable to some MPI implementations either because the implementation mechanisms chosen by the specific libraries do not match the definition of the events or because implementing the callback mechanism for these events might be too intrusive.

It is recognized that attempts to measure a given attribute of a program may perturb the program. As measurements become more intrusive, they may actually become less valuable. PERUSE expects that MPI libraries will provide only information which is both accurate and relevant to their architecture's performance. The definition of PERUSE events is based on common MPI concepts; however, it is ***not*** expected that all MPI implementations will be able to supply relevant performance information for all concepts included in PERUSE. If an event suggests that the MPI library needs to create artificial constructs in order to present relevant performance data, it is best to not provide the particular event. PERUSE provides a portable ASCII string based query mechanism to allow users to query the MPI library implementations about which PERUSE events are supported. This mechanism will not cause compiler/link problems for applications that are written to utilize this mechanism. Also, it is suggested that the MPI libraries provide efficient mechanisms for running with and without PERUSE. This may be accomplished by different builds with and without PERUSE (e.g. debug and production

libraries), or by a dynamic implementation which is able to switch code paths. Quality MPI implementations will expose many PERUSE events, even such that are intrusive, but possibly still providing useful performance information without affecting the peak performance of the production library.

The main objective of PERUSE can be summarized in the following statement: ***PERUSE presents non-portable MPI performance related information in a portable manner.***

2.2 Design concept

The PERUSE design presented in this document is based on the use of user defined callback functions that are invoked by the MPI library when an event of interest to the PERUSE user occurs. A special callback registration facility is provided. Using this model, the application or the performance-monitoring tool, requests that the MPI library invoke a user registered callback at the places where the library performs operations related to the specific events. The section of this document that describes events definition provides more details on when the

PERUSE Pseudo code
(see Section 9.1.4 for corresponding example in C)

```
Main()
{
      initialize MPI;
      initialize PERUSE;
      initialize PERUSE event handles;
      register PERUSE events with My_callback_routine();
      activatePERUSE event handles;

      while (parallel tool is active) {
          do normal parallel tool stuff;
      }

      report statistics gathered by My_callback_routine();
}

My_callback_routine()
{
     get PERUSE event;
     switch (event)
    {
          update statistics based on event information
    }
}
```

callbacks will be invoked.

2.3 Levels of support and compliance

PERUSE support can be provided at different levels by its implementations. A portable mechanism for querying PERUSE about its level of support is provided. This mechanism is based on ASCII string queries. All PERUSE implementations must provide the full set of API functions, data types, and constants. The optional support refers only to the set of events that are supported by the implementation. Since the goal of PERUSE is to provide accurate and detailed performance information in a non-intrusive manner, MPI libraries that are unable to provide an event without significant performance or design impact are encouraged not to implement this event.

PERUSE encourages MPI implementations to provide implementation specific events that are not included in the specification but can offer beneficial information to users. PERUSE provides a special discovery mechanism for querying all supported events in a portable manner, including the implementation-specific ones. If the MPI implementation provides implementation-specific events, it is the responsibility of this implementation to describe their meaning and intended way of use.

2.4 Portability

PERUSE facilitates both implementation and user-level portability. PERUSE allows implementations to chose the specific mechanisms for declarations of data types and constants so that providing PERUSE extensions by MPI vendors requires minimum structural changes and processing in the existing MPI libraries. User-level portability is similar to the user-level portability of the MPI standard – it is guaranteed by a standardized set of API function calls, data types, and constants. Also, in the same vein as *mpi.h*, PERUSE suggests that a header file named *peruse.h* is used by all implementers.

This specification does not aim to provide binary interoperability, as this has not been among the goals of the MPI standard. The string names of PERUSE events can be represented with any NULL terminated string and are implementation dependent. One possible scheme that will improve the user level portability is if the strings correspond to the definitions of the event identifier constants, e.g., the string for PERUSE_COMM_POSTED_QUEUE_INSERT is "PERUSE_COMM_POSTED_QUEUE_INSERT".

PERUSE implementations are required to support all functions and data types of the API. The optional support is only related to the supported events. For increased portability, it is suggested that the user programs be written so that they use only the string based query mechanism for discovering what events are supported and obtaining the numerical event descriptor identifiers. This will also enable a portable use of non-standard, vendor-specific events.

2.5 Intended audience

The audience of this specification are providers of PERUSE implementations and developers of codes that utilize the PERUSE extensions. Such developers can be either MPI application designers or providers of MPI performance monitoring tools.

2.6 Example uses of PERUSE

Appendix B presents example MPI programs that utilize the PERUSE interface. These examples demonstrate the use of PERUSE and assist the reader in understanding the semantics and intended use of PERUSE and its concepts.

3. Main Concepts and Terms

This section presents definitions and assumptions of the main concepts used in the design of PERUSE. These definitions and assumptions are based on the semantics of MPI as specified in the MPI standard. Of special interest to PERUSE are the issues related to message requests, message transfers, activation/initiation and completion of requests and transfers, message ordering and matching, and the two-sided model of the send/receive message-passing mode of communication defined by MPI.

3.1 Message requests and message transfers

For the purposes of PERUSE, a *message request* represents the specification of the work that the MPI library is requested to perform by the user process, specifically, the message (defined by its buffer address, size, and datatype), the communication operation (send or receive), the peer process (source or target), and the MPI communication space (defined by user tag and communicator). *Message requests* can be created by using non-blocking MPI calls (MPI_Isend, MPI_Irecv, MPI_Send_init, MPI_Recv_init) or blocking calls (MPI_Send, MPI_Recv). In the context of MPI-2, the definition of *message request* is extended to include file I/O operations and one-sided communication. The *message request* is also used for notification when the requested work is completed. The MPI library may use various internal mechanisms and protocols to perform the requested work, which eventually include an invocation of one or more *data transfer* operations that move the bytes of the requested message and possibly control packets associated with the MPI message protocols. These *data transfer* operations are provided by the underlying communication system software, such as TCP socket *send()* and *recv()* operations or memory copy operations in SMP configurations. PERUSE defines a *message transfer* as the collection of *data transfers* (one or more) that actually perform the physical transfer of the entire user message, not including control packets that might be used by the MPI library for implementing internal protocols and flow control schemes. For example, the control packets associated with rendezvous protocols are not considered part of the *message transfer*. Consequently, it is expected that for the PERUSE events that indicate request activation and *message transfer* initiation, the MPI library will make two distinct calls to the user-registered callbacks. The period of time between these calls will be equal to the period between the moment when the user activates the request and when the first *data transfer* that actually moves the first byte of the user message (not necessarily the byte with lowest memory address) is scheduled. If the MPI library needs to exchange control packets, which are likely performed by the same *data transfer* (byte movement) operations, these packets should not cause invocation of the PERUSE event that corresponds to *message transfer* initiation.

The *message request* is an MPI concept whereas the *data transfer* (used for *message transfers*) is a generic concept that represents mechanisms provided by the underlying communication system software to move bytes of data from one location to another, regardless of the actual means of this movement. Completing a *message request* involves a *message transfer* (composed of one or more *data transfers*), ordering, matching, completion notification, other library processing, and possibly special protocols that may include additional data transfers (not counted as part of the *message transfer*).

3.2 Request activation and message transfer initiation

Activation of a message request is the moment when the user process executes an MPI call that suggests a communication operation, or according to the MPI terminology, the request becomes “active.” Examples of MPI calls, which activate messages include MPI_Recv, MPI_Irecv, MPI_Send, MPI_Isend, and MPI_Start. A number of PERUSE events refer to

initiation (start, beginning) and completion of *message transfers*. The meaning of these operations is limited only to what the MPI library can guarantee or "see." For example, the beginning of a send *message transfer* for a TCP socket is the moment when the library calls the *send()* system call over the socket file descriptor. Clearly, the MPI library has no knowledge if the operating system will actually initiate the physical transfer over the network interface at the time the *send()* call is made or if the data will be buffered and the physical transfer will begin later. As the low-level communication information is generally unavailable to user level processes (the most common mode of MPI library use), PERUSE does not require that the MPI libraries provide hardware-specific information and all references to certain events and timings are only from standpoint of the MPI library. However, an MPI library with access to low-level hardware or firmware-related information is not restricted from providing such information.

In line with the definition of *message request* and *message transfer*, message request activation and message transfer initiation are two distinct operations, possibly executed with a long period of time in between. Users expect that when they activate a *message request*, the *message transfer* associated with this request will be initiated as soon as possible. Thus, providing a mechanism for measurement of the length of the interval between message request activation and message transfer initiation can significantly benefit MPI user program performance analysis.

For protocols that use "get" based data transfer primitives, the initiation of the send message transfer may be transparent to the sender process, so the MPI library may be unable to detect when the "get" operation is initiated by the receiver. In these cases, it is recommended that the library does not implement special mechanisms for providing the expected sender functionality at the receiver, which may involve additional processing and communication overhead. Therefore, it is recommended that the MPI library that uses "get" based communication primitives for message transfers do not implement the PERUSE events that indicate transfer initiation at the sender. The PERUSE callback mechanism offers an opportunity to performance mentoring tools to make inference about remote events and thus correlate activities on different MPI processes. These tools may be able to provide valuable performance data with global semantics, which is not available directly through the PERUSE API.

3.3 Request completion, request completion notification, and transfer completion

For the purposes of PERUSE, a distinction is made between request completion from standpoint of the MPI library and from standpoint of the user process (user process notification). Since MPI does not provide any asynchronous means of notification, all request completion notifications are done only when the user process specifically requests such notifications thorough the MPI_Wait and MPI_Test family of calls (or within blocking MPI calls). However, the library can effectively complete the message transfer associated with a given request before the user asks for notification. If the period between internal request completion and notification is long, the parallel algorithm designer may decide to check the status of the request of interest earlier or more frequently. PERUSE is designed to provide such detailed information.

Message request completion and message transfer completion are distinct operations, similar to request activation and transfer initiation as described above. Request completion refers to the moment when the library internally marks a message request completed. Message transfer completion is the moment when the library has scheduled for sending the last byte of an outgoing message or has received the last byte of an incoming message. The MPI library can indicate request completion immediately after a message transfer has completed and often the two completions are equivalent with respect to time. However, in other designs, the MPI library may

not indicate completion of the request immediately after the completion of the message transfer, thus there may be a delay between transfer completion and request completion.

For MPI libraries that use remote memory operations for indicating completion of message transfers (such as in the case when a memory flag is updated through a remote DMA operation), providing an accurate timing about transfer completion may require substantial processing overhead, similar to the one described above for the case of "get"-based protocols. In such cases, it is recommended that the MPI library forego implementing PERUSE events that facilitate measurements related to the message transfer completion timing.

3.4 Message/Request queues

The MPI standard defines the semantics of message ordering and of the matching of receive requests to sent messages (Section 3.5 of the MPI 1.1 standard). Messages are non-overtaking. Thus, if a process sends two messages in the same communication domain to the same receiver, using the same message tag, the receiver must match them in the order that they were sent. The MPI standard does not mandate any particular order in which the message transfers will be actually completed after they are matched. If the receiver posts two receive requests with the same envelope, not using wild cards, the second request cannot be matched before the first one. These ordering and matching rules for sends and receives imply that the MPI library needs to maintain an internal order of the message requests. For the purposes of PERUSE, the mechanism that is generically used to represent this order is called a "message queue" or alternatively a "request queue". PERUSE does not attempt to present an abstract model for implementing the message queues and the protocols for message transmission, and does not favor receiver based matching versus sender based matching. PERUSE provides the same queue concepts for both sender and receiver-based matching.

Note that the "message queue" and "request queue" need not be implemented via any specific programmatic mechanism or data structure; it is only necessary that they preserve the ability to provide MPI ordering and matching semantics. Thus, PERUSE does not impose any specific architecture or programmatic approaches to the MPI implementations. Furthermore, PERUSE does not mandate whether the specific mechanisms that implement the message/request queues are global for the entire library, or on a per-communicator basis. Other implementations are also possible. (PERUSE provides a query mechanism to inform the user about the scope of the message queues.) Consequently, message and request queues used in PERUSE to represent the MPI ordering and matching semantics should not be confused with the actual implementation of these concepts.

3.5 Expected (posted) and unexpected (early arrival) queues

The MPI send/receive mode of communication (as opposed to the one-sided mode of communication defined in MPI-2) is a two-sided model. According to this model, the necessary (but not sufficient) condition for a message to be transferred from the sender to the receiver is that the sender activates a send request and the receiver activates a matching receive request. Since the sender provides the entire information about the message (including the content of the message) at the moment when the send request is activated, no *message transfer* can be initiated before the sender actually activates the send request. The MPI standard refers to this model as "push" two-sided communication. Both the send and receive requests have local semantics and the MPI standard does not impose any requirements in terms of temporal ordering of matching send and receive requests, thus allowing a send request to be activated at the sender before or

after a matching receive request is activated (posted) at the receiver[12]. According to the relative time at which the receiver posts a receive request and the time at which the sender posts the send request PERUSE defines two types of message/request queues related to the receiver process – expected (posted) and unexpected. Unexpected messages are sometimes called "early arrival" messages and are unexpected from standpoint of the MPI library – a message envelope sent by the sender process arrives prior to (earlier than) the matching receive request activation by the receiver. These unexpected messages are in fact "expected" from standpoint of the user program but not yet posted.

The definition of the receive posted and unexpected queues is relevant to MPI libraries that perform the matching at the receiver process. PERUSE introduces a similar definition about expected and unexpected queues with respect to the sender process for MPI libraries that perform the matching at the sender process. Hybrid matching models are also possible.

It is important to note that the above definition allows for two alternative models of MPI message protocol implementations – "pull" based and "push" based – and that PERUSE does not impose an implementation requirement of push, pull, or both models. Both of these message protocol models can support the MPI two-sided communication semantics correctly. According to the first model, the MPI library will inform the receiver about the envelope of the send message only after the send request has been posted, regardless of whether the matching receive request has been posted before or after the send request. According to the second model, the MPI library will notify the sender about the envelope of the expected message at the receiver process only after the receiver posts its receive request, regardless of whether the send request has been posted before or after the receive request. In this second model, the sender cannot initiate a message transfer before the receiver sends the message envelope. A third, hybrid model is also possible, and allowed by the PERUSE definition of expected and unexpected queues. For simplicity of the presentation, the following explanations are introduced only for the receiver based matching model. These explanations can also be related to the sender based matching without additional semantic changes.

The expected (posted) receive queue defined by PERUSE contains requests that are posted by the receiver before the matching send requests have been posted (or more precisely, before the send message envelope have arrived at the receiver). The time that a receive request spends on the posted queue is the period between the moment when the user calls an MPI function for activation of a receive request (MPI_Recv, MPI_Irecv, or MPI_Start on a persistent receive request created by MPI_Recv_init) and the moment when the library receives a message envelope that matches the posted request. A possible analysis based on the duration of the time that a request spends on the posted queue may conclude that the particular request may have been completed earlier if the algorithm can allow the matching send message to be sent earlier.

The unexpected message queue contains message envelopes (possibly also including the actual message data) of messages that have arrived before a matching receive request has been posted. The duration of time a message spends on the unexpected queue or the length of this queue may indicate to the MPI performance analyst that this particular process falls behind the other processes in the MPI job and is unable to process the incoming messages in a timely manner. This may indicate a performance and scalability bottleneck in the parallel system. Such critical information for a detailed performance analysis is unavailable through PMPI.

