The MPI_T Events Interface: An Early Evaluation and Overview of the Interface

Marc-André Hermanns[a,b], Nathan T. Hjelm[c], Michael Knobloch[b], Kathryn Mohror[d], Martin Schulz[e]

[a] *JARA-HPC, 52425 Jülich, Germany*
[b] *Jülich Supercomputing Centre,Forschungszentrum Jülich GmbH, Germany*
[c] *HPC Division,Los Alamos National Laboratory, USA*
[d] *Center for Applied Sci. Comp.,Lawrence Livermore National Laboratory, USA*
[e] *Department of Informatics, Technical University of Munich, Germany*

Abstract

Understanding the behavior of parallel applications that use the Message Passing Interface (MPI) is critical for optimizing communication performance. Performance tools for MPI currently rely on the PMPI Profiling Interface or the MPI Tool Information Interface, MPI_T, for portably collecting information for performance measurement and analysis. While tools using these interfaces have proven to be extremely valuable for performance tuning, these interfaces only provide synchronous information, i.e., when an MPI or an MPI_T function is called. There is currently no option for collecting information about asynchronous events from within the MPI library. In this work we propose a callback-driven interface for event notification from MPI implementations. Our approach is integrated in the existing MPI_T interface and provides a portable API for tools to discover and register for events of interest. We implement our MPI_T Events interface in Open MPI and demonstrate its functionality and usability with a small logging tool (MEL) as well as an early integration into the comprehensive measurement infrastructure Score-P.

Keywords: MPI; callback functions; runtime introspection; performance measurement; tool interfaces

1. Introduction

A detailed understanding of parallel application behavior in High-Performance Computing (HPC) is key to performance optimization. The vast majority of HPC parallel applications use the Message Passing Interface (MPI) [1] for distributed memory programming. Although the use of MPI is ubiquitous, its optimal use is often not straightforward and the cause of potential performance bottlenecks in many applications.

For decades, MPI developers have used performance measurement and analysis tools to gain insight into application behavior. Performance tools collect a wide variety of information about applications during execution, including time spent in different activities, e.g., function calls, application phases, or particular lines of code and communication and synchronization details, e.g., communication patterns between processes and the overhead of particular synchronization operations. Using this collected information, developers can identify the performance bottlenecks in their code and target their optimization efforts at the most severe performance problems in order to achieve the highest performance gains.

The current MPI Standard offers two interfaces for tools to extract information from an MPI application, namely PMPI, the MPI Profiling Interface, and MPI_T, the MPI Tool Information Interface. The concept of PMPI is simple and has been used successfully for decades; tool developers write libraries that intercept selected (or all) MPI calls in an application execution and perform a wide variety of tasks, including measuring the time spent in MPI calls, collecting communication details (such as bytes transferred or communication partners), or replacing a communication pattern altogether, e.g., replacing a broadcast operation with point-to-point calls.

While PMPI has proven to be powerful, information about the internal workings of the MPI library were not available to tools with PMPI. Thus, in MPI 3.0 the MPI_T interface was added to the standard to enable an MPI implementation to expose selected details about its configuration and execution. With this interface, tools or applications can query and possibly set MPI-internal and MPI implementation specific configuration variables. Examples of such variables could be the eager limit for messages or the type of collective algorithm used for particular operations. Additionally, tools can obtain performance data recorded within the MPI library. Examples for the latter could be the amount of memory used in MPI-internal buffers or the length of message queues. In both cases, the MPI implementation can choose what to expose and in which form. This was a key element in the design of the interface as to not restrict the implementation options of individual libraries. While this leads to a certain degree of vagueness, as tools cannot rely on the existence of particular variables or measurements, MPI_T has proven to be quite popular and several approaches have shown the benefit of using MPI implementation internal information for tuning MPI applications [2, 3, 4, 5, 6, 7].

While the introduction of MPI_T improved the amount performance information available to tools, one area is still miss-

Email addresses: m.a.hermanns@fz-juelich.de (Marc-André Hermanns), hjelmn@lanl.gov (Nathan T. Hjelm), m.knobloch@fz-juelich.de (Michael Knobloch), kathryn@llnl.gov (Kathryn Mohror), schulzm@in.tum.de (Martin Schulz)

ing: information about event occurrences within MPI implementations. To mitigate this gap, we propose to extend the MPI_T interface with a new callback-driven mechanism to notify tools of events that occur during execution. Using the event interface, a tool can register for and be notified of events that the MPI implementation exposes. These events could include the progress of communication activities, the time the actual transfer of a non-blocking send operation starts, or the time of the message arrival on the receiver side. Such events could be useful in diagnosing performance issues of applications, e.g., by comparing the non-blocking message arrival time with the time of the post of the matching wait operation. As with MPI_T, the number and types of events exposed is MPI implementation dependent, allowing for flexibility in the implementation and avoids forcing particular implementation styles in the standard.

Our MPI_T Events interface, a callback-driven extension to MPI_T, is currently being discussed for inclusion in the MPI Standard. As this extension is currently not a part of the MPI Standard, we anticipate that (if accepted) the API for MPI_T Events finally integrated into the standard may differ slightly from what is presented here. However, our intent is to describe the state of the interface as of this writing and to explore the design choices we have faced in our efforts, both to demonstrate the feasibility of the proposed API as well as to give the larger MPI community an option to provide feedback.

This paper is an extended version of our earlier publication [8] on this topic. We retain the contributions of the original paper:

- A detailed description of the design of the MPI_T Events interface, including a novel transparent buffering approach and a discussion of design trade-offs;
- A prototype implementation in Open MPI;
- An evaluation of the interface with an event logger tool; and
- An integration of the interface into the comprehensive Score-P [9] measurement infrastructure.

Additionally, the contributions of this extended version include:

- An extended discussion of overhead measurements with different tool configurations across multiple benchmark applications;
- An updated list of event types supported by the presented Open MPI prototype.

The remainder of this paper is structured as follows. Section 2 provides background and related work. Section 3 describes the API and discusses design decisions. Section 4 details the prototype implementation in Open MPI. Section 5 discusses multiple case studies focusing on overhead and usability, and Section 6 concludes with a brief look at future work.

2. Background and Related Work

2.1. Event Information in MPI

In the current MPI Standard, tool developers have two portable mechanisms to obtain information on the communication behavior of an MPI program: either using wrappers through the PMPI interface or by explicitly querying performance information through MPI_T. However, the drawback of both approaches is that data collection is limited to synchronous information, i.e., when the user application calls an MPI routine that is then intercepted by PMPI wrappers or when a tool library explicitly calls into MPI_T.

Performance tool developers have long stated the need for capturing asynchronous MPI event information [10, 11]. By gaining insight to the relative timings of events as they occur in an MPI library, one can understand the order of events, track causality, and with that uncover additional performance problems not visible in synchronous data. For example, if a message is received by the MPI library and the user code has not yet posted a matching receive call, the message will be placed in the unexpected message queue. By knowing the relative time between the arrival of the message and the posting of the receive call, one can infer potential causes. For example, this could be an indicator of load imbalance that is causing the receiving process to post the receive call late.

2.2. Event Interfaces for Tools

The de facto approach for propagating event information to tools is with callback-driven interfaces: the system being monitored (in our case the MPI library) notifies tools of events near the time of their occurrence in order to indicate state changes in the system. Such interfaces are available or are being designed for a variety of systems. For Unified Parallel C (UPC) [12] and the underlying GASNet [13] library, the callback-driven interface GASP [14] provides information for tools such as the Parallel Performance Wizard (PPW) [15, 16]. The upcoming tools interface [17] for OpenMP [18] will also provide a callback-driven tool interface.

2.3. PERUSE

The first effort to introduce event notification into MPI was called PERUSE [19, 20]. PERUSE was an international effort to design a callback interface to collect internal information from MPI implementations. PERUSE was implemented in Open MPI [21] and used by selected projects in the MPI community [22, 23, 24].

The design of PERUSE differs from our events interface as it defines several specific events as part of its interface, like "message activation" and "message transfer initiation" that refer to the times when MPI starts processing a message request and when it actually begins the data transfer, respectively. Overall, PERUSE defined a large number of events related to message transfers and message queues, and the draft document contains event definitions for collective communications, MPI I/O, one-sided communication and dynamic process creation. The PERUSE specification states that PERUSE-compliant MPI implementations are required to support all PERUSE functions

and data types. However, it also states that if a particular defined event does not directly correlate to a particular MPI implementation or would incur undue overhead to support, implementors are free to ignore that event.

Ultimately, the PERUSE specification was not standardized. The main criticism of the interface was that supporting the defined MPI internal events could lead to performance bottlenecks or restrictions for some MPI implementations if their design doesn't follow the PERUSE concepts. Although the interface specified that implementations could ignore problematic events, there was concern that in order to be considered competitive in procurement bids, MPI implementors would need to support the full PERUSE interface.

Our design of the MPI_T events interface is in direct response to this criticism; as with the existing MPI_T interface, no events are pre-defined or enforced. Instead, we provide a query interface for tools to discover the events that an MPI implementation supports.

3. Design

The design of our callback-driven events interface integrates cleanly with the MPI_T interface. Following the approach of MPI_T for performance and configuration variables, we do not specify any events that must be supported by MPI implementations, but instead leave the choice of which event to provide to the MPI implementor. The MPI_T Events interface then provides functions for tools to discover available events and to register callbacks for them. This allows MPI implementors complete freedom to choose the events to be exposed and how to implement them in their library. As with MPI_T, the proposed events API can be used whenever MPI_T is active, which can be before `MPI_Init` and after `MPI_Finalize`.