---

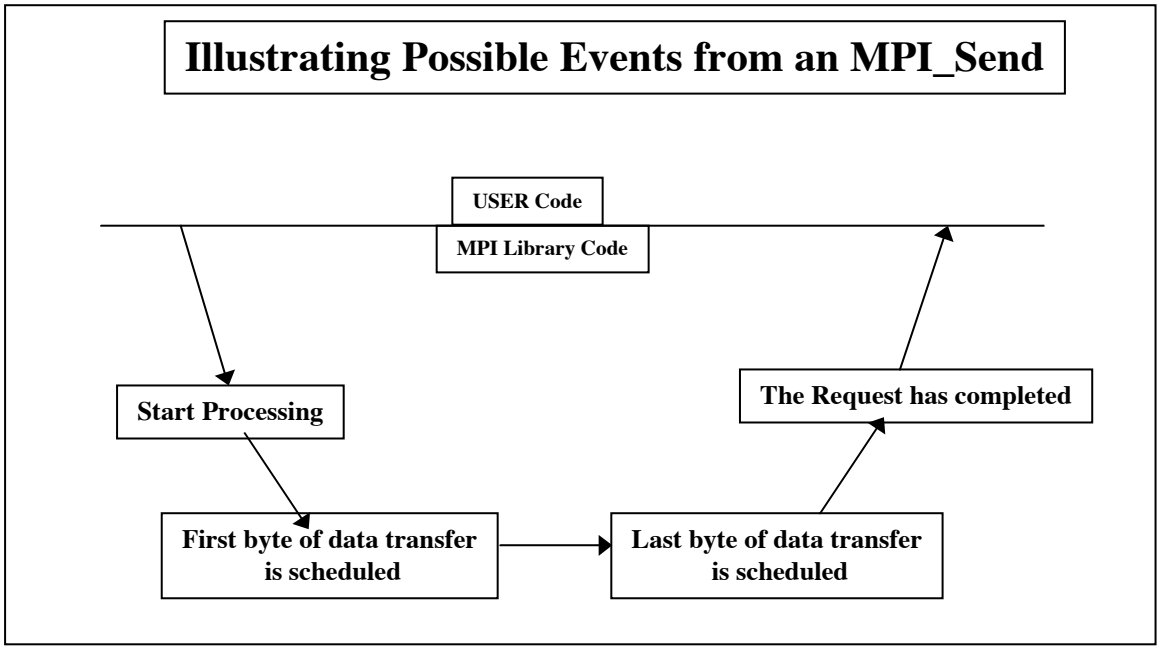
[1] This documents uses interchangeably the terms "post" and "activate" in relation to message requests.

[2] This does not apply to MPI_Rsend().

4. EVENTS

PERUSE events are one of the fundamental concepts of PERUSE. These events are connected to certain activities/phases of the internal MPI processing associated with user message requests. Examples of such events are the insertion/removal of requests into/from the posted or the unexpected request queues.

This section of the document presents the events and their descriptions defined in this specification. In this version of the specification, only events related to point-to-point communication and request queue operations are provided. These events are registered with communication (MPI_Comm) objects. Collective operations events associated with MPI_Comm objects as well as events associated with MPI_File and MPI_Win are discussed in Appendix E and are subject of further clarification in subsequent versions of the specification. The diagram below illustrates possible events associated with an MPI_Send. Actual events and the time that the callback is expected will differ depending upon the MPI implementation. For example, the events and callback timing for an RDMA based system, shared memory based system, and tcp/ip based system will likely have differences.



4.1 Association of events with requests

In order to allow PERUSE users to make efficient use of the event callbacks, a mechanism for correlating events related to the same message requests is needed. In response to this need, PERUSE requires that the MPI library pass the *same* request identifier to the event callbacks when the callbacks are associated with the *same* message request. The identifier must be *unique* during the period between the creation of the request and its release. This uniqueness is necessary for PERUSE users to be able to relate measurements taken during the event callback invocations to the same internal request, thus allowing for the collection of valid performance data. There are no other requirements on the request identifier.

PERUSE providers may elect to pass to PERUSE event callbacks the MPI_Request handles as the unique identifier, if the requests were created with calls to the non-blocking MPI

API's. However, this is not mandatory and the PERUSE implementation is free to choose any mechanism for generating unique identifiers as long as it meets the uniqueness requirement.

It is important to note that PERUSE does not provide mechanisms for uniquely linking callback events to specific user level MPI API calls. Although clearly useful, such linking is not sufficiently supported by MPI's API - MPI does not provide unique identifiers/handles for all user requests, as in the case with blocking MPI_Send and MPI_Recv. In order to assist PERUSE users in achieving such linking (if desired), PERUSE requires that the MPI libraries pass a request specification parameter to the event callbacks (see section 5.1.2 for more details). This parameter carries information about the input parameters passed to the MPI calls that the user made in order to create the particular message request.

4.2 Scope of PERUSE events

In its current version, all PERUSE events have local scope. Events with global scope may be able to also provide useful performance information. However, such events will require communication of control packets for the exchange of event-related information and facilities for handling such control communication. This is considered beyond the current scope and purpose of PERUSE but may be pursued in future efforts. Through its callback mechanism, PERUSE allows layered tools to be invoked by the library on the critical message processing path and possibly make correlation between events on remote processes. This approach could be successfully used for implementing global events.

4.3 Point-to-point communication events

Point-to-point events are intended to trace the phases of the execution of a user request from its creation to the user notification of its completion. The event definitions and descriptions are followed by a request processing flow diagram that specifies the sequence in time of the generation of these events.

The events in this section are divided into two groups – (i) PERUSE_COMM_REQ events generated during activities associated with MPI processing of user requests and (ii) PERUSE_COMM_MSG events generated when an incoming message that will be used in point-to-point matching with user requests arrives. The second type of events is not associated with any particular user request and will have a unique ID different from the request unique ID matched to the incoming message. The rationale for the second type of events is that these events can help PERUSE users to observe activities related to incoming messages and unexpected queues and discover processing or communication imbalances in the MPI jobs. For example, if the average time for a message spent in the unexpected queue is large, this may indicate to the MPI application designer that the observed process is falling behind possibly as a result of larger processing or communication load. The designer can then attempt to improve the load balance by altering the data distribution or communication pattern of the application algorithm.

The two groups of events (REQ and MSG) are indicated with different line styles in the event diagram. The solid lines show transitions between REQ events and these events are associated with user requests. All events connected with solid lines will have the same unique request ID for the same user request. The dashed lines represent MSG events. They will have the same unique ID for the same incoming message, but it will be different then the unique ID of the request to which the message will be matched.

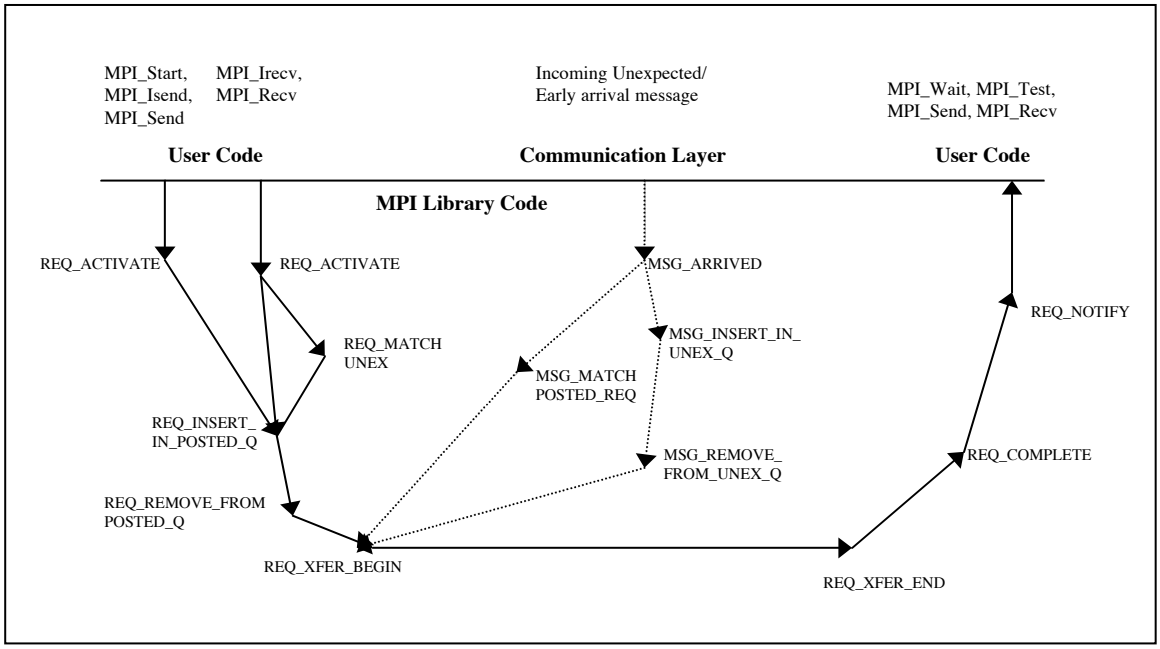
4.3.1 Request and message event definition and description

PERUSE_COMM_REQ_ACTIVATE	This event indicates that the MPI library starts processing that would lead to the message transfer specified by the user request. This event will be generated by MPI_Start, MPI_Startall, MPI_Irecv, MPI_Isend as well as in MPI_Send and MPI_Recv.

	Rationale. This event indicates to the PERUSE user that the MPI library has entered the critical message path. A time stamp here can be used as a mark to measure various time periods, such as ACTIVATE to XFER_BEGIN, or ACTIVATE to COMPLETE.
PERUSE_COMM_REQ_MATCH_UNEX	This event is generated when the MPI library matches a user request to an unexpected message. *Rationale*. This event can be used in determining the delay between the moment when the request is matched and the beginning of the message transfer.
PERUSE_COMM_REQ_INSERT_IN_POSTED_Q	The MPI library inserts a request in the posted request queue. No match was found to an unexpected message in the unexpected queue. *Rationale*: This event can be used to measure how long a request stayed in the posted queue before it was matched as well as the length of the posted queue.
PERUSE_COMM_REQ_REMOVE_FROM_POSTED_Q	The MPI library removes a request from the posted request queue as a result of successful matching to an incoming message. This can be caused by MPI_Cancel(). *Rationale*: See PERUSE_COMM_REQ_INSERT_IN_POSTED_Q.
PERUSE_COMM_REQ_XFER_BEGIN	This event indicates that the MPI library has schedules the first data transfer associated with the message transfer specified by the user request. The message transfer may be composed of multiple data transfers. Control messages used by MPI library protocols are not counted as part of the message transfer. See sections 3.1 and 3.2 for more detail. *Rationale*: This event can be used by PERUSE users to measure how long it took the MPI library to begin a message transfer after the user request was posted and started. Long delays can indicate that messages are not progressed in a timely fashion. If the library does not have an independent message progress engine, this may indicate that the user process may need to call the MPI library more frequently in order to help in the progress of the scheduled messages.
PERUSE_COMM_REQ_XFER_END	The callback is called with this event when the MPI library has scheduled the data transfer (with the last byte) of the user message for transmission with the underlying communication method. *Rationale*: MPI libraries commonly use specifically designed protocols for exchanging messages. Depending on the implementation of these protocols and the interaction between the user process and the MPI library, these protocols may not operate in a fashion that is expected by the user. This event, in collaboration with the XFER_BEGIN and REQ_START can give an indication of how quickly the messages are sent once the user posts a send request.
PERUSE_COMM_REQ_COMPLETE	The callback will be called when the MPI library marks the request completed for internal purposes. If the user can make a synchronization call, such as MPI_Wait to MPI_Test following this event, this synchronization call will succeed. *Rationale*: PMPI does not allow to check when the communication associated with a request is actually completed. Overly long times from completion to notification can indicate to the programmer that checks for completion can be made earlier or more frequently.
PERUSE_COMM_REQ_NOTIFY	The user process is notified about the request completion. The callback is called during a synchronization call, such as MPI_Wait and MPI_Test or before the library return from MPI_Send or MPI_Recv. *Rationale*: See PERUSE_COMM_REQ_COMPLETE
PERUSE_COMM_MSG_ARRIVED	This event is generated when the MPI library receives an incoming message from the communication layer, which will be

	used for matching with user requests. Control messages or messages associated with one-sided communication and file I/O will not generate this event.
PERUSE_COMM_MSG_INSERT_IN_UNEX_Q	The MPI library inserts an unexpected (early arrival) message into the unexpected queue. The arriving message is not matched to a request in the posted queue. *Rationale*: This event can be used to measure how long a request stayed in the unexpected queue before it was matched as well as the length of the unexpected queue.
PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q	The MPI library removes a message from the unexpected message queue as a result of successful matching to a user request. *Rationale*: See PERUSE_COMM_MSG_INSERT_IN_UNEX_Q.
PERUSE_COMM_MSG_MATCH_POSTED_REQ	This event is generated when the MPI library matches an incoming message to a posted request. The message will not be inserted to the unexpected queue.

4.3.2 Request and message event diagram



4.4 Queue search events

The queue events are intended for measurements of the internal processing overhead that the MPI libraries incur in operations related to request and message matching. Two categories of events are defined – events related to posted queues and events related to unexpected queues.

PERUSE_COMM_SEARCH_POSTED_Q_BEGIN	This event is generated when the library begins a search in the posted queue for matching an incoming unexpected queue. *Rationale*: Using this event, users can observe the time spent on searching in the unexpected queue. This information may help to discover source of application communication optimization so that the size of the unexpected queues is reduced, thus

	reducing the processing overhead associated with searching.
PERUSE_COMM_SEARCH_POSTED_Q_END	This event is generated when the MPI library finishes a search in the posted queue for matching of an incoming unexpected message. *Rationale*: See PERUSE_COMM_SEARCH_POSTED_Q_BEGIN.
PERUSE_COMM_SEARCH_UNEX_QUEUE_BEGIN	The event is generated when the library begins a search in the unexpected queue for matching a posted request to an unexpected message. This event together with PERUSE_COMM_SEARCH_UNEX_Q_END gives information about the processing overhead related to matching a posted request. This overhead will depend on the length of the unexpected queue. *Rationale*: Using this event, users can observe the time spent on searching in the unexpected queue. This information may help to discover source of application communication optimization so that the size of the unexpected queues is reduced, thus reducing the processing overhead associated with searching.
PERUSE_COMM_SEARCH_UNEX_Q_END	This event is generated when the MPI library finishes a search for matching in the unexpected queue. *Rationale*: See PERUSE_COMM_SEARCH_UNEX_QUEUE_BEGIN.

5. PERUSE API

The PERUSE design offers a uniform and compact API (presented in section 5.2 and Appendix A) with an extensible structure that enables easy addition of new events. The small number of functions is intended to facilitate easy adoption by parallel development application tool developers. End users familiar with an asynchronous callback programming paradigm may find PERUSE useful for tuning studies, but we recommend parallel development tools when available. PERUSE is composed of a set of data types and constants and a set of function calls, forming the API of the interface. The PERUSE specification defines only C bindings.