In Table 1 we show the proposed API supporting our design. The functions of the API fall into five categories: 1. Querying the availability of events and their descriptions 2. Callback registration management 3. Reading event-instance data within a callback function 4. Reading event-instance metadata within a callback function 5. Querying information on event sources. MPI implementations can, but are not required to propagate event occurrences to callbacks immediately, but can defer and buffer events internally. Our goal with this design choice is to reduce the potential for prohibitively high overhead within MPI implementations caused by having to supporting immediate notification of all events. Furthermore, the API introduces the notion of information *sources* for specific event instances. A source might be internal to the MPI implementation, e.g., internal message queues, or external, e.g., a network device. By introducing the concept of sources, we enable transparent buffering of events from sources with disparate control flows, without the need to enforce event ordering across those flows (see Section 3.6).

3.1. Query Event Type Information

The API for querying event type information follows the same approach as the API for querying MPI_T variables. A subtle difference is that for events we query for available *event types* in the information gathering phase. Then, during execution we collect information about, i.e., register callbacks for specific events or *event instances* that belong to the queried event types. Users query the current number, N, of available event types via `MPI_T_event_get_num`. By calling `MPI_T_event_get_info`, tools can then query detailed information about specific event types provided by the specific MPI implementation identified by its index between 0 and $N-1$. The event type information returned to the user comprises:

Name A string that uniquely identifies the event type among all other event types available

Description An optional string documenting the event type

Verbosity Level The verbosity level of the event type

Event Type Structure A set of arguments describing the structure of the event data, including element data types and displacements, as well as the number of elements

Enumeration Type An optional MPI_T enumeration describing all elements of the event type

Bound Object The type of MPI object (if any) to which the event type must be bound

MPI implementations can add new event types as they become available during execution, e.g., through dynamic loading of components; however, they cannot change event type information, change event type indices or delete indices once they have been added to the set.

The name, description and verbosity have the same semantics as with MPI_T variables. Specifically, the *name* uniquely identifies a given event type among all available event types and must identify equivalent event types across all connected MPI processes. The *description* clear-text string is optional, but can be used to convey semantic information for event types to users. Thus, a high-quality implementation should provide descriptions for all event types to aid users in understanding the information provided. The *verbosity* allows users to judge whether a specific event type represents high- or low-level information, e.g., whether the event type is intended to be helpful for application users or whether it is intended for specialized uses for MPI implementors.

Event types can be complex and can potentially return multiple elements of different types during the callback as parameters. Therefore, returning a single basic datatype, as with MPI_T performance variables, will not suffice, and we represent them conceptually with an *event type structure*. The event type structure comprises 1. the number of elements contained in the event type, 2. an array of MPI basic datatypes allowed for MPI_T describing the type of each element, 3. an array of displacements to identify the location of each element in the event buffer as provided by `MPI_T_event_copy`, and 4. the extent (including potential padding among elements) of such a buffer. We use this approach instead of using MPI derived datatypes directly, as MPI_T may be initialized and used before MPI is initialized

Table 1: List of functions of the proposed MPI_T Events API. The return type is always *int*, returning an MPI error code with the same semantics and scope as existing MPI_T functions.

Name	Arguments
EVENT TYPE INFORMATION	
MPI_T_event_get_num	*int** num_events
MPI_T_event_get_info	*int* event_index, *char** name, *int** name_len, *int** verbosity, *MPI_Datatype** arrary_of_datatypes, *MPI_Aint** array_of_displacements, *int** num_elements, *MPI_T_enum** enum, *MPI_Aint** extent, *char** description, *int** description_len, *int** bind
MPI_T_event_get_index	*const char** name, *int** event_index
CALLBACK REGISTRATION MANAGEMENT	
MPI_T_event_handle_alloc	*int* event_index, *void** object_handle, *void** user_data, *MPI_T_event_cb_function* event_cb_function, *MPI_T_event_registration** event_registration
MPI_T_event_handle_free	*MPI_T_event_registration* event_registration, *MPI_T_event_free_cb_function* free_cb_function
MPI_T_event_set_dropped_handler	*MPI_T_event_registration* event_registration, *MPI_T_event_dropped_cb_function* dropped_cb_function
READING EVENT DATA	
MPI_T_event_read	*MPI_T_event_instance* event, *int* element_index, *void** buffer, *int* size
MPI_T_event_copy	*MPI_T_event_instance* event, *void** buffer, *int* size
READING EVENT METADATA	
MPI_T_event_get_timestamp	*MPI_T_event_instance* event, *MPI_Count** event_timestamp
MPI_T_event_get_source	*MPI_T_event_instance* event, *int** source_index
SOURCE HANDLING	
MPI_T_source_get_num	*int** num_sources
MPI_T_source_get_info	*int* source_index, *char** description, *int** description_len, *MPI_T_source_order** ordering, *MPI_Count** ticks_per_second
MPI_T_source_get_timestamp	*int* source_index, *MPI_Count** timestamp

and, thus, the full MPI type system may not be available during tool initialization.