5.1 PERUSE types and constants

PERUSE defines the following types –

- *peruse_event_h*
- *peruse_comm_spec_t*
- *peruse_comm_callback_f*

These types are defined in the *peruse.h* header file. The *peruse_event_h* type represents the PERUSE event handle. It is an opaque object that is intended to improve portability of the interface, to facilitate different compliant implementations, and to help users write portable layers on top of PERUSE. The actual type definition (typedef) is left to the implementers. The *peruse_xxx_callback_f* are the types for the user callback functions that are used for notifying the PERUSE user when events of interest occur. The *peruse_xxx_spec_t* types are used to provide to user callbacks information about the specific MPI operation that caused the invocation of the callback. Since the MPI standard does not provide a mechanism for explicit annotation of communication and I/O operations, this information can be used by the callbacks to make a correlation between the MPI library processing that caused the callback invocation and a specific user request. More discussion on this topic was presented in section 4.1. The fields in the *peruse_xxx_spec_t* types provide the values that the user passed to the MPI library when making the communication or I/O requests whose processing resulted in the callback invocation.

PERUSE also uses a number of constants whose declarations are left to the implementations. All constants must be of C integer types *int* or *long*. PERUSE constant declarations can be enumerations, C definitions, or constant declarations and will be listed in *peruse.h*. Variables of type *int* initialized during *PERUSE_Init()* are also permitted. However, in this case, the vendor should provide adequate documentation to explain any pertinent restrictions. Appendix A provides and example *peruse.h* C header file. Appendix D provides a list of the constants that must be supported by all PERUSE implementations.

5.1.1 Event handles (peruse_event_h)

PERUSE defines two separate terms for representing events – event descriptors and event handles. Event descriptors describe the events supported by PERUSE and their meaning. Event descriptors do not suggest any processing by the PERUSE-enabled MPI library. They are used for the creation of event handles of type *peruse_event_h*. The event handles represent objects that can be acted upon by the MPI library. The concept of an event handle is introduced in order to allow the user to associate an event descriptor with the context of MPI objects. The MPI object is a communicator (MPI_Comm). Operations on these objects result in communication or I/O activities, which are of interest to this specification. The association of an event descriptor with an MPI object resulting in an event handle is achieved through the invocation of the appropriate *PERUSE_Event_comm_register()* call. The prefix of the name of each event indicates with which MPI object this event is supposed to be registered.

PERUSE event handles have two states: *active* and *inactive*. The term *activation window* is defined as the period during which event callbacks will be called when the MPI event of interest occurs. (Note that PERUSE activation windows can be overlapping; more than one PERUSE window may be active at any given time.) Event handles are inactive during the following periods of the handle lifecycle:

- between handle initialization and opening of the handle activation window, and
- between closing of the activation window and a subsequent activation of the window or handle release.

The activation window of handle is opened with *PERUSE_Event_activate()* and closed with *PERUSE_Event_deactivate()*. The MPI library will not invoke callbacks for inactive event handles. Once the window of a handle is activated, the MPI library will start invoking the callback registered with the event handle at the locations where the library performs relevant to the event operations. If a given event handle is in its window of activation (i.e., it is activated), but the MPI library does not perform relevant operations, the callback will not be invoked.

In summary, in order to cause the MPI library to invoke the event callback the user program must:

- create an event handle for this event by providing a callback function and associating the event descriptor with the desired MPI object,
- activate the event handle by calling *PERUSE_Event_activate ()*, and
- perform MPI activities that are related to the event in question.

If a user creates event handles by attaching the same event descriptor to different MPI objects (for example, MPI_COMM_WORLD and a duplicate of it), the resulting event handles are independent and distinct and their activation windows will not be related in any way. As a result, the user can monitor the same event associated with different MPI objects.

PERUSE allows users to create multiple event handles by attaching the one event descriptor to the same MPI object in different event registration calls. These handles are distinct and their activation windows will be also independent. For example, if an activation window of one of these event handles is opened, the activation windows of the rest are unaffected. The

PERUSE implementation will invoke the callbacks of all activated events in some order. Also, the user can register the same or different callback functions for these event handles. This functionality can be used by multithreaded user programs or by layered MPI libraries. These topics are discussed in more detail further in this document.

5.1.2 User callbacks

PERUSE provides a callback for the MPI object with which PERUSE events can be associated (MPI_Comm); the prototype for this callback is:

```
typedef int (peruse_comm_callback_f)(peruse_event_handle event_h, MPI_Aint unique_id,
                                     peruse_comm_spec_t *spec, void *param);
```

PERUSE callbacks are designed to represent an event-based model for data collection. This model assumes that the user process will perform the actual data collection and statistics processing. The user process can be an instrumented application or a performance-monitoring tool. The registration of user callbacks with event handles is achieved in the event handle constructors *PERUSE_Event_xxx_register*. The callbacks can be set to new values with *PERUSE_Event_xxx_callback_set*. When a new callback is set, the old callback is lost. Callbacks can be registered with *PERUSE_Event_xxx_callback_set* only while the handles are inactive. If the handles are active, callback registration will fail. The value of the currently registered callback can be obtained by *PERUSE_Event_xxx_callback_get*, which can be called on an event handle in both active and inactive state.

The user callbacks are invoked when the MPI library performs activities relevant to the event represented by its event handle. The definition of PERUSE events specifies when events are generated by the MPI library. The constraints on the callback code are as follows:

- When a callback is invoked, it is undefined if the MPI library is under a lock or not and the callback code should not make any assumptions about the lock state of the MPI library;
- Callbacks should be prepared to be invoked from different threads when used with MPI implementations with independent progress engine using internal system threads;
- Callbacks should be signal safe as some MPI libraries use signals for their progress engine and the callback can be invoked form within a signal handler;
- Callbacks should not make any MPI library calls with the exception of *MPI_Wtime()* and *MPI_Wtick()*;
- Callbacks should not hold any locks that are placed around MPI calls in the main code.
- Callbacks should be limited to “read-only” operations on PERUSE handles.

When a user defined callback is invoked, four parameters are passed to this callback (see the definition of the user callback prototypes):

- *event_handle*
- *unique_id*
- *spec*
- *param*

The *event_handle* parameter is the event handle for the event that was registered with the specific MPI object. Using this handle, the callback can perform allowed operations on the handle, e.g., using *PERUSE_Event_get* call, the callback can obtain the event descriptor for *event_handle*.

The parameter *unique_id* is for providing user callbacks with the capability to associate different events for processing of the same request, message, or queue. This parameter is opaque and is implementation-dependent. User callbacks cannot make any assumptions about the actual values of the *unique_id* parameter. The scope of uniqueness of *unique_id* is defined in the following table, depending on the type of events generated. Once the ending event is generated, the value passed to *unique_id* can be reused by the library for a different request, message, or queue and the callback code needs to make appropriate adjustments in order to avoid correlation of unrelated events:

***Unique_id* scope**	
Beginning Event	**Ending Event**
PERUSE_COMM_REQ_ACTIVATE	PERUSE_COMM_REQ_NOTIFY
PERUSE_COMM_MSG_ARRIVED	PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q or PERUSE_COMM_MSG_MATCH_POSTED_REQ
PERUSE_COMM_SEARCH_POSTED_Q_BEGIN	PERUSE_COMM_SEARCH_POSTED_Q_END
PERUSE_COMM_SEARCH_UNEX_QUEUE_BEGIN	PERUSE_COMM_SEARCH_UNEX_Q_END

The third callback parameter *spec* is a pointer to a *peruse_xxx_spec_t* structure that holds information related to the user request that caused the callback invocation. The memory for this structure is allocated and managed by the MPI library. The memory contents is guaranteed to be valid and consistent for the duration of the callback execution. Upon return from the callback, the MPI library can de-allocate the memory for the *spec* structure or can modify its contents.

One of the fields of the *spec* structure is the handle to the MPI object with which the event handle was associated. This handle was also passed by the user in the user request that resulted in the callback invocation. The remaining fields of the spec structures contain the complete request specification in order to allow the code in the callback to correlate the particular callback invocation with a specific user request. The “operation” field has the following values PERUSE_SEND, PERUSE_RECV, PERUSE_PUT, PERUSE_GET, PERUSE_ACC, PERUSE_IO_READ, and PERUSE_IO_WRITE and indicates the type of communication or file I/O operation that caused the callback invocation.

For example, if an event handle is registered by the following call:

```
PERUSE_Event_comm_register(PERUSE_COMM_REQ_COMPLETE, MPI_COMM_WORLD,
                my_comm_callback, NULL, &my_event_h);
```

and a subsequent call to:

```
MPI_Recv(my_buf, 100, MPI_INT, 5, 1001, MPI_COMM_WORLD, &status);
```

is made, the user callback *my_comm_callback* will be invoked when the request corresponding to the MPI_Recv call is marked for internal completion with the following values of its input parameters:

```
event_h = my_event_h
unique_id = <uid>
spec = {
        comm = MPI_COMM_WORLD,
        buf = my_buf,
        count = 100
        datatype = MPI_INT
        peer = 5
        tag = 1001
        operation = PERUSE_RECV
        }
param = NULL
```

The *param* callback parameter is the same parameter passed by the user when the callback was registered. Commonly, this parameter would be the address of some control structure that the callback can use in order to obtain context that might be necessary for its operation. It can also be used for exchanging information between the main user code and the callback. This parameter is meaningful only to the user code and is transparent to the MPI library.

User callbacks are invoked by the PERUSE-enabled MPI library; hence, this library will also obtain the return values of the callbacks. If successful, user callbacks return MPI_SUCCESS. All non-success returns are fatal with the MPI library simply reporting the error was returned by a PERUSE upcall and then cleanly bringing down the job.

Note that while MPI Collectives are not included in this specification, collective calls may generate a point-to-point call back; this will depend on the implementation.

5.2 PERUSE Function Calls

This sub-section describes in detail the API function calls of PERUSE. The functions constituting the API are declared in the *peruse.h* header file (an example version of *peruse.h* is provide in Appendix A). Programs using PERUSE must include *peruse.h*. The API contains the following function groups: environment initialization, event handle registration, event handle manipulation, and user callback manipulation. A complete list of all PERUSE calls is provided in Appendix C.

The return values of PERUSE function calls are defined in *peruse.h*. PERUSE functions can return error codes that indicate that an input MPI parameter is invalid. These return codes are semantically equivalent to the corresponding MPI_ERR_XXX error classes. For example, PERUSE_ERR_COMM indicates that if the input MPI_Comm argument was used in a standard MPI call, the MPI library would have returned an error of class MPI_ERR_COMM. The list of all PERUSE functions return values and their meaning is provided in Appendix D.

5.2.1 PERUSE_Init

Synopsis

int PERUSE_Init()

Input parameters

Output parameters

Return value

PERUSE_SUCCESS, PERUSE_ERR_MPI_INIT,

Description

Used for initialization of PERUSE library run-time infrastructure. Must be called before any other PERUSE function. Must be called after MPI_Init and before MPI_Finalize.

PERUSE_Init may be called from multiple tools.

PERUSE_ERR_MPI_INIT is returned if PERUSE_Init is called before MPI_Init or MPI_Finalize. It is advisable that the user calls PERUSE_Init before any MPI communication operations are initiated as the MPI library may perform communication-related activities that could interfere with the initialization of PERUSE. When multiple calls to MPI_Init are made, only the first initializes PERUSE – the others are equivalent to NOOP and return PERUSE_SUCCESS;

5.2.2 PERUSE_Query_supported_events

Synopsis

int PERUSE_Query_supported_events(int *num_supported, char ***event_names, int **events);

Input parameters

n/a

Output parameters

num_supported – number of supported events and size of the event_names array
event_names – an array of the string names of all supported events
events – an array of the event descriptor identifiers corresponding to the string names

Return value

PERUSE_SUCCESS

Description

This function is intended to provide a portable way for determining what events are implemented by the particular PERUSE implementation. In conjunction with PERUSE_Query_event, by using this function, users can write fully portable programs. Also, this function enables vendors to provide implementation specific events that are not defined in the PERUSE specification. The *event_names* and *events* arrays are maintained internally by the implementation and are guaranteed to be valid between PERUSE_Init and MPI_Finalize. The caller does not allocate or free any space for these arrays. The function always returns PERUSE_SUCCESS. In the case when the implementation does not provide any events, *num_supported* is set to 0, and both *event_names* and *events* output parameters are set to NULL.

5.2.3 PERUSE_Query_event

Synopsis

int PERUSE_Query_event(const char *event_name, int *event);

Input parameters

event_name – NULL terminated string containing the name of an event

Output parameters

event – event descriptor corresponding to *event_name*

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT

Description

This function is used for querying the PERUSE implementation about the support of the event described by *event_name* in a portable manner that will also facilitate vendor specific non-standard extensions. If the event specified by *event_name* is supported, the function returns PERUSE_SUCCESS and the output parameter *event* contains the corresponding event descriptor, which can then be passed to event handle constructors. If the event is not supported, the function returns PERUSE_ERR_EVENT and *event* is set to PERUSE_EVENT_INVALID. Any NULL terminated string can be passed as input value of *event_name*. The actual strings are implementation dependent. Suggested values of the *event_name* are the names of the constants defined in the PERUSE header file and as presented in this specification. This will improve code portability. For example, a query for the availability of PERUSE_COMM_REQ_XFER_BEGIN will look as follows:

rv = PERUSE_Query_event(“PERUSE_COMM_REQ_XFER_BEGIN”, &event).

If the return value *rv* is PERUSE_SUCCESS, the value of *event* will be set to the event descriptor for the event in question, which can be passed to PERUSE functions that accept event descriptors as an input parameter.

5.2.4 PERUSE_Query_event_name

Synopsis

int PERUSE_Query_event_name(int event, char **event_name);

Input parameters

event – event descriptor corresponding

Output parameters

event_name – a pointer to an internal string

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT

Description

This function is used for obtaining the string name of the input event. If the event is supported by the PERUSE implementation a pointer to string name representation of *event* is returned in *event_name*. This function is opposite of PERUSE_Query_event. PERUSE_ERR_EVENT is returned if *event* is invalid. The output is a pointer to an internally to the MPI library maintained string. The caller does not allocate memory for *event_name* and should not free the pointer returned. The pointer is guaranteed to be valid between PERUSE_Init and MPI_Finalize.

5.2.5 PERUSE_Query_environment

Synopsis

int PERUSE_Query_environment(int *env_size, char ***env)

Input parameters

n/a

Output parameters

env_size – number of elements in *env*
env – array of NULL terminated strings

Return value

PERUSE_SUCCESS

Description

This function provides the environment variables and their values that affect the behavior of the MPI library. The MPI-specific environment is returned through the *env* output parameter as an array of NULL terminated strings. Each string is of the form <env_var>=<value>. The output parameter *env_size* specifies the number of elements in *env*. All strings in *env* are allocated internally by the MPI library during MPI_Init and are guaranteed to exist until MPI_Finalize is called. If no the MPI library does not use any environment variables, env_size is set to 0 and env to NULL.

5.2.6 PERUSE_Query_queue_event_scope

Synopsis

int PERUSE_Query_queue_event_scope (int *scope)

Input parameters

n/a

Output parameters

scope – scope of queue events

Return value

PERUSE_SUCCESS

Description

This function provides information about the scope of queue events. Some MPI implementations keep only one pair of queues for posted and unexpected messages. Providing information on a per-communicator basis for these implementations may be complex and performance intrusive. These libraries may elect to provide information only for the global queue pair. In this case, the return value of scope will be PERUSE_GLOBAL. If queue events are generated on a per-communicator basis, the value of scope is set to PERUSE_PER_COMM. Other options are PERUSE_PER_TAG, PERUSE_PER_SOURCE and PERUSE_PER_PEER.