An optional *enumeration type* provides additional information about the individual elements of the event type. The intent is to allow performance tools to harvest specific element descriptions in machine accessible form, rather than parsing the natural language of the description text for element descriptions. For example, an event type that occurs when an incoming two-sided message is matched may return the tag and size of the incoming message. In this case the enumerator for this event type could return the strings "tag" and "size".

Some event types may be required to be bound to a specific MPI handle as a *bound object* at event callback registration. Binding event callbacks to specific MPI objects allows for more refined event collection. For example, a tool could collect message queue events for a particular MPI communicator instead of all communicators.

As stated previously, event types are identified by their index in the set of all currently available event types, from 0 to $N - 1$. However, as is true for variables in the MPI_T interface, event indices may change between executions, and thus an index is not a reliable identifier for events. However, if the unique name of the event type is known to the user, a call to `MPI_T_event_get_index` will provide the index of the associated event type for that execution. This avoids an iterative search of the full set of event types for a specific, known event type. If no event type is available with the given name, the call returns with an appropriate error code. Also, because event types may become available at different stages of execution, a tool may retry failed attempts to query the event type index in case it becomes available.

3.2. Event Handle Management

In order to receive notifications of individual event occurrences of a particular event type—called event instances—a tool must register a callback function using `MPI_T_event_handle_alloc`[1]. The user provides the following arguments to the registration call:

Index The index of the event type with which the callback function is associated.

Bound Object Handle If needed, a valid MPI object handle to bind to the event instances.

User Data User data that will be provided to the registered callback function. This is intended to pass a pointer to user-controlled memory, but a tool is free to choose what is actually passed.

Callback Function The callback function to call to process event information with the given event type and MPI object.

[1] Note that the name of the call is chosen in symmetry to the existing functions `MPI_T_cvar_handle_alloc` and `MPI_T_pvar_handle_alloc` within the MPI Tool Information Interface.

Table 2: List of all callback function types of the proposed MPI_T events API. All types have a return type of *void*.

Name	Arguments
MPI_T_event_cb_function	*MPI_T_event_instance* event, *MPI_T_event_registration* event_registration, *MPI_T_event_cb_safety* cb_safety, *void** user_data
MPI_T_event_free_cb_function	*MPI_T_event_registration* event_registration, *MPI_T_event_cb_safety* cb_safety, *void** user_data
MPI_T_event_dropped_cb_function	*int* count, *MPI_T_event_registration* event_registration, *MPI_T_event_cb_safety* cb_safety, *void** user_data

Event Registration A handle for identifying this event type registration.

After successful event-callback registration, an *event-registration handle* is returned. The handle is used subsequently for several purposes: 1. As input to each callback invocation 2. For registering a callback for handling information loss due to dropped events, 3. For de-registering the callback. The handle is used as input to each callback because one callback function can be registered for multiple event types, and the handle differentiates the event type for the particular invocation of the callback. Note that multiple handles may exist for a given event type, but each handle is associated with only one specific event type.

If multiple event registration handles exist for the same event type and bound object, the corresponding event instance data is provided to the callback function invocation of each of those handles. This enables multiple tool libraries to register callbacks for the available event types without further coordination.

By calling `MPI_T_event_handle_free`, a user initiates the deallocation of an event registration handle and the de-registration of the associated callback function. Because the API allows event data to be transparently buffered and event callback invocations to be postponed, the MPI implementation may not be able to guarantee that no event data corresponding to the event registration handle is still buffered in the system at the time of the call to `MPI_T_event_handle_free`. Thus, the user can provide a pointer to a callback function of type `MPI_T_event_free_cb_function` (see Table 2 and Section 3.3) that can free any resources allocated by the tool associated with the handle. The callback function is invoked when the MPI implementation can guarantee that no event data for the corresponding handle is pending. After the return of the callback function, the event registration handle is deallocated.

3.3. Event Callback Requirements

In Table 2, we show the C function prototypes for the callbacks in our event interface. These include the

1. `MPI_T_event_cb_function` to be used for event instance notification;
2. `MPI_T_event_free_cb_function` to indicate completed handle deallocation after raising events potentially buffered before the corresponding call to `MPI_T_event_handle_free`; and
3. `MPI_T_event_dropped_cb_function` to handle events that may have been dropped by the MPI implementation.

The MPI implementation may invoke a callback function as soon as it is registered. The API is designed to support different execution contexts for the callback function. To enable the safe and flexible handling of execution contexts both with respect to the tool and the MPI implementation, the requirements for safe execution of a specific callback invocation are communicated to the callback function via the argument `cb_safety`.

The callback-safety requirements are defined in a hierarchy, where each level includes all restrictions of its predecessor in the hierarchy as listed below:

`MPI_T_CB_REQUIRE_NONE` The callback function does not need to fulfill specific requirements.

`MPI_T_CB_REQUIRE_MPI_RESTRICTED` The use of MPI within the callback function is restricted to a specific set of functions.

`MPI_T_CB_REQUIRE_THREAD_SAFE` The callback must expect to be interrupted by and/or run concurrently with itself and other callback functions.

`MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE` The callback must expect to be run in an asynchronous signal handler context.

The callback safety level `MPI_T_CB_REQUIRE_NONE` is the lowest level, with no restrictions on the callback function. We provide this level as a defined minimum. While MPI implementations may never actually provide this level to a callback in an HPC production environment, we do not want to require an MPI implementation to enforce specific restrictions if they are not needed.

The callback safety level `MPI_T_CB_REQUIRE_MPI_RESTRICTED` restricts the use of MPI calls within a callback to 1. reading event data, meta data, and event type information 2. reading source information 3. managing event callback registrations 4. starting, stopping and reading performance variables

The level of `MPI_T_CB_REQUIRE_THREAD_SAFE` requires the callback function to be reentrant and thread safe. This means a developer needs to expect the execution of a callback to be interrupted by any other callback function or happen concurrently with any other callback function.

The most restrictive level, `MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE`, requires the callback to be safe inside a signal handler.

The distinction in callback safety levels allows flexibility for the MPI implementation to make decisions about the needed safety for a specific callback invocation. It provides for interrupt-based calling contexts which require the highest safety level,

as well as calling contexts via a function pointer from a controlled place in the code, which have weaker safety requirements. The weaker requirements may allow the tool to process the event data inside the callback, without requiring any tool-internal buffering. Additionally, allowing a callback to perform further processing than just copying data to a buffer may enable completely self-contained tools that are not dependent on an extra thread or using the PMPI interface to process event information.

Because event information may be buffered by the MPI implementation and not returned immediately upon event occurrence, the internal buffer space may be depleted at some point during the execution. This could occur if event data is generated faster than it is processed by calling the associated callback functions. As a consequence, the MPI implementation will then have to drop some event data. For some tools, the loss of event data may be problematic, depending on the semantic connection of recorded and lost events. Because of this we provide the `MPI_T_event_dropped_cb_function` callback to be called as soon as the MPI implementation can inform the tool of the observed loss of data. The *count* argument tells the tool the count of event instances that were lost. The counter itself only requires constant space for each event type so it should not be a burden on MPI implementations. Depending on the importance of the lost events, the tool may abort its execution, warn the user, interpolate the missing data or simply ignore the lost events. The *event_registration* argument provides the event registration handle for the lost events. As with all other callback functions, the required safety level and the associated pointer to user data is provided.

3.4. Reading Event Data

Our MPI_T Events API provides two methods for extracting information from within an event callback function once it is invoked, namely 1. reading event data one element at a time and 2. copying the event data as one opaque memory chunk for later processing.

Reading single elements. A tool can read single data elements from the event data, represented by the opaque type `MPI_T_event_instance`, with a call to `MPI_T_event_read`. This enables users to copy elements of the event instance to specific memory locations directly. Furthermore, the user does not need to know the displacement for the individual event elements, but can rely on the MPI implementation to copy the element data from the correct memory location. This enables MPI implementations to hide implementation details from the callback (i.e., data layout at callback invocation) and allows tools to copy one or more event data elements directly to tool allocated variables, without the need to copy the event data as a whole.

Copying the event data. In some calling contexts, a callback may not be able to process individual data elements, e.g., due to required asynchronous signal safety. In this case, a tool may choose to copy the event data as a whole (including potential padding) into a user-provided buffer with a call to `MPI_T_event_copy`. The user must provide a buffer with enough capacity to copy as many bytes as returned in the *extent* argument of the call `MPI_T_event_get_info` for the corresponding event type. This enables tools to postpone the processing of event data to a time off of the critical path of the application and possibly to a more permissive execution context. While the event type structure is communicated in the `MPI_T_event_get_info` call, access to the event data is only possible through the event instance handle provided to the callback function. This enables MPI implementations to assemble the event data buffer copied on the fly in the structure communicated through *array_of_datatypes* and *array_of_displacements* of the `MPI_T_event_get_info` call. Of course, on-the-fly assembly contradicts the premise of a fast copy, so implementations are encouraged to implement the copy as efficiently as possible.