5.2.7 PERUSE_Event_comm_register

Synopsis

int PERUSE_Event_comm_register(int event, MPI_Comm comm,
peruse_comm_callback_t *callback_fn, void *param, peruse_event_h *event_h)

Input parameters

event – event descriptor
comm – valid MPI communicator handle
callback_fn – user callback
param – user-specific data

Output parameters

event_h – inactive event handle

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT, PERUSE_ERR_COMM, PERUSE_ERR_PARAMETER

Description

Used to create an event handle *event_h* related to an MPI communicator by associating the event descriptor *event* and communicator handle *comm*. The user callback *callback_fn* is registered with the output event handle. This callback function will be called when the MPI library performs an action that will affect the event described by *event_h*. If *callback_fn* is NULL, PERUSE_ERR_PARAMETER will be returned. PERUSE_ERR_EVENT will be returned if the input *event* descriptor is invalid, and PERUSE_ERR_COMM if the *comm* handle is invalid. PERUSE_ERR_COMM indicates that the MPI library would have returned MPI_ERR_COMM class if the user code tried to reference *comm* in an MPI call.

Note that PERUSE activation window permits overlap.

5.2.8 PERUSE_Event_activate

Synopsis

int PERUSE_Event_activate(peruse_event_h event_h)

Input parameters

event_h – event handle

Output parameters

n/a

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_MPI_OBJECT

Description

Opens an event activation window. The input event handle becomes active and the library will start invoking the user callback function registered with *event_h* every time that the library performs an activity which affects the event. If the input handle *event_h* has already been activated, the function will return PERUSE_SUCCESS. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter, the function returns PERUSE_ERR_EVENT_HANDLE. The return code PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

In a multi-tool scenario, PERUSE_Event_activate has global scope (effects all tools).

5.2.9 PERUSE_Event_deactivate

Synopsis

int PERUSE_Event_deactivate (peruse_event_h event_h)

Input parameters

event_h – event handle

Output parameters

n/a

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_MPI_OBJECT

Description

Closes an event activation window. As a result, the input event handle becomes inactive and the library will stop calling the user callback registered with *event_h*. If the input event handle *event_h* is inactive, the function has no effect and will return PERUSE_SUCCESS. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter, the function returns PERUSE_ERR_EVENT_HANDLE. The return code PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

5.2.10 PERUSE_Event_release

Synopsis

int PERUSE_Event_release(peruse_event_h *event_h)

Input parameters

event_h – event handle

Output parameters

event_h – invalid event handle (PERUSE_EVENT_HANDLE_NULL)

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_MPI_OBJECT

Description

Frees an active or inactive event handle and sets *event_h* to PERUSE_EVENT_HANDLE_NULL. Any subsequent uses of this event handle will lead to an error PERUSE_ERR_EVENT_HANDLE. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter, the function returns PERUSE_ERR_EVENT_HANDLE. The return code PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

5.2.11 PERUSE_Event_get

Synopsis

int PERUSE_Event_get(peruse_event_h mh, int *event)

Input parameters

Event_h – event handle

Output parameters

event – event descriptor that was used for event handle creation

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_MPI_OBJECT

Description

Performs a reverse lookup for discovering the event descriptor that was passed as an input parameter when the *event_h* event handle was created using the PERUSE_Event_xxx_register calls. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter, the function returns PERUSE_ERR_EVENT_HANDLE. The return code PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

5.2.12 PERUSE_Event_object_get

Synopsis

int PERUSE_Event_object_get(peruse_event_h mh, void **mpi_object)

Input parameters

event_h – event handle

Output parameters

mpi_object – an opaque handle of the MPI object to which this event handle is attached

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_MPI_OBJECT

Description

At return, the output *mpi_object* parameter contains the handle of the MPI object that was used when the input *event_h* event handle was created. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter, the function returns PERUSE_ERR_EVENT_HANDLE. Since the type of the output parameter is void*, the caller must know what kind of MPI object is expected in order to perform appropriate type casting. It is the user's responsibility to ensure the validity of the returned MPI handle. The returned handle is a copy of the original MPI handle passed to the specific PERUSE event initialization function. The return code PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed. For more details, see the section that treats the relationship between MPI handles and PERUSE event handles.

5.2.13 PERUSE_Event_comm_callback_set

Synopsis

int PERUSE_Event_comm_callback_set(peruse_event_h event_h,
peruse_comm_callback_t *callback_fn, void *param)

Input parameters

event_h – event handle
callback_fn – user defined callback function
param – user specific parameter that will be passed to the callback function

Output parameters

n/a

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_PARAMETER, PERUSE_ERR_MPI_OBJECT

Description

This function associates a user defined communicator *callback_fn* function with an inactive event handle *event_h*. The *event_h* and *param* input parameters will be passed to *callback_fn* when it is invoked. The old callback will be lost and only the callback registered with this call will be kept. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter or *event_h* is active, the function returns PERUSE_ERR_EVENT_HANDLE. If NULL is passed as *callback_fn*, PERUSE_ERR_PARAMETER will be returned. PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

5.2.14 PERUSE_Event_comm_callback_get

Synopsis

```
int PERUSE_Event_comm_callback_get(peruse_event_h event_h,
        peruse_comm_callback_t **callback_fn, void **param)
```

Input parameters

event_h – event handle

Output parameters

callback_fn – user defined callback function
param – user specific parameter that was passed to the callback function

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_MPI_OBJECT

Description

This function obtains the user defined callback function that is associated with the event handle *event_h*. The value of the output parameter *param* is the one passed in by the user when the callback was registered. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter or *mh* is active, the function returns PERUSE_ERR_EVENT_HANDLE.
PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

5.2.15 PERUSE_Event_propagate

Synopsis

```
int PERUSE_Event_propagate(peruse_event_h event_h, int mode)
```

Input parameters

Event_h – event handle
mode – propagation mode of event handle when the MPI object is duplicated

Output parameters

n/a

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_MPI_OBJECT

Description

This function sets a propagation mode of an inactive event handle *event_h*. The default propagation mode of all handles is 0 (false). This mode will remain unchanged unless is explicitly set by this function. If 1(true) is specified as input to this function, the callback registered with the event handle *event_h* will be propagated to MPI objects that are obtained as a result of duplication of the original MPI object with which the handle *event_h* was initially associated. If the propagation mode is turned on, the event handle callback will be invoked by operations performed on the duplicated MPI object in the same way as they are for the original MPI object. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter or *event_h* is active, the function returns PERUSE_ERR_EVENT_HANDLE. If *event_h* is

associated with an MPI object, which cannot be duplicated, the behavior of *event_h* and its callback will not be affected, i.e., PERUSE_Event_propagate is equivalent to NOOP in these cases. PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

5.2.16 PERUSE_Lock

Synopsis

int PERUSE_Lock()

Input parameters

n/a

Output parameters

n/a

Return value

PERUSE_SUCCESS, PERUSE_ERR_LOCK, PERUSE_LOCK_NOT_GRANTABLE

Description

This function provides for mutual exclusion in multi-threaded MPIs. In multi-threaded MPIs, it's use is required to guarantee correctness in the presence of multiple threads (e.g. updates on a linked list). If PERUSE_Lock has already been called by another thread, the calling thread blocks until the PERUSE Code mutex becomes available. This operation returns with the calling thread as its owner. Only lock free implementations of MPI_Wtime() and MPI_Wtick() are permitted within code segments guarded by PERUSE_Lock and PERUSE_Unlock. PERUSE_LOCK_NOT_GRANTABLE remains true during the scope (entire life) of the upcall that found it could not grant the lock.

> **Advice to PERUSE users:** A single "lock_not_grantable" leaves the state of the data from the tool dubious for the remainder of the job if the tool reacts by dropping the record. In such cases, the tool writer might want to set a flag of their own to record that fact. This may safely be accomplished by a flag, Records_lost, that is statically initialized to FALSE which it sets to TRUE if there is a "lock_not_grantable" return.

5.2.17 PERUSE_Unlock

Synopsis

int PERUSE_Unlock()

Input parameters

n/a

Output parameters

n/a

Return value

PERUSE_SUCCESS, PERUSE_ERR_LOCK

Description

This function releases the PERUSE code mutex. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, an error will be returned.. MPI calls are prohibited within code segments guarded by PERUSE_Lock and PERUSE_Unlock.

5.3 Semantics in multithreaded mode

The definition of PERUSE suggests close interaction between the code that implements PERUSE and the main MPI library code. Therefore, it is expected that PERUSE will operate in the same multithreaded mode as the entire MPI library. The consequence of this definition is that if the MPI library is not thread safe, the code that implements PERUSE does not need to be either. Further, if the MPI library works in a thread-safe mode, providing thread-safe PERUSE

functionality will not require separate thread safety mechanisms inside the PERUSE implementation. The user code in the event callbacks will need to take the necessary precautions to protect user-level shared structures that can be accessed from callbacks within different threads when the MPI implementation uses internal system (service) threads. PERUSE defers the thread environment initialization and management to the MPI library. The code that uses PERUSE will inquire about the thread safety of the interface through the mechanisms provided by the MPI library, specifically the *MPI_query_thread* call.

PERUSE provides the necessary mechanisms for multi-threaded MPI implementations through the PERUSE_Lock and PERUSE_Unlock functions. These calls provide a simple mechanism for guarding PERUSE internal code in a threaded MPI environment by mutexs. Note that only lock free implementations of MPI_Wtime() and MPI_Wtick() are permitted within code segments guarded by PERUSE_Lock and PERUSE_Unlock.

For signals-based multi-threaded MPI implementations, the use of MPI_Lock() and MPI_Unlock() ***alone*** is insufficient as a safety mechanism; the PERUSE user (parallel tool developer) will need to augment MPI_Lock() and MPI_Unlock() usage with any of the other standard safety techniques employed in signals-based multi-threaded contexts.

PERUSE does not maintain any special context for user threads. Similarly to MPI, the entire PERUSE functionality is defined on a per-process basis. This means that if a thread initializes an event handle, the PERUSE library will not distinguish whether the same or a different thread will subsequently manipulate the event handle. For example, one thread can initialize an event handle, a second thread can open the activation window of the handle, and a third thread can close the window and free the event. The user code is responsible for mitigating the access to the same event handle when there are both read and modify operations performed from different threads.

5.4 PERUSE and layered libraries

The experience of using PMPI has shown that in certain cases multiple layered MPI libraries can coexist and in such a scenario it is difficult to measure the performance of a specific layer without also including the effect of layers below. PERUSE has provisions that address this problem. First, there is no restriction on the number of calls to PERUSE_Init or their order. The only requirement is that PERUSE_Init is called after MPI_Init. This means that all layers can safely call PERUSE_Init without conflicts. PERUSE will be initialized by the first call and all others will be ignored. PERUSE does not need to be finalized, so this also enhances PERUSE behavior in multi-layered environment.

Another PERUSE feature for support of layered libraries is the capability to indicate that if a PERUSE event handle is attached to a communicator (MPI_Comm) and this communicator is duplicated, the event and its window status will be propagated to the copy of the original communicator. The function that indicates that the event handle will have this behavior is *PERUSE_Event_propagate*. This function has an input parameter, which indicates the desired mode of handle propagation when the MPI object associated with the handle is duplicated. The values of the parameter can be PERUSE_TRUE or PERUSE_FALSE.

It is expected that in layered libraries more than one user callback will need to be registered with a certain event at a time. Since PERUSE allows for multiple callbacks to be registered for a given {event, MPI object} pair, this enables different library layers to register their own callbacks without interfering with callbacks of other layers. The MPI library will invoke all registered callbacks in some undetermined order.

5.5 Querying PERUSE support options and MPI's run-time environment

PERUSE provides a set of *PERUSE_Query_xxx* functions that query different options related to PERUSE support capabilities and the MPI run-time environment. *PERUSE_Query_supported_events* is used to query the MPI library about all supported PERUSE events. *PERUSE_Query_event* and *PERUSE_Query_event_name* are used for retrieving event identifiers and event string names

PERUSE_Query_environment is intended to provide information about the MPI library run-time environment (environment variables) that affects the behavior of the MPI library. PERUSE_Query_queue_event_scope is intended to inform the user about the meaning of the queue-related event – whether they are on per communicator basis or are global for all communicators. This query is necessary to address MPI libraries that maintain only one global pair of queues for posted and unexpected requests.

5.6 Relationship between MPI handles and PERUSE event handles

PERUSE event handle constructors *PERUSE_Event_xxx_register* associate PERUSE events with MPI object handles by registering user PERUSE callbacks to the particular MPI object. Operations on PERUSE event handles are allowed only when the MPI handle associated with this event handle is valid, i.e., it is not released with *MPI_Xxx_free*. It is erroneous to use a PERUSE event handle after the associated MPI handle with this PERUSE event handle is released. The PERUSE calls should return PERUSE_ERR_MPI_OBJECT in such cases. It can be inferred from this definition of the MPI and PERUSE handle relationship that these associations are not treated as increments of the internal reference counts of the MPI objects.

During event callback registration, the constructor functions can return an error code of the type PERUSE_ERR_COMM. These error codes indicate that the MPI library would have returned an error class MPI_ERR_XXX if the user code had tried to use the corresponding MPI handle in an MPI call. Hence, it can be assumed that the error codes PERUSE_ERR_XXX are mapped to the corresponding MPI_ERR_XXX error classes.

6. External Interfaces

6.1 Target operating systems and platforms

The specification is intended to be independent of operating systems and hardware platforms.

6.2 Language bindings

At this stage, PERUSE provides only C bindings of its API. The C bindings of the API are shown in the example *peruse.h* file in Appendix A.

6.3 Library versions

The provider of PERUSE enabled MPI libraries could provide a debug and a production library. The debug library will implement the PERUSE interface while the production library will only provide dummy routines to satisfy unresolved externals. The goal is to avoid any impact on codes that are run in production mode and are not subjected to performance evaluation.

7. References

[1] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI–The Complete Reference: Volume 1, The MPI Core, 2nd edition*. MIT Press, Cambridge, MA, 1998.