3.5. Reading Event Metadata

Instances stemming from all event types share some basic metadata information, including a time stamp and the source of the event (see Section 3.6). Additionally, event instances may include other metadata specific to their event type. In our design, we do not include this part of the metadata in the specification. The primary reason is flexibility; as new event types are supplied by MPI implementations, we do not need to update our API or type definitions. Further, it enables MPI implementations to store the metadata separately from the other event information (or generate it on the fly), and the metadata may not be interesting for all tools, so it is left to the tool to query information when necessary.

Observed Timestamp. A call to `MPI_T_event_get_timestamp` returns the time stamp the event was observed by the MPI implementation, which may be significantly different to when the corresponding callback is invoked. This enables MPI implementations to 1. Postpone the invocation of a callback to a more convenient or less restricted execution time 2. Provide multiple event sources, including hardware components, to provide event data without their explicit support to raise a signal or invoke a function callback.

Users are not required to use `MPI_T_event_get_timestamp` to obtain a time stamp and can use other timer routines; however user-generated time stamps will always reflect the time of the callback invocation rather than the time when the event was initially observed.

Decoupling internal event data generation and notification to the user also allows for internal recording of high-frequency events to burst buffers through the MPI implementation before calling the individual callback functions. Control of such buffers could be granted to the user through MPI_T control and performance variables.

Event Source. Sources provide additional optional information on the origination of the event data and callback invocation. The source concept is introduced into the API to allow for flexible handling of the chronological ordering requirements on event data by a tool, as explained in the next section in more detail. As the event-instance data available to a callback function of a specific registration handle may stem from different

sources for distinct invocations, a callback function can query the source index for a specific event-instance via `MPI_T_event_get_source`.

3.6. Event Sources

Allowing transparent buffering of events in our design may enable MPI implementations to support novel sources for generating events, e.g., directly from the network hardware. However, these sources may not be capable of maintaining the necessary synchronization with other sources for a centralized, coordinated event data buffer. This presents a challenge for some tools and data formats, such as Score-P [9] and OTF2 [25], which have strict requirements on event ordering for the events they record in one stream. Sorting and ordering events from disparate sources during execution would be challenging and error-prone for both tools and MPI implementations. The concept of sources reconciles these challenges with low overhead. Here, a source is a tag attached to the event data that identifies ordered event sub-streams from the unordered combination of event callback invocations from multiple unsynchronized data sources. This means, instead of attempting to coordinate multiple software and hardware components to provide a single chronologically ordered stream of events, an MPI implementation can supply multiple sub-streams identified with source tags, for which ordering can be guaranteed to a tool. Generally speaking an MPI implementation should create a separate source for each control flow that generates event data, e.g., the main thread, a progress thread or a network card.

Users can query the number of sources via `MPI_T_source_get_num` and then query detailed information on a specific source via `MPI_T_source_get_info`, similar to querying events themselves. The detailed information query returns 1. The description of the source 2. The kind of ordering guarantees of the source 3. The number of ticks the timestamp of this source advances per second The kind of ordering guarantee can either be `MPI_T_SOURCE_ORDERED` for guaranteed chronological order or `MPI_T_SOURCE_UNORDERED` otherwise. This allows MPI implementations to provide event information even from sources where ordering cannot be guaranteed or only with substantial overhead and inform the tool accordingly. The tool can then choose how to handle the unordered source. Nevertheless, an MPI implementation should strive to keep the number of unordered sources low.

The time stamp returned by `MPI_T_event_get_timestamp` is a count of ticks since some time in the past and a user can query the current time stamp of a source via `MPI_T_source_get_timestamp`. However, time stamps of different sources may not be directly comparable without a manual translation into a common time format. In combination with a common reference time of the same timestamp and the number of ticks per seconds reported for the source, any later time stamp of that source can be translated to such a common time format easily.

4. Implementation

To evaluate the proposed MPI_T Events interface, we implement it in Open MPI. The authors' familiarity with the implementation, the existing PERUSE implementation and the modular design of Open MPI make it well suited for implementing this prototype, but the results generalize to other MPI implementations as well, especially those with a robust MPI_T support.

4.1. Open MPI

Open MPI is designed around the concept of a Modular Component Architecture, know as the MCA. At a high level, the implementation is split into three layers; the Open Platform Abstraction Layer (OPAL), the Open Run-Time Layer (ORTE) and the Open MPI Layer (OMPI). OPAL implements the core of the MCA. Each layer encompasses multiple interfaces known as frameworks, which are then each implemented in the form of one or more components.

The OPAL layer includes the code responsible for implementing both performance and configuration variables exposed by the existing MPI_T interface, as this allows variables to be exposed from any of the layers in the Open MPI implementation. The core of the prototype event-driven extension to MPI_T is therefore also implemented in the same layer.

The new event support in Open MPI consists of both internal and external facing APIs. The internal calls handle the registration, de-registration, and invocation of event instances. The external-facing calls handle all the functions necessary to implementing the new MPI_T Events API calls as specified earlier in Table 1.

We expect that, as more internal event information is exposed via the internal event registration mechanism, there will be additional overhead, possibly even on the critical path. In our implementation we therefore mitigate as much of this overhead as possible. This includes the use of low-overhead, single conditional, inline functions for the invocation of event instances and a handle allocation callback function that can be specified at event registration time. The handle allocation callback is called when the tool calls `MPI_T_event_handle_alloc`. This allows the component implementing a particular event to defer some of the overhead associated with using the event to a point when a tool is actually attaching to the event. The goal of this design is to allow all events to be compiled into the implementation with minimal overhead and hence be always present without switching library versions. The alternative would be to conditionally compile support for these event types, which would reduce the usefulness of the implementation.

4.2. Events

For the initial implementation of the MPI_T Events prototype we focus on implementing event types to cover two-sided (send/recv) and one-sided communication. The two-sided event types are implemented in the *ob1* Point-to-point Management Layer (PML) component and cover the complete set of events that were implemented to support PERUSE. One-sided events are implemented to cover network operations in the *ugni* Byte Transport Layer (BTL) [26] component. These event types indicate the initiation or completion of a one-sided (i.e., *put*, *get* or *Atomic Memory Operation (AMO)*) operation. The one-sided

events are added to assist in the evaluation of Open MPI on Cray systems when using the RMA-MT benchmark suite [27]. Table 3 provides a complete list of the events exposed in the prototype implementation.

5. Case Studies

To demonstrate the use of the MPI_T Events, we provide several examples of smaller case studies in this section. Their purpose is to show how individual parts of the API can be used to obtain generic or specific performance-relevant information.

5.1. Overhead Study

Figure 1 shows results from an overhead study performed with selected benchmarks of the SPEC MPI 2007 version 2 benchmark suite. The measurements were performed on the Intel cluster JURECA [28] at Forschungszentrum Jülich with four nodes and 24 processes per node resulting in 96 processes in total. For each configuration the `mref` input size was chosen. The original benchmarks were patched to measure the time spent in the main loop excluding initialization using `MPI_Wtime`. The figure shows the average execution time of 5 measurements per configuration, with error bars indicating the minimum and maximum across the measurements. The *Events disabled* configuration has the MPI_T callback system disabled at compile time in the Open MPI runtime system. The *Events enabled w/o tool* configuration has the MPI_T callback system enabled, but no tool attached. The *MEL w/o callbacks* configuration has the MEL tool attached, with no callbacks registered. In this configuration the MPI initialization and finalization is intercepted via PMPI wrappers and the MPI_T subsystem is initialized and finalized in those wrappers, respectively. The *Events enabled w/ empty callbacks* also attaches a the MEL tool, now registering an empty callback function for each event available.

As can be seen in the figure, the overhead of the MPI_T callback system is within or close to the measurement uncertainties of an Open MPI version with a callback system fully disabled at compile time, for three of the four benchmarks. For the 121.pop2 benchmarks, the overhead with an enabled callback system is consistently larger than without, but stays below 3 percent (2.8%) when comparing the lowest runtime of *Events disabled* with the largest runtime of *MEL w/ empty callbacks*. 121.pop2 is a very communication intensive benchmark, and consequently even a small overhead for communication functions is immediately noticeable. With Score-P attached, running in runtime-summarization mode, the overhead is again significantly higher than observed with MEL, even with no additional callback functions registered. When Score-P also registers the callback functions that track searches in the posted and unexpected message queue, the overhead increases yet again, as the time spend searching the in the queues is so low per instance that the overhead of recording their beginning and end is relatively high.

The fluctuations in the runtime of 132.zeusmp2 still needs to be investigated further. While four out of the five runs are very similar to the comparable runs with the other benchmarks, we observed one isolated run with a significantly reduced runtime, which influenced the average execution time significantly. The fact that 132.zeusmp2 shows significant runtime variation across the measurements for each configuration could indicate that this benchmark is much more sensitive to noise than the other tested applications, but it could also just point to a severe anomaly in the machine during the one outlier run.