[2] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI-The Complete Reference: Volume 2, THE MPI-2 Extensions.* MIT Press, Cambridge, MA, 1998

[3] MPI Software Technology Inc. *Performance Revealing Extensions Interface (PERUSE) version 1.2*

8. Appendix A: Example PERUSE Header File (peruse.h)

```
#ifndef _PERUSE_H_
#define _PERUSE_H_
#include <mpi.h>

/* PERUSE type declarations */
typedef long peruse_event_h;    /* Opaque event handle */
typedef struct _peruse_comm_spec_t
{
        MPI_Comm        comm;
        void            *buf;
        int             count;
        MPI_Datatype    datatype;
        int             peer;
        int             tag;
        int             operation;
} peruse_comm_spec_t;



typedef int (peruse_comm_callback_f)(peruse_event_h event_h,
                MPI_Aint unique_id, peruse_comm_spec_t *spec, void *param);


/* PERUSE constants */
enum
{
        PERUSE_SUCCESS,                 /* success */
        PERUSE_ERR_INIT,                /* PERUSE initialization failure */
        PERUSE_ERR_GENERIC,             /* generic unspecified error */
        PERUSE_ERR_MALLOC,              /* memory-related error */
        PERUSE_ERR_EVENT,               /* invalid event descriptor */
        PERUSE_ERR_EVENT_HANDLE,        /* invalid event handle */
        PERUSE_ERR_PARAMETER,           /* invalid input parameter */
        PERUSE_ERR_MPI_INIT,            /* MPI has not been initializes */
        PERUSE_ERR_COMM,                /* MPI_ERR_COMM class */
PERUSE_ERR_MPI_OBJECT           /* error with associated MPI object */
      PERUSE_ERR_LOCK,              /* error associated with PERUSE_Lock() */
      PERUSE_ERR_UNLOCK,            /* error associated with PERUSE_Unlock()
*/
      PERUSE_ERR_LOCK_NOT_GRANTABLE /* error: unable to grant PERUSE lock */
};

enum
{
        PERUSE_EVENT_INVALID,

        /* Point-to-point request events */
        PERUSE_COMM_REQ_ACTIVATE,
        PERUSE_COMM_REQ_MATCH_UNEX,
        PERUSE_COMM_REQ_INSERT_IN_POSTED_Q,
        PERUSE_COMM_REQ_REMOVE_FROM_POSTED_Q,
        PERUSE_COMM_REQ_XFER_BEGIN,
        PERUSE_COMM_REQ_XFER_END,
```

```
        PERUSE_COMM_REQ_COMPLETE,
        PERUSE_COMM_REQ_NOTIFY,
        PERUSE_COMM_MSG_ARRIVED,
        PERUSE_COMM_MSG_INSERT_IN_UNEX_Q,
        PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q,
        PERUSE_COMM_MSG_MATCH_POSTED_REQ,

        /* Queue events */
        PERUSE_COMM_SEARCH_POSTED_Q_BEGIN,
        PERUSE_COMM_SEARCH_POSTED_Q_END,
        PERUSE_COMM_SEARCH_UNEX_QUEUE_BEGIN,
        PERUSE_COMM_SEARCH_UNEX_Q_END,

        /* Collective events */
        /* IO events */
        /* One-sided events */
        PERUSE_FIRST_CUSTOM_EVENT
};

/* Scope of message queues */
enum
{
        PERUSE_PER_COMM,
        PERUSE_PER_TAG,
        PERUSE_PER_SOURCE,
        PERUSE_GLOBAL
};

enum
{
        PERUSE_SEND,
        PERUSE_RECV,
        PERUSE_PUT,
        PERUSE_GET,
        PERUSE_ACC,
        PERUSE_IO_READ,
        PERUSE_IO_WRITE
};

#define PERUSE_EVENT_HANDLE_NULL ((peruse_event_h)0)

/*
 * I. Environment
 */
/* PERUSE initialization */
int PERUSE_Init();

/* Query all implemented events */
int PERUSE_Query_supported_events(
                int     *num_supported,
                char    ***event_names,
                int     **events);

/* Query supported events */
int PERUSE_Query_event(const char *event_name, int *event);

/* Query event name */
```

```
int PERUSE_Query_event_name(int event, char **event_name);

/* Get environment variables that affect MPI library behavior */
int PERUSE_Query_environment(int *env_size, char ***env);

/* Queryig the scope of queue metircs - global or per communicator */
int PERUSE_Query_queue_event_scope(int *scope);

/* Acquire PERUSE code mutex */
int PERUSE_Lock();

/* Release PERUSE code mutex */
int PERUSE_Unlock();


/*
 * II. Events objects initialization and manipulation
 */
/* Initialize event associated with an MPI communicator */
int PERUSE_Event_comm_register(
                int                     event,
                MPI_Comm                comm,
                peruse_comm_callback_f  *callback_fn,
                void                    *param,
                peruse_event_h          *event_h);


/* Start collecting data (activate event) */
int PERUSE_Event_activate(peruse_event_h event_h);

/* Stop collecting data (deactivate event) */
int PERUSE_Event_deactivate(peruse_event_h event_h);

/* Free event handle */
int PERUSE_Event_release(peruse_event_h *event_h);

/* Set a new comm callback */
int PERUSE_Event_comm_callback_set(
                peruse_event_h          event_h,
                peruse_comm_callback_f  *callback_fn,
                void                    *param);


/* Get the current comm callback */
int PERUSE_Event_comm_callback_get(
                peruse_event_h          event_h,
                peruse_comm_callback_f  **callback_fn,
                void                    **param);



/* Obtain event descriptor from a event handle (reverse lookup) */
int PERUSE_Event_get(peruse_event_h event_h, int *event);

/* Obtain MPI object associated with event handle */
int PERUSE_Event_object_get(peruse_event_h event_h, void **mpi_object);
```

```
/* Propagation mode */
int PERUSE_Event_propagate(peruse_event_h event_h, int mode);

#endif
```

9. Appendix B: PERUSE Examples

9.1 Examples of instrumented user MPI programs

9.1.1 Using environment, event, and queue event scope queries

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "peruse.h"

char *ename[] =
{
        "PERUSE_COMM_REQ_ACTIVATE",
        "PERUSE_COMM_REQ_MATCH_UNEX",
        "PERUSE_COMM_REQ_INSERT_IN_POSTED_Q",
        "PERUSE_COMM_REQ_REMOVE_FROM_POSTED_Q",
        "PERUSE_COMM_REQ_XFER_BEGIN",
        "PERUSE_COMM_REQ_XFER_END",
        "PERUSE_COMM_REQ_COMPLETE",
        "PERUSE_COMM_REQ_NOTIFY",
        "PERUSE_COMM_MSG_ARRIVED",
        "PERUSE_COMM_MSG_INSERT_IN_UNEX_Q",
        "PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q",
        "PERUSE_COMM_MSG_MATCH_POSTED_REQ",
        NULL
};

int main(int argc, char **argv)
{
        int eid, *eids;
        char **env, **names;
        int rv, size, i, scope = -1, n_sup;

        MPI_Init(&argc, &argv);
        PERUSE_Init();
        PERUSE_Query_environment(&size, &env);
        printf("Number of env. variables: %d\n", size);
        for(i = 0; i < size; i++)
                printf("%s\n", env[i]);

        PERUSE_Query_supported_events(&n_sup, &names, &eids);
        printf("Number of supported events: %d\n", n_sup);
        for(i = 0; i < n_sup; i++)
                printf("%s=%d\n", names[i], eids[i]);

        PERUSE_Query_queue_event_scope(&scope);
        printf("SCOPE=%s\n", (scope == PERUSE_PER_COMM) ?
                "PER_COMM" : "GLOBAL");

        for(i = 0; ename[i] != NULL; i++)
        {
                rv = PERUSE_Query_event(ename[i], &eid);
                printf("event=%s, event ID=%d is %s\n",
                        ename[i], eid,
                        (rv == PERUSE_SUCCESS) ? "supported":"unsupported");

        }

        MPI_Finalize();

        return 0;
}
```

9.1.2 Using callbacks

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "peruse.h"

#define NUM_MSG 100
#define MSG_SIZE 160

typedef struct
{
        int num_stamps;
        double time;
        double max;
} measure_t;

typedef struct _hash_elem_t
{
        MPI_Aint key;
        double stamp;
        struct _hash_elem_t *next;
} hash_elem_t;

hash_elem_t *HashGetElem(MPI_Aint key)
{
        hash_elem_t *h_elem;
        /* Use some hash function to find an existing element with key
         * in the hash table or allocate a new element */
        return h_elem;
}

int callback_unex(peruse_event_h event_h, MPI_Aint unique_id,
                        peruse_comm_spec_t *cs, void *param)
{
        measure_t *mt = (measure_t *)param;
        char *event_name;
        int event;
        double t;
        hash_elem_t *helem;

        PERUSE_Event_get(event_h, &event);
        PERUSE_Query_event_name(event, &event_name);
        printf("Callback called for event %s\n", event_name);

        helem = HashGetElem(unique_id);
        switch(event)
        {
        case PERUSE_COMM_MSG_INSERT_IN_UNEX_Q:
                helem->stamp = MPI_Wtime();
                break;

        case PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q:
                t = MPI_Wtime() - helem->stamp;
                mt->time += t;
                if(t > mt->max)
                        mt->max = t;
                mt->num_stamps++;
                break;

        default:
                printf("Unexpected event in callback\n");
```

```
                return MPI_ERR_INTERN;
        }
        return MPI_SUCCESS;
}

int msg[MSG_SIZE];

int main(int argc, char **argv)
{
        peruse_event_h e_unex_insert, e_unex_remove;
        int rv, size, i, rank, stat;
        MPI_Comm wrld = MPI_COMM_WORLD;
        MPI_Status status;
        MPI_Request r[NUM_MSG];
        measure_t unex = {0, 0.0, 0.0};

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        rv = PERUSE_Init();
        if(rv != PERUSE_SUCCESS)
        {
                printf("Error in PERUSE_Init: rv=%d\n", rv);
                fflush(stdout);
                exit(1);
        }

        /* HashTableSetup(); */
        if(rank == 0)
        {
                /* Interested only in rank 0 */
                PERUSE_Event_comm_register(PERUSE_COMM_MSG_INSERT_IN_UNEX_Q,
                        wrld, callback_unex, &unex, &e_unex_insert);
                PERUSE_Event_comm_register(PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q,
                        wrld, callback_unex, &unex, &e_unex_remove);
                PERUSE_Event_activate(e_unex_insert);
                PERUSE_Event_activate(e_unex_remove);

                for(i = 0; i < NUM_MSG; i++)
                        MPI_Irecv(msg, MSG_SIZE, MPI_INT, 1, 0, wrld, &r[i]);

                MPI_Send(NULL, 0, MPI_INT, 1, 0, wrld);

                for(i = 0; i < NUM_MSG; i++)
                        MPI_Wait(&r[i], &status);

                PERUSE_Event_deactivate(e_unex_insert);
                PERUSE_Event_deactivate(e_unex_remove);

                printf("Number of measurements: %d\n", unex.num_stamps);
                printf("Average time in unexpected queue: %f sec\n",
                                unex.time / unex.num_stamps);
                printf("maximum time in unexpected queue: %f sec\n",
                                unex.max);
                PERUSE_Event_release(&e_unex_insert);
                PERUSE_Event_release(&e_unex_remove);
        }
        else if (rank == 1)
        {
                for(i = 0; i < NUM_MSG; i++)
                {
                        MPI_Send(msg, MSG_SIZE, MPI_INT, 0, 0, wrld);
                        if(i == NUM_MSG / 2)
```

```
                                MPI_Recv(NULL, 0, MPI_INT, 0, 0, wrld, &status);
                }
        }

        MPI_Finalize();
        /* HashTableCleanup(); */

        return 0;
}
```

9.1.3 Using queue events

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "peruse.h"

#define NUM_MSG          100
#define MSG_SIZE         160
#define NUM_Q_EVENTS    4

int qevents[NUM_Q_EVENTS] =
{
        PERUSE_COMM_SEARCH_POSTED_Q_BEGIN,
        PERUSE_COMM_SEARCH_POSTED_Q_END,
        PERUSE_COMM_SEARCH_UNEX_QUEUE_BEGIN,
        PERUSE_COMM_SEARCH_UNEX_Q_END
};

double time_in_unex_q = 0.0, time_in_posted_q = 0.0;

int callback(peruse_event_h event_h, MPI_Aint unique_id,
                        peruse_comm_spec_t *cs, void *param)
{
        int event;

        PERUSE_Event_get(event_h, &event);
        switch(event)
        {
        case PERUSE_COMM_SEARCH_POSTED_Q_BEGIN:
                /* Take a time stamp for unique_id */
                break;

        case PERUSE_COMM_SEARCH_POSTED_Q_END:
                /* Take a time stamp for unique_id, substract
                 * previous time stamp, and add to time_in_posted_q */
                break;

        case PERUSE_COMM_SEARCH_UNEX_QUEUE_BEGIN:
                /* Take a time stamp for unique_id */
                break;

        case PERUSE_COMM_SEARCH_UNEX_Q_END:
                /* Take a time stamp for unique_id, substract
                 * previous time stamp, and add to time_in_unex_q */
                break;

        default:
                printf("Unexpected event\n");
                return MPI_ERR_INTERN;
        }

        return MPI_SUCCESS;
```

```
}

int msg[MSG_SIZE];

int main(int argc, char **argv)
{
        peruse_event_h eh[NUM_Q_EVENTS];
        int rv, size, i, rank, stat;
        MPI_Comm wrld = MPI_COMM_WORLD;
        MPI_Status status;
        MPI_Request r[NUM_MSG];

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        rv = PERUSE_Init();
        if(rv != PERUSE_SUCCESS)
        {
                printf("Error in PERUSE_Init: rv=%d\n", rv);
                fflush(stdout);
                exit(1);
        }

        rv = PERUSE_SUCCESS;
        for(i = 0; i < NUM_Q_EVENTS; i++)
                rv |= PERUSE_Event_comm_register(qevents[i], wrld,
                        callback, NULL, &eh[i]);

        if(rv != PERUSE_SUCCESS)
        {
                printf("Cannot register events\n");
                fflush(stdout);
                MPI_Finalize();
                exit(1);
        }

        MPI_Barrier(wrld);

        if(rank == 0)
        {
                for(i = 0; i < NUM_Q_EVENTS; i++)
                        PERUSE_Event_activate(eh[i]);

                for(i = 0; i < NUM_MSG; i++)
                        MPI_Irecv(msg, MSG_SIZE, MPI_INT, 1, 0, wrld, &r[i]);

                MPI_Send(NULL, 0, MPI_INT, 1, 0, wrld);

                for(i = 0; i < NUM_MSG; i++)
                        MPI_Wait(&r[i], &status);

                for(i = 0; i < NUM_Q_EVENTS; i++)
                        PERUSE_Event_deactivate(eh[i]);
        }
        else if (rank == 1)
        {
                for(i = 0; i < NUM_MSG; i++)
                {
                        MPI_Send(msg, MSG_SIZE, MPI_INT, 0, 0, wrld);
                        if(i == NUM_MSG / 4)
                                MPI_Recv(NULL, 0, MPI_INT, 0, 0, wrld, &status);
                }
        }
```

```
        for(i = 0; i < NUM_Q_EVENTS; i++)
                PERUSE_Event_release(&eh[i]);

        MPI_Finalize();

        return 0;
}
```

9.1.4 Counting posted and unexpected receives

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "peruse.h"

#define QEVENTS (4)

/* Queue events */
char *qevents[QEVENTS] =
{
        "PERUSE_COMM_REQ_INSERT_IN_POSTED_Q",
        "PERUSE_COMM_REQ_REMOVE_FROM_POSTED_Q",
        "PERUSE_COMM_MSG_INSERT_IN_UNEX_Q",
        "PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q"
};

/* Declaration of a type for collecting statistics */
typedef struct _measure_t
{
        int unex_num;
        int unex_len;
        int unex_max_len;
        int unex_ave_len;
        int posted_num;
        int posted_len;
        int posted_max_len;
        int posted_ave_len;
} measure_t;

measure_t *darr;
int np, my_rank;

/* Callback for collecting statistics */
int qcallback(peruse_event_h eh, MPI_Aint unique_id,
                        peruse_comm_spec_t *spec, void *param)
{
        double t;
        int event;
        measure_t *mt;

        PERUSE_Event_get(eh, &event);
        mt = &darr[spec->peer]; /* get the data element for the peer */

        switch(event)
        {
        case PERUSE_COMM_REQ_INSERT_IN_POSTED_Q:
                mt->posted_num++;
                mt->posted_len++;
                if(mt->posted_len > mt->posted_max_len)
                        mt->posted_max_len = mt->posted_len;
                mt->posted_ave_len =
                        ((mt->posted_num - 1) * mt->posted_ave_len +
                                mt->posted_len) / mt->posted_num;
                break;
```

```
        case PERUSE_COMM_REQ_REMOVE_FROM_POSTED_Q:
                mt->posted_len--;
                break;

        case PERUSE_COMM_MSG_INSERT_IN_UNEX_Q:
                mt->unex_num++;
                mt->unex_len++;
                if(mt->unex_len > mt->unex_max_len)
                        mt->unex_max_len = mt->unex_len;
                mt->unex_ave_len =
                        ((mt->unex_num - 1) * mt->unex_ave_len +
                                mt->unex_len) / mt->unex_num;
                break;

        case PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q:
                mt->unex_len--;
                break;

        default:
                printf("Unexpected event in callback\n");
                return MPI_ERR_INTERN;
        }

        return MPI_SUCCESS;
}

void UserMpiProcessing(){}

int main(int argc, char **argv)
{
        int i, rv, eid[QEVENTS];
        peruse_event_h eh[QEVENTS];

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &np);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        /* Initialize PERUSE */
        rv = PERUSE_Init();
        if(rv != PERUSE_SUCCESS)
        {
                printf("Unable to initialize PERUSE\n");
                return 1;
        }

        darr = (measure_t *)calloc(np, sizeof(measure_t));
        /* Initialize queue event handles and activate them */
        for(i = 0; i < QEVENTS; i++)
        {
                PERUSE_Query_event(qevents[i], &eid[i]);
                PERUSE_Event_comm_register(eid[i], MPI_COMM_WORLD, qcallback,
                                                NULL, &eh[i]);
                PERUSE_Event_activate(eh[i]);
        }

        /* User code subjected to PERUSE evaluation */
        UserMpiProcessing();

        /* Deactivate event handles and free them */
        for(i = 0; i < QEVENTS; i++)
        {
                PERUSE_Event_deactivate(eh[i]);
                PERUSE_Event_release(&eh[i]);
        }
```

```
        /* Report results */
        for(i = 0; i < np; i++)
        {
                printf("===== Peer rank: %d =====\n", i);
                printf("number of unexpected messages: %d\n",
                                        darr[i].unex_num);
                printf("max unexpected queue length: %d\n",
                                        darr[i].unex_max_len);
                printf("ave unexpected queue length: %d\n",
                                        darr[i].unex_ave_len);
                printf("number of posted receives    : %d\n",
                                        darr[i].posted_num);
                printf("max posted queue length: %d\n",
                                        darr[i].posted_max_len);
                printf("ave posted queue length: %d\n",
                                        darr[i].posted_ave_len);
        }
        free(darr);

        return 0;
}
```

9.2 Example performance profiler code

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "peruse.h"

#define NUM_REQ_EVENTS  (12)

/* Events for the occurrence of which the profiler will be notified */
char *comm_events[NUM_REQ_EVENTS] =
{
        "PERUSE_COMM_REQ_ACTIVATE",
        "PERUSE_COMM_REQ_MATCH_UNEX",
        "PERUSE_COMM_REQ_INSERT_IN_POSTED_Q",
        "PERUSE_COMM_REQ_REMOVE_FROM_POSTED_Q",
        "PERUSE_COMM_REQ_XFER_BEGIN",
        "PERUSE_COMM_REQ_XFER_END",
        "PERUSE_COMM_REQ_COMPLETE",
        "PERUSE_COMM_REQ_NOTIFY",
        "PERUSE_COMM_MSG_ARRIVED",
        "PERUSE_COMM_MSG_INSERT_IN_UNEX_Q",
        "PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q",
        "PERUSE_COMM_MSG_MATCH_POSTED_REQ"
};

/* Declaration of a type for collecting statistics */
/* The statistics are accumulated over all communicators */
typedef struct _req_estamp_t
{
        double req_activate;
        double req_match_unex;
        double req_posted_q_in;
        double req_posted_q_out;
        double req_xfer_begin;
        double req_xfer_end;
        double req_complete;
        double req_notify;
        double msg_arrived;
        double msg_unex_q_in;
        double msg_unex_q_out;
        double msg_match_posted;
} req_estamp_t;
```

```
typedef struct _pevent_t
{
        int eid;
        peruse_event_h *eh_arr;
} pevent_t;

typedef struct _measure_t
{
        char *name;
        int num;
        double ave;
        double max;
        double len;
} measure_t;


/* Define metrics of interest based on the pre-defined events */
enum
{
        T_REQ_ACTIVATE_TO_MATCH = 0,
        T_REQ_ACTIVATE_TO_XFER_BEGIN,
        T_XFER_BEGIN_TO_END,
        T_COMPLETE_TO_NOTIFY,
        T_ACTIVATE_TO_NOTIFY,
        T_IN_UNEX_Q,
        N_POSTED_Q_LEN,
        N_UNEX_Q_LEN,
        NUM_METRICS
};

char *metric_names[] = {
        "T_REQ_ACTIVATE_TO_MATCH", "T_REQ_ACTIVATE_TO_XFER_BEGIN",
        "T_XFER_BEGIN_TO_END", "T_COMPLETE_TO_NOTIFY", "T_ACTIVATE_TO_NOTIFY",
        "T_IN_UNEX_Q", "N_POSTED_Q_LEN","N_UNEX_Q_LEN", NULL};

/* Hash Table for request unique_id */
#define HASH_TABLE_SIZE 256
#define HASH_FUNC(_key_) ((_key_) % HASH_TABLE_SIZE)

typedef struct _hash_elem_t
{
        MPI_Aint key;
        req_estamp_t stamp;
        struct _hash_elem_t *next;
} hash_elem_t;

int HashTableSetup();
void HashTableCleanup();
req_estamp_t *HashTableFindOrInsert(MPI_Aint key);
void HashTableRemove(MPI_Aint key);

int InitiEvents(MPI_Comm comm, peruse_comm_callback_f *callback);
int CleanupEvents(int comm_idx);
void ComputeTimeMetric(measure_t *mt, double time);
void ComputeCounterMetric(measure_t *mt);
void print_stat(measure_t *mt);

int np, my_rank, num_comms = 0;
MPI_Comm *comm_arr = NULL;
pevent_t events[NUM_REQ_EVENTS];
measure_t metrics[NUM_METRICS];
hash_elem_t **HashTable;

/* Callback for collecting statistics */
```

```
int comm_callback(peruse_event_h event_h, MPI_Aint unique_id,
                        peruse_comm_spec_t *spec, void *param)
{
        measure_t *mt;
        req_estamp_t *estamp;
        int event;
        double t;

        /* Assume that we are only interested in point to point
         * communication to/from remote ranks */
        if(spec->peer == my_rank)
                return MPI_SUCCESS;

        PERUSE_Event_get(event_h, &event);
        estamp = HashTableFindOrInsert(unique_id);
        switch(event)
        {
        case PERUSE_COMM_REQ_ACTIVATE:
                estamp->req_activate = PMPI_Wtime();
                break;

        case PERUSE_COMM_REQ_MATCH_UNEX:
                estamp->req_match_unex = PMPI_Wtime();
                mt = &metrics[T_REQ_ACTIVATE_TO_MATCH];
                t = estamp->req_match_unex - estamp->req_activate;
                ComputeTimeMetric(mt, t);
                break;

        case PERUSE_COMM_REQ_INSERT_IN_POSTED_Q:
                estamp->req_posted_q_in = PMPI_Wtime();
                mt = &metrics[N_POSTED_Q_LEN];
                ComputeCounterMetric(mt);
                break;

        case PERUSE_COMM_REQ_REMOVE_FROM_POSTED_Q:
                estamp->req_posted_q_out = PMPI_Wtime();
                mt = &metrics[N_POSTED_Q_LEN];
                mt->len--;
                break;

        case PERUSE_COMM_REQ_XFER_BEGIN:
                estamp->req_xfer_begin = PMPI_Wtime();
                mt = &metrics[T_REQ_ACTIVATE_TO_XFER_BEGIN];
                t = estamp->req_xfer_begin - estamp->req_activate;
                ComputeTimeMetric(mt, t);
                break;

        case PERUSE_COMM_REQ_XFER_END:
                estamp->req_xfer_end = PMPI_Wtime();
                mt = &metrics[T_XFER_BEGIN_TO_END];
                t = estamp->req_xfer_end - estamp->req_xfer_begin;
                ComputeTimeMetric(mt, t);
                break;

        case PERUSE_COMM_REQ_COMPLETE:
                estamp->req_complete = PMPI_Wtime();
                break;

        case PERUSE_COMM_REQ_NOTIFY:
                estamp->req_notify = PMPI_Wtime();
                mt = &metrics[T_COMPLETE_TO_NOTIFY];
                t = estamp->req_notify - estamp->req_complete;
                ComputeTimeMetric(mt, t);
                mt = &metrics[T_ACTIVATE_TO_NOTIFY];
                t =  estamp->req_notify - estamp->req_activate;
```

```
                ComputeTimeMetric(mt, t);
                break;

        case PERUSE_COMM_MSG_ARRIVED:
                estamp->msg_arrived = PMPI_Wtime();
                break;

        case PERUSE_COMM_MSG_INSERT_IN_UNEX_Q:
                estamp->msg_unex_q_in = PMPI_Wtime();
                mt = &metrics[N_UNEX_Q_LEN];
                ComputeCounterMetric(mt);
                break;

        case PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q:
                estamp->msg_unex_q_out = PMPI_Wtime();
                mt = &metrics[T_IN_UNEX_Q];
                t = estamp->msg_unex_q_out - estamp->msg_unex_q_in;
                ComputeTimeMetric(mt, t);
                mt = &metrics[N_UNEX_Q_LEN];
                mt->len--;
                break;

        case PERUSE_COMM_MSG_MATCH_POSTED_REQ:
                estamp->msg_match_posted = PMPI_Wtime();
                break;

        default:
                printf("Unexpected event in callback\n");
                return MPI_ERR_INTERN;
        }

        /* If the event is last for teh request, release the hash element */
        if(event == PERUSE_COMM_REQ_NOTIFY ||
           event == PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q ||
           event == PERUSE_COMM_MSG_MATCH_POSTED_REQ)
        {
                HashTableRemove(unique_id);
        }

        return MPI_SUCCESS;
}

/* Profiler functions */
int MPI_Init(int *argc, char ***argv)
{
        int i, rv, eid;
        peruse_event_h eh;

        PMPI_Init(argc, argv);
        PMPI_Comm_size(MPI_COMM_WORLD, &np);
        PMPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

        /* Initialize PERUSE */
        rv = PERUSE_Init();
        if(rv != PERUSE_SUCCESS)
        {
                printf("Unable to initialize PERUSE\n");
                return MPI_ERR_INTERN;
        }

        HashTableSetup();
        memset(metrics, 0, NUM_METRICS * sizeof(measure_t));
        for(i = 0; i < NUM_METRICS; i++)
                metrics[i].name = metric_names[i];
```

```
        /* Query PERUSE to see if the events of interest are supported */
        for(i = 0; i < NUM_REQ_EVENTS; i++)
                PERUSE_Query_event(comm_events[i], &events[i].eid);

        return InitEvents(MPI_COMM_WORLD, comm_callback);
}

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
{
        int rv;

        rv = PMPI_Comm_create(comm, group, newcomm);
        if(rv != MPI_SUCCESS)
                return rv;

        return InitEvents(*newcomm, comm_callback);
}

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
{
        int rv;

        rv = PMPI_Comm_dup(comm, newcomm);
        if(rv != MPI_SUCCESS)
                return rv;

        return InitEvents(*newcomm, comm_callback);
}

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
{
        int rv;

        rv = PMPI_Comm_split(comm, color, key, newcomm);
        if(rv != MPI_SUCCESS)
                return rv;

        return InitEvents(*newcomm, comm_callback);
}

int MPI_Comm_free(MPI_Comm *comm)
{
        int i, comm_idx, rv;
        MPI_Comm cm = *comm;

        rv = PMPI_Comm_free(comm);
        if(rv != MPI_SUCCESS)
                return rv;

        for(i = 0; i < num_comms; i++)
                if(cm == comm_arr[i])
                        comm_idx = i;
        return CleanupEvents(comm_idx);
}

int MPI_Finalize()
{
        int i;

        /* Deactivate event handles and free them for all comms */
        for(i = 0; i < num_comms; i++)
                CleanupEvents(i);

        for(i = 0; i < NUM_REQ_EVENTS; i++)
        {
```

```
                if(events[i].eid == PERUSE_EVENT_INVALID)
                {
                        printf("Event %s not supported\n", comm_events[i]);
                        continue;
                }
                free(events[i].eh_arr);
        }

        /* Print stiatistics for all metrics */
        for(i = 0; i < NUM_METRICS; i++)
                print_stat(&metrics[i]);

        HashTableCleanup();
        free(events);
        free(comm_arr);

        return PMPI_Finalize();
}

/* Support functions */
int InitEvents(MPI_Comm comm, peruse_comm_callback_f *callback)
{
        int i;
        peruse_event_h eh;

        /* Initialize event handles with comm and activate them */
        num_comms++;
        comm_arr = (MPI_Comm *)realloc(comm_arr, num_comms * sizeof(MPI_Comm));
        comm_arr[num_comms - 1] = comm;
        for(i = 0; i < NUM_REQ_EVENTS; i++)
        {
                PERUSE_Event_comm_register(events[i].eid, comm,
                                                callback, NULL, &eh);
                events[i].eh_arr = (peruse_event_h *)realloc(events[i].eh_arr,
                                        num_comms * sizeof(peruse_event_h));
                events[i].eh_arr[num_comms - 1] = eh;
                PERUSE_Event_activate(eh);
        }

        return MPI_SUCCESS;
}

int CleanupEvents(int comm_idx)
{
        int i;

        for(i = 0; i < NUM_REQ_EVENTS; i++)
        {
                PERUSE_Event_deactivate(events[i].eh_arr[comm_idx]);
                PERUSE_Event_release(&events[i].eh_arr[comm_idx]);
        }
}

void ComputeTimeMetric(measure_t *mt, double time)
{
        if(time > mt->max)
                mt->max = time;
        mt->ave = (mt->num * mt->ave + time) / (mt->num + 1);
        mt->num++;
}

void ComputeCounterMetric(measure_t *mt)
{
        mt->len++;
        if(mt->len > mt->max)
```

```
                mt->max = mt->len;
        mt->ave = (mt->num * mt->ave) / (mt->num + 1);
        mt->num++;
}

void print_stat(measure_t *mt)
{
        printf("metric: %s\n", mt->name);
        printf("        number of measurements: %d\n", mt->num);
        printf("        average               : %f\n", mt->ave);
        printf("        maximum               : %f\n", mt->max);
}

/* Hash Table interface */
int HashTableSetup()
{
        HashTable = (hash_elem_t **)calloc(
                        HASH_TABLE_SIZE, sizeof(hash_elem_t *));
        return (HashTable) ? 1 : 0;
}

void HashTableCleanup()
{
        int i;
        hash_elem_t *he, *oe;

        for(i = 0; i < HASH_TABLE_SIZE; i++)
        {
                for(he = HashTable[i]; he != NULL; )
                {
                        oe = he;
                        he = he->next;
                        free(oe);
                }
        }
        free(HashTable);
}

req_estamp_t *HashTableFindOrInsert(MPI_Aint key)
{
        hash_elem_t *he, *pe = NULL;
        int idx = HASH_FUNC(key);

        for(he = HashTable[idx]; he != NULL && he->key != key; )
        {
                pe = he;
                he = he->next;
        }
        if(he != NULL && he->key == key)
                return &he->stamp;

        /* Did not find the entry; make a new one */
        he = (hash_elem_t *)calloc(1, sizeof(hash_elem_t));
        he->key = key;
        if(pe == NULL)
                HashTable[idx] = he;
        else
                pe->next = he;

        return &he->stamp;
}

void HashTableRemove(MPI_Aint key)
{
        hash_elem_t *he, *pe = NULL;
```

```
        int idx = HASH_FUNC(key);

        for(he = HashTable[idx]; he != NULL && he->key != key; )
        {
                pe = he;
                he = he->next;
        }
        if(he == NULL)  /* Not found */
                return;
        if(pe == NULL)
                HashTable[idx] = NULL;
        else
                pe->next = NULL;

        free(he);
}
```

10. Appendix C: PERUSE API FUNCTIONS

```
PERUSE_Init
PERUSE_Query_supported_events
PERUSE_Query_event
PERUSE_Query_event_name
PERUSE_Query_environment
PERUSE_Query_queue_event_scope
PERUSE_Event_comm_register
PERUSE_Event_activate
PERUSE_Event_deactivate
PERUSE_Event_release
PERUSE_Event_comm_callback_set
PERUSE_Event_comm_callback_get
PERUSE_Event_get
PERUSE_Event_object_get
PERUSE_Eventpropagate
PERUSE_Lock
PERUSE_Unlock
```

11. Appendix D: PERUSE CONSTANTS

```
PERUSE_SUCCESS             /* Error code: success */
PERUSE_ERR_INIT            /* Error code: PERUSE initialization failure */
PERUSE_ERR_GENERIC         /* Error code: generic unspecified error */
PERUSE_ERR_MALLOC          /* Error code: memory-related error */
PERUSE_ERR_EVENT           /* Error code: invalid event descriptor */
PERUSE_ERR_EVENT_HANDLE    /* Error code: invalid event handle */
PERUSE_ERR_PARAMETER       /* Error code: invalid input parameter */
PERUSE_ERR_MPI_INIT        /* Error code: MPI has not been initializes */
PERUSE_ERR_COMM            /* Error code: MPI_ERR_COMM class */
PERUSE_ERR_MPI_OBJECT      /* Error code: error with associated MPI object */
PERUSE_ERR_LOCK            /* Error code: error associated with PERUSE_Lock */
PERUSE_ERR_UNLOCK          /* Error code: error associated with PERUSE_Unlock */
PERUSE_ERR_LOCK_NOT_GRANTABLE /* Error code: unable to grant PERUSE lock */

PERUSE_COMM_REQ_ACTIVATE
PERUSE_COMM_REQ_MATCH_UNEX
PERUSE_COMM_REQ_INSERT_IN_POSTED_Q
PERUSE_COMM_REQ_REMOVE_FROM_POSTED_Q
PERUSE_COMM_REQ_XFER_BEGIN
PERUSE_COMM_REQ_XFER_END
PERUSE_COMM_REQ_COMPLETE
PERUSE_COMM_REQ_NOTIFY
PERUSE_COMM_MSG_ARRIVED
PERUSE_COMM_MSG_INSERT_IN_UNEX_Q
PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q
PERUSE_COMM_MSG_MATCH_POSTED_REQ
PERUSE_COMM_SEARCH_POSTED_Q_BEGIN
PERUSE_COMM_SEARCH_POSTED_Q_END
PERUSE_COMM_SEARCH_UNEX_QUEUE_BEGIN
PERUSE_COMM_SEARCH_UNEX_Q_END
PERUSE_FIRST_CUSTOM_EVENT

PERUSE_PER_COMM
PERUSE_PER_TAG
PERUSE_PER_SOURCE
PERUSE_GLOBAL

PERUSE_SEND
PERUSE_RECV
PERUSE_PUT
PERUSE_GET
PERUSE_ACC
PERUSE_IO_READ
PERUSE_IO_WRITE

PERUSE_EVENT_HANDLE_NULL
```

12. Appendix E: PROPOSED ADDITIONS TO PERUSE RETAINED FOR FUTURE VERSIONS

12.1 Collective communication metrics (MPI_Comm)

PERUSE attempts to provide additional detail in MPI collective operations. Some of the PERUSE metrics refer to individual send and receive operations. These operations are the primitive point-to-point or collective operations (if available) supported by the underlying low-level communication infrastructure. Such operations are TCP sockets send() and recv(), SMP memory copy in and out, LAPI_Put and LAPI_Get, VipPostSend and VipPostRecv, etc. If a collective algorithm can be implemented with only one underlying primitive operation, such as a shared memory barrier, than the count of primitive operations for the corresponding MPI operation MPI_Barrier() will be one. By basing the definition on the number of primitive operations used to implement an MPI collective operations, PERUSE strives to be more generic. For example, an alternative definition based on the count of MPI point-to-point operations might not be appropriate for some MPI implementations as they may provide collective operations that are not layered on top of the MPI point-to-point calls. PERUSE does not associate any interpretation of the performance capabilities of the MPI collective operations based on the number of primitive communication operations. A very efficient algorithm may use a larger number of primitive operations ordered or pipelined in a manner that results in a better overall performance. PERUSE collective metrics only provide information about the number of the primitive transfers – the interpretation is left to the MPI library or performance tool developers.

If the implementation of the collective operations in the MPI library is based on point-to-point MPI operations and PERUSE requests or queue metrics are activated, the MPI library will collect performance data for these metrics for the collective operations as well. The definition of the request and queue metrics does not distinguish on the bases of who the initiator of the operations is – whether the user is calling directly point-to-point operations, or the library implements collective communication or is performing other control MPI-level communication. An alternative definition is possible, according to which of the point-to-point metrics are not affected by collective operations. This alternative is not covered in this version of the specification.

PERUSE_COMM_N_SENDS	Number of individual primitive send operations associated with a collective operation. The measurement (counter update) is made before every primitive send operation executed in the collective operation. *Rationale*: Collective algorithms can be implemented by MPI libraries in many different ways. This metric gives an indication of the level of participation of the particular process in collective operations by counting the number of send primitive operations. On some platforms, this metric can be used for finding a more appropriate allocation of processes to processors so that the number of primitive send operations is minimized.
PERUSE_COMM_N_RECVS	Number of individual primitive receive operations associated with a collective operation. The measurement (counter update) is made before every primitive receive operation executed in the collective operation. *Rationale*: Collective algorithms can be implemented by MPI libraries in many different ways. This metric gives an indication of the level of participation of the particular process in collective operations by counting the number of receive primitive operations. On some platforms, this metric can be

	used for finding a more appropriate allocation of processes to processors so that the number of primitive receive operations is minimized.
PERUSE_COMM_T_BTWN_OPS	Time between primitive send or receive operations associated with a collective operation, if more than one primitive operation is executed by the process. This metric gives an indication about the progress of individual transfers associated with collective operations. The time stamps are taken at the same locations as the PERUSE_COMM_N_SENDS and PERUSE_COMM_N_RECVS metrics. *Rationale*: This statistics can be used to infer information about the progress of send and receive messages, possibly intermediate ones. This information can help detect issues with the implementation of collective operations or with scheduling of the individual primitive transfers.

12.2 Parallel IO metrics (MPI_File)

PERUSE provides a set of parallel file I/O metrics. In regards to these metrics, the term *disk access* or I*/O operation* refers to the operating system *read* and *write* operations, as observed by the MPI library. If a user level file system with OS-bypass is used instead, the I/O operations will be those calls made to the OS-bypass library that initiated file read and write operations. This definition is consistent with the definition of the message transfer initiation specified above. A *readv* operation is considered one primitive operation similar to the primitive collective operations counted in the MPI collective operations metrics. One possible alternative definition would reflect the actual physical writes and reads to/from the storage medium. However, this information is generally available only through the operating system and the disk drivers. Since the MPI library is most frequently implemented as a user-level library, this second definition is impractical.

The I/O *cached bytes* are those bytes that the MPI library caches internally, possibly for performance purposes, and not the bytes that the operating system buffers. *Shipped bytes* represent data that the MPI implementation transfers to other processes that might perform the actual disk I/O operations. Shipped bytes are introduced because PERUSE metrics have local semantics and some MPI I/O optimizations might result in a situation where a process that has received an *MPI_File_write/read()* request may actually perform only communication operations to other processes (on the same or on different machines). Finally, *immediate bytes* are the bytes that the MPI library directly writes/reads using the operating system I/O calls or the user-level library I/O calls.

PERUSE_FILE_N_DISK_ACCESSES_PER_IOREQ	Number of disk accesses associated with an IO request. The non-contiguous I/O access pattern of the parallel application can be specified with a single MPI-IO read/write call by setting appropriate file views. Based on the algorithm used in the implementation, the non-contiguous file access pattern could be accomplished using a single or multiple disk accesses. This metric finds the total number of disk accesses involved in the processing of an I/O request. The measurement is made before each primitive I/O operation related to accessing a file. *Rationale*: This metric gives an indication about the actual implementation of I/O in the MPI library and may help designers of I/O applications create more efficient type maps for reducing the number of disk accesses.
PERUSE_FILE_N_SHIPPED_BYTES	Number of bytes sent/received over the network as opposed to written/read into/from a file for an I/O request. In collective I/O,

	MPI libraries can implement optimizations for reducing the total number of disk accesses by re-organizing the user buffers. This metric shows the number of bytes that are sent/received to/from other processes along the lines of these optimizations. The measurement is taken after the last "shipped" byte associated with the I/O request is sent/received. *Rationale*: Using this information, users can observe the behavior of the MPI library in selecting the optimal disk access decisions. Designers can develop more efficient decisions for distributing the data among processes.
PERUSE_FILE_N_ACCESS_BYTES	Number of bytes actually written/read to/from disk for an I/O request. In collective I/O, MPI libraries can implement optimizations for reducing the total number of disk accesses by re-organizing the user buffers. This metric shows the number of bytes that are actually written/read to/from disk. The measurement is taken after the last "accessed" byte associated with the I/O request is written/read. *Rationale*: Using this information, users can observe the behavior of the MPI library in selecting the optimal disk access decisions. Designers can develop more efficient decisions for distributing the data among processes.
PERUSE_FILE_N_TEMP_FILES	Number of temporary files. Depending on the nature of the I/O request, certain number of temporary files are generated to store the intermediate results. These temporary files may be deleted after the successful completion of the I/O requests. This metric reports the number of temporary files that are created during the course of an MPI-I/O operation. The measurement is taken before the creation of every temporary file. *Rationale*: If a large number of temporary files are used by the MPI I/O implementation, this may lead to unexpected delays and negative impact on performance. This metric can provide information for detecting such situations.
PERUSE_FILE_T_IOREQ_ACCESS	Time spent on accessing the I/O system beginning with the initiation of the I/O request to its completion. This metric measures the total time for an I/O request spent on performing one or more disk accesses. The completion of the I/O request may also involve other operations, such as buffer agglomeration and communication for global buffer reorganization. The first time stamp is taken before the first disk access. The second time stamp is taken after the last disk access is completed. *Rationale*: This metric measures the actual time spent on disk accesses. Using this information the user might be able to estimate the efficiency of the I/O operation and get an understanding about the overhead activities.
PERUSE_FILE_T_GET_SHARED_POINTER	Time for obtaining shared file pointer. File access using shared file pointers is an atomic file operation and only one process can possess the shared file pointer at a single instance of time. Hence, when other processes need to perform shared file access operations, they need to wait until the process holding the shared file pointer relinquishes the file pointer. The first time stamp is taken before the attempt for obtaining the shared file pointer. The second time stamp is taken after the shred file pointer is obtained. *Rationale*: Using this information designers of I/O programs that use shared lock may be able to develop algorithms with better scheduling of I/O activities so obtaining the shared lock does not become a source of unnecessary synchronization overhead.
PERUSE_FILE_T_FIRST_PHASE_IN_SPLIT_IO	Time spent in the first phase of split I/O. Split I/O is an asychronous version of the collective I/O operations. In split I/O, the I/O operation is asynchronously commenced during the MPI_XXX_Begin() call and completed using the

	MPI_XXX_End() call. This metric measures the actual time spent on all I/O related activities, including disk accesses and internal communication in the first phase. The first time stamp is taken before the first I/O or communication activity. The second time stamp is taken after the last I/O or communication activity. *Rationale*: The information provided by this metric can be used for understanding when the actual I/O activities take place – during the first phase, during the second phase, or asynchronously between the two phases.
PERUSE_FILE_T_SECOND_PHASE_IN_SPLIT_IO	Time spent in the second phase of split I/O. This metric measures the actual time spent on I/O and communication activities in the second phase of the split-collective I/O. The first time stamp is taken before the first I/O or communication activity. The second time stamp is taken after the last I/O or communication activity. *Rationale*: The information provided by this metric can be used for understanding of when the actual I/O activities take place – during the first phase, during the second phase, or asynchronously between the two phases.

12.2.1 PERUSE_Event_file_register

Synopsis

int PERUSE_Event_file_register(int event, MPI_File file,
peruse_file_callback_t *callback_fn, void *param, peruse_event_h * event_h)

Input parameters

event – event descriptor
file – valid MPI file handle
callback_fn – user callback
param – user-specific data

Output parameters

event_h – inactive event handle

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT, PERUSE_ERR_FILE, PERUSE_ERR_PARAMETER

Description

Used to create an event handle *event_h* related to an MPI file object by associating the event descriptor *event* and file handle *file*. The user callback *callback_fn* is registered with the output event handle. This callback function will be called when the MPI library performs an action that will affect the event described by *event_h*. If *callback_fn* is NULL, PERUSE_ERR_PARAMETER will be returned. PERUSE_ERR_EVENT will be returned if the input *event* descriptor is invalid, and PERUSE_ERR_FILE if the *file* handle is invalid. PERUSE_ERR_FILE indicates that the MPI library would have returned MPI_ERR_FILE class if the user code tried to reference *file* in an MPI call.

12.2.2 PERUSE_Event_file_callback_set

Synopsis

int PERUSE_Event_file_callback_set(peruse_event_h event_h,
peruse_file_callback_t *callback_fn, void *param)

Input parameters

event_h – event handle
callback_fn – user defined callback function
param – user specific parameter that will be passed to the callback function

Output parameters

n/a

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_PARAMETER, PERUSE_ERR_MPI_OBJECT

Description

This function associates a user defined file *callback_fn* function with an inactive event handle *event_h*. The *event_h* and *param* input parameters will be passed to *callback_fn* when it is invoked. The old callback will be lost and only the callback registered with this call will be kept. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter or *event_h* is active, the function returns PERUSE_ERR_EVENT_HANDLE. If NULL is passed as *callback_fn*, PERUSE_ERR_PARAMETER will be returned. PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

12.2.3 PERUSE_Event_file_callback_get

Synopsis

int PERUSE_Event_file_callback_get(peruse_event_h event_h,
peruse_file_callback_t **callback_fn, void **param)

Input parameters

event_h – event handle

Output parameters

callback_fn – user defined callback function
param – user specific parameter that was passed to the callback function

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENTHANDLE, PERUSE_ERR_MPI_OBJECT

Description

This function obtains the user defined callback function that is associated with the event handle *event_h*. The value of the output parameter *param* is the one passed in by the user when the callback was registered. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter or *event_h* is active, the function returns PERUSE_ERR_EVENT_HANDLE.
PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

12.2.4 File Related Items to be incorporated in peruse.h

```
typedef struct _peruse_file_spec_t
{
        MPI_File        file;
        void            *buf;
        int             count;
        MPI_Datatype    datatype;
        MPI_Offset      offset;
        int             operation;
} peruse_file_spec_t;

typedef int (peruse_file_callback_f)(peruse_event_h event_h,
                MPI_Aint unique_id, peruse_file_spec_t *spec, void *param);

/* Initialize event associated with an MPI file */
int PERUSE_Event_file_register(
                int                     event,
                MPI_File                file,
                peruse_file_callback_f  *callback_fn,
                void                    *param,
                peruse_event_h          *event_h);

/* Set a new file callback */
int PERUSE_Event_file_callback_set(
```

```
                peruse_event_h          event_h,
                peruse_file_callback_f  *callback_fn,
                void                    *param);
```

12.2.5 MPI File code example

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include "peruse.h"

#define FMETRICS        9
#define DATA_SIZE       (64 * 1024)
#define FNAME           "peruse_file"

char *fmetrics[FMETRICS] =
{
        "PERUSE_FILE_N_DISK_ACCESSES_PER_IOREQ",
        "PERUSE_FILE_N_CACHED_BYTES",
        "PERUSE_FILE_N_SHIPPED_BYTES",
        "PERUSE_FILE_N_IMMEDIATE_BYTES",
        "PERUSE_FILE_N_TEMP_FILES",
        "PERUSE_FILE_T_IOREQ_COMPLETION",
        "PERUSE_FILE_T_GET_SHARED_POINTER",
        "PERUSE_FILE_T_FIRST_PHASE_IN_SPLIT_IO",
        "PERUSE_FILE_T_SECOND_PHASE_IN_SPLIT_IO",
};

typedef struct _measure_t
{
        int n_measure;
        int count;
        double stamp;
        double total_time;
        double ave_time;
        double max_time;
} measure_t;

measure_t fdata[FMETRICS];

int file_callback(peruse_metric_h mh, int mstate, long count_val,
                        peruse_file_spec_t *fspec, void *param)
{
        measure_t *ft = (measure_t *)param;
        double t;

        switch(mstate)
        {
        case PERUSE_TIME_BEGIN:
                ft->stamp = MPI_Wtime();
                break;

        case PERUSE_TIME_END:
                t = MPI_Wtime() - ft->stamp;
                ft->total_time += t;
                if(t > ft->max_time)
                        ft->max_time = t;
                ft->ave_time = (ft->n_measure * ft->ave_time + t) /
                                        (ft->n_measure + 1);
                ft->n_measure++;
```

```
                break;

        case PERUSE_COUNTER:
                ft->count += count_val;
                ft->n_measure++;
                break;

        default:
                printf("Unexpected metric type\n");
                return MPI_ERR_INTERN;
        }

        return MPI_SUCCESS;
}

void UserFileIoCode(){}

int main(int argc, char **argv)
{
        peruse_metric_h mh[FMETRICS];
        int rv, size, i, rank, mid;
        MPI_File file;
        MPI_Info info;
        char fname[MPI_MAX_OBJECT_NAME];

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        rv = PERUSE_Init();
        if(rv != PERUSE_SUCCESS)
        {
                printf("Error in PERUSE_Init: rv=%d\n", rv);
                fflush(stdout);
                exit(1);
        }

        MPI_Info_create(&info);
        MPI_Info_set(info, "data_access", "NON_BLOCKING");
        sprintf(fname, "%s.%d\n", FNAME, rank);

        MPI_File_open(MPI_COMM_WORLD, fname,
                MPI_MODE_CREATE | MPI_MODE_RDWR | MPI_MODE_DELETE_ON_CLOSE,
                info, &file);

        for(i = 0; i < FMETRICS; i++)
        {
                memset(&fdata[i], 0, sizeof(measure_t));
                mh[i] = PERUSE_METRIC_HANDLE_NULL;
                PERUSE_Query_metric(fmetrics[i], &mid);
                if(mid == PERUSE_METRIC_INVALID)        /* not supported */
                        continue;
                PERUSE_Metric_file_init(mid, file, file_callback,
                                                &fdata[i], &mh[i]);
                PERUSE_Metric_start(mh[i]);
        }

        /* User code with file I/O operations begin here */
        UserFileIoCode();

        for(i = 0; i < FMETRICS; i++)
        {
                if(mh[i] == PERUSE_METRIC_HANDLE_NULL)
                        continue;
                PERUSE_Metric_stop(mh[i]);
                PERUSE_Metric_free(&mh[i]);
```

```
        }

        MPI_File_close(&file);
        MPI_Finalize();

        return 0;
}
```

12.3 One sided communication (MPI_Win)

PERUSE_WIN_T_BTWN_REQ_AND_XFER	Time between request submission and data transfer start. One-sided communication is non-blocking. The actual delivery of data is not required to happen before the access epoch is closed. This metric provides information about the delay between the user requests for one-sided communication and the moment when the library actually initiates the transfer. The first time stamp is taken when the one-sided request is submitted. The second time stamp is taken when the library initiates the transfer of the first byte of the user buffer. Control packets related to internal packets are not counted as part of the user buffer. *Rationale*: Thie metric provides information about the progress of the non-blocking one-sided communication operations. Although the standard allows the library to delay the communication until the access epoch is closed, user programs may need a better understanding of the behavior and the policies of the MPI library for completing the one-sided requests.
PERUSE_WIN_T_XFER	Time between transfer initiation and completion. This metric measures the actual time for transmitting the one-sided message, possibly accounting for special protocols and overheads that the MPI library may introduce. The first time stamp is taken before the first byte of the user buffer is scheduled for transfer. The second time stamp is taken after the last byte of the user message is scheduled for transfer. *Rationale*: Similarly to the non-blocking send requests, the MPI libraries may have message progress engines are unable to move messages independently and the user process may need to call the library frequently in order to ensure timely progress. This metric can help users understand the behavior of the MPI library and modify their program to use better the capabilities of the MPI library.

12.3.1 PERUSE_Event_win_register

Synopsis

int PERUSE_Event_win_register(int event, MPI_Win win,
peruse_win_callback_t *callback_fn, void *param, peruse_event_h *event_h)

Input parameters

event – event descriptor
win – valid MPI window handle
callback_fn – user callback
param – user-specific data

Output parameters

event_h – inactive event handle

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT, PERUSE_ERR_WIN,
PERUSE_ERR_PARAMETER

Description

Creates a event handle *event_h* related to an MPI window object by associating the event descriptor *event* and window handle *win*. The user callback *callback_fn* is registered with the

output event handle. This callback function will be called when the MPI library performs an action that will affect the event described by *event_h*. If *callback_fn* is NULL, PERUSE_ERR_PARAMETER will be returned. PERUSE_ERR_EVENT will be returned if the input *event* descriptor is invalid, and PERUSE_ERR_WIN if the *win* handle is invalid. PERUSE_ERR_WIN indicates that the MPI library would have returned MPI_ERR_WIN class if the user code tried to reference *win* in an MPI call.

12.3.2 PERUSE_Eevent_win_callback_set

Synopsis

int PERUSE_Event_win_callback_set(peruse_event_h event_h,
peruse_win_callback_t *callback_fn, void *param)

Input parameters

event_h – event handle
callback_fn – user defined callback function
param – user specific parameter that will be passed to the callback function

Output parameters

n/a

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_PARAMETER, PERUSE_ERR_MPI_OBJECT

Description

This function associates a user defined window *callback_fn* function with an inactive event handle *event_h*. The *event_h* and *param* input parameters will be passed to *callback_fn* when it is invoked. The old callback will be lost and only the callback registered with this call will be kept. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter or *mh* is active, the function returns PERUSE_ERR_EVENT _HANDLE. If NULL is passed as *callback_fn*, PERUSE_ERR_PARAMETER will be returned. PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

12.3.3 PERUSE_Event_win_callback_get

Synopsis

int PERUSE_Event_win_callback_get(peruse_event_h mh,
peruse_win_callback_t **callback_fn, void **param)

Input parameters

event_h – event handle

Output parameters

callback_fn – user defined callback function
param – user specific parameter that was passed to the callback function

Return value

PERUSE_SUCCESS, PERUSE_ERR_EVENT_HANDLE, PERUSE_ERR_MPI_OBJECT

Description

This function obtains the user defined callback function that is associated with the event handle *event_h*. The value of the output parameter *param* is the one passed in by the user when the callback was registered. If PERUSE_EVENT_HANDLE_NULL is passed as an input parameter or *event_h* is active, the function returns PERUSE_ERR_EVENT_HANDLE. PERUSE_ERR_MPI_OBJECT is returned if the MPI object with which *event_h* is associated has been freed.

12.3.4 Win related info to be included in peruse.h

```
typedef struct _peruse_win_spec_t
{
        MPI_Win         win;
        void            *o_buf;
```

```
        int             o_count;
        MPI_Datatype    o_datatype;
        void            *t_buf;
        int             t_count;
        MPI_Datatype    t_datatype;
        MPI_Op          acc_op;
        int             peer;
        int             operation;
} peruse_win_spec_t;

typedef int (peruse_win_callback_f)(peruse_event_h event_h,
                MPI_Aint unique_id, peruse_win_spec_t *spec, void *param);

/* Initialize event associated with an MPI window */
int PERUSE_Event_win_register(
                int                     event,
                MPI_Win                 win,
                peruse_win_callback_f   *callback_fn,
                void                    *param,
                peruse_event_h          *event_h);


/* Set a new win callback */
int PERUSE_Event_win_callback_set(
                peruse_event_h          event_h,
                peruse_win_callback_f   *callback_fn,
                void                    *param);
```

12.4 Improved Tracking of MPI Objects

The initial specification of PERUSE does not provide efficient mechanisms for uniquely linking callback events to specific user level MPI API calls. (A somewhat obtuse mechanism is described in section 5.1.2) An efficient mechanism could be incorporated by exposing an additional status field within MPI messages, but such an implementation requires an exposed change to MPI (which violates one of PERUSE's goals). This item is therefore placed in the "Extensions for Future Consideration" appendix with the thought that if the MPI forum reconvenes, this topic could be addressed.

This could also be used to help distinguish between point-to-point messages and collective messages.

The following provides two mechanisms that illustrate how the MPI unique id could be employed by PERUSE (currently there is no way for callback routines to get user-defined names because only MPI_Wtick and MPI Wtime are allowed within a callback - can this be expanded to MPI_get_comm_name?):

1. User-defined names from MPI_(Win,Comm)_set_name routines. Knowing the user-defined names for MPI objects enables performance tools to display this name to facilitate user's understanding of performance data, because they can more easily associate a performance measurement with a particular MPI object.

2. MPI implementation given unique identifiers for MPI objects. If a user-defined name is not given to an MPI object, then a performance tool can display a unique identifier for that object instead. It is not sufficient to use the MPI object handle in the MPI function call arguments to identify and differentiate between MPI objects, because an MPI implementation may use pointers as the handles for MPI Objects. Because of this, the value of the MPI object handle may not be equal across processes. A performance tool, such as Paradyn, may detect new MPI objects by looking at the values of the arguments in the MPI function calls. If it sees two distinct values, it may erroneously determine that there are two new MPI objects. It would be helpful if there were a portable way to get this unique identifier for MPI objects from the MPI implementation.

Here are two ways to provide this functionality:

1. Add query functions to the interface to provide additional information about MPI objects.

Here's an example of what I mean for communicators:

```
struct _peruse_comm_t *    PERUSE_Query_Comm_info(MPI_Comm comm);
where
typedef struct _peruse_comm_t{
      MPI_Comm    comm;
      char *      name;       //user-defined name
      int         unique_id;  //MPI implementation given unique id for comm
} peruse_comm_t;
```

2. Instead of providing just the handle to the MPI object in the peruse_xxx_spec_t type, use a PERUSE defined type that has more information. For example:

```
_peruse_comm_spec_t{
      peruse_comm_t     pcomm;  // where _peruse_comm_t is as it is defined above
      void *            buf;
      int               count;
      .....      // continued as in the PERUSE specification
}
```

12.5 Information wanted for support of dynamic process creation

1. A notification of a spawn start and spawn end. This will be useful if a tool is interested in knowing how much time is spent in spawning operations and to notify the tool that new processes are now part of the application.

2. Information about those new processes, such as PID, image name, and the node it runs on. This information is needed so that the tool can find and possibly attach to the new processes to measure their performance.

This functionality could be provided in the following way:

Add two new events:
PERUSE_SPAWN_START
PERUSE_SPAWN_END

Add two new functions:
PERUSE_Event_spawn_callback_set
PERUSE_Event_spawn_callback_get

Add a new datatype for the callback fuctions:

```
typedef struct _peruse_spawn_spec_t{
      _peruse_comm_t    *p_comm;
      char              ** argv;
      int               maxprocs;
      MPI_Info          info;
      int               root;
      _peruse_comm_t    *p_intercomm;   //or MPI Comm
      int               ** array_of_errcodes;
      _peruse_process_t * list_of_processes;
} peruse_spawn_spec_t;
```

Add a new datatype to provide information about processes:

```
typedef struct _peruse_process_t{
      int   PID;
      char *      image_name;
      char *      node_name;
      int   peruse_process_id; // a unique id for a process within peruse
} peruse_process_t;
```

12.6 Information wanted for remote memory access

1. Notification of events pertaining to transfer of data by and synchronization of RMA operations. This will help a performance tool give more detailed timing information about RMA operations. It may help the user decide on a synchronization method, a particular data transfer routine, or determine the placing of synchronization routines within their code. For example, it might be helpful if a performance tool could show a user that the RMA data transfer was actually complete long before the synchronization routine was called to end the epoch.

Support for this could be provided by addition of new events and new operations.
These four events will give a performance tool information about data transfer routines:

```
PERUSE_WIN_REQ_ACTIVATE       - RMA operation initiated by the user
PERUSE_WIN_REQ_XFER_BEGIN     - transfer of data for RMA operation begins
PERUSE_WIN_REQ_XFER_END       - transfer of data for RMA operation ends
PERUSE_WIN_REQ_COMPLETE       - the operation is complete with respect to
                                the definition for the particular
                                synchronization method being used. For
                                example, if lock/unlock synchronization
                                is used, then this event will occur when
                                the data transfer is complete at both the
                                origin and the target processes, as that
                                is how complete is defined for
                                lock/unlock synchronization in the MPI
                                standard.  Another way to define this
```

```
                                    would be to say that this event would
                                    occur when a synchronization operation
                                    called for the data transfer operation
                                    would not block.
```

The next two events will give a performance tool information about RMA synchronization routines. From these the tool will know that a synchronization operation has occurred and how long it took to execute.

```
      PERUSE_WIN_SYNC_BEGIN   - the synchronization routine in the user
application begins
      PERUSE_WIN_SYNC_END     - the synchronization routine in the user
application ends

      Add new operations so that the tool can differentiate between different
kinds of RMA synchronization:
      PERUSE_WIN_FENCE
      PERUSE_WIN_START
      PERUSE_WIN_COMPLETE
      PERUSE_WIN_POST
      PERUSE_WIN_WAIT
      PERUSE_WIN_LOCK
      PERUSE_WIN_UNLOCK
```

12.7 Information wanted for MPI-I/O

A user-friendly name for MPI File objects to display in the user interface. The standard allows an MPI implementation dependent format for the filename argument given to the MPI_File_open routine. In addition to containing the name of the file, it could also contain information such as a hostname, or a username and password. The standard does not supply an MPI_File_set_name routine. Perhaps implementers of the MPI Standard could supply the actual name of the file through the PERUSE interface. This way, the performance tool would have an MPI implementation independent way to get this information.

12.8 Information wanted for Control Packets

The initial specification of PERUSE does not provide sufficient mechanisms to distinguish control packets (such as the rendezvous control packet) from user data. This information is of use to parallel tool developers.

12.9 Additional Language Bindings

Additional language bindings may be desirable (e.g. python, java, …).