Figure 2 shows the results of the overhead from the RMA-MT benchmark suite. These results were performed on the same JURECA cluster as the SPEC benchmarks. Open MPI was configured to use the osc/rdma[29] and btl/uct components for communication. These components have been heavily optimized for RMA-MT communication. The RMA-MT benchmark suite was configured to run with 16 threads, binding worker threads to cores, and run for 1000 iterations. The figures show the results of the `rmamt_bw` bandwidth benchmark with the flush, flush_all, and all_flush (all threads calling `MPI_Win_flush`) synchronization methods with both `MPI_Put` and `MPI_Get`. With the MEL tool attached with no callbacks there is little degradation in performance across the range of message sizes when using `MPI_Put`. The overhead is higher when running with MEL and empty callbacks. In this case it leads to a degradation in performance around 10% for small messages ($< 1kB$). The overhead on the small message bandwidth when using `MPI_Get` and the large message bandwidth for both `MPI_Get` and `MPI_Put` is within the uncertainty of the benchmark.

5.2. MEL—MPI_T Events Logger

We developed the *MPI_T Events Logger* library (MEL) as a prototypical example of a generic events logger and extended it with basic message queue profiling capabilities. MEL is a profiling library that employs both the PMPI and MPI_T interfaces to obtain performance relevant information. As described in Section 3, event types can either be unbound (i.e., not tied to a specific object) or bound (i.e., tied to a specific object handle such as a specific communicator). Handles for unbound event types can be allocated once during the initialization of the measurement system. Handles for bound event types need to be allocated anew for each newly created object handle. For that purpose, MEL intercepts all MPI calls that create new handles. As the Open MPI prototype currently only supports event types bound to communicators, the MEL prototype used for this paper only intercepts communicator handle creation routines. At startup, MEL allocates event handles for event types bound to communicators for the implicitly defined communicator handles `MPI_COMM_WORLD` and `MPI_COMM_SELF`. In the default behavior, during execution MEL evaluates the environment variable `MEL_EVENTS`, which may contain a list of event type names separated by comma, colon, semicolon, or spaces. If the variable is unset or empty, MEL will query all event types available at the end of the execution and dump the gathered information. If the variable is set, MEL allocates handles for all events types listed.

5.2.1. Generic Events Logging

Using a generic event callback for all event types, MEL uses the information available on the structure of the event type to

Table 3: List of events exposed by the pml/ob1 component in the prototype MPI_T events implemenation in Open MPI

Event Name	Binding	Description	Event Data
message_arrived	Comm.	Message arrived for match	Communicator ID, Source rank, Tag, Sequence number
search_posted_begin	Comm.	Starting search of the posted receive queue	Source rank, Tag
search_posted_end	Comm.	Finished search of the posted receive queue	Source rank, Tag
search_unexpected_begin	Comm.	Starting search of the unexpected message queue	Request pointer
search_unexpected_end	Comm.	Finished search of the unexpected message queue	Request pointer
posted_insert	Comm.	Added request object to the posted receive queue	Request pointer
posted_remove	Comm.	Removed request object to the posted receive queue	Request pointer
unex_insert	Comm.	Added request object to the unexpected message queue	Request pointer
unex_remove	Comm.	Removed request object to the unexpected message queue	Request pointer
transfer_begin	Comm.	Data transfer has begun for a request	Request pointer
transfer	Comm.	Data transfer on request	Request pointer
cancel	Comm.	Receive request was canceled	Request pointer
free	Comm.	MPI request was freed	Request pointer

Table 4: List of events exposed by the osc/rdma component in the prototype MPI_T events implemenation in Open MPI

Event Name	Binding	Description	Event Data
lock_acquired	Win.	A lock on a remote peer was acquired	Target rank (-1 for lock all)
lock_released	Win.	A lock on a remote peer was released	Target rank (-1 for unlock all)
put_started	Win.	A network put operation on a contiguous region has been started	Target rank, Remote address, Size
put_complete	Win.	A network put operation on a contiguous region is complete (may not be available on all platforms)	Target rank, Remote address, Size
get_started	Win.	A network get operation on a contiguous region has been started	Target rank, Remote address, Size
get_complete	Win.	A network get operation on a contiguous region is complete (may not be available on all platforms)	Target rank, Remote address, Size
flush_started	Win.	A flush synchronization operation has started	Target rank (-1 for lock all)
flush_complete	Win.	A flush synchronization operation has completed	Target rank (-1 for lock all)
pscw_expose_start	Win.	A Post-Start-Complete-Wait (PSCW) exposure epoch has started	None
pscw_expose_complete	Win.	A PSCW exposure epoch has completed	None
pscw_access_start	Win.	A PSCW access epoch has started	Target rank
pscw_access_complete	Win.	A PSCW access epoch has completed	Target rank
rdma_fence	Win.	A fence epoch has started	None

Table 5: List of events exposed by the btl/ugni component in the prototype MPI_T events implemenation in Open MPI

Event Name	Binding	Description	Event Data
event_netop_rdma	None	Network event	Network-op type, Target rank, Size, Local address, Remote address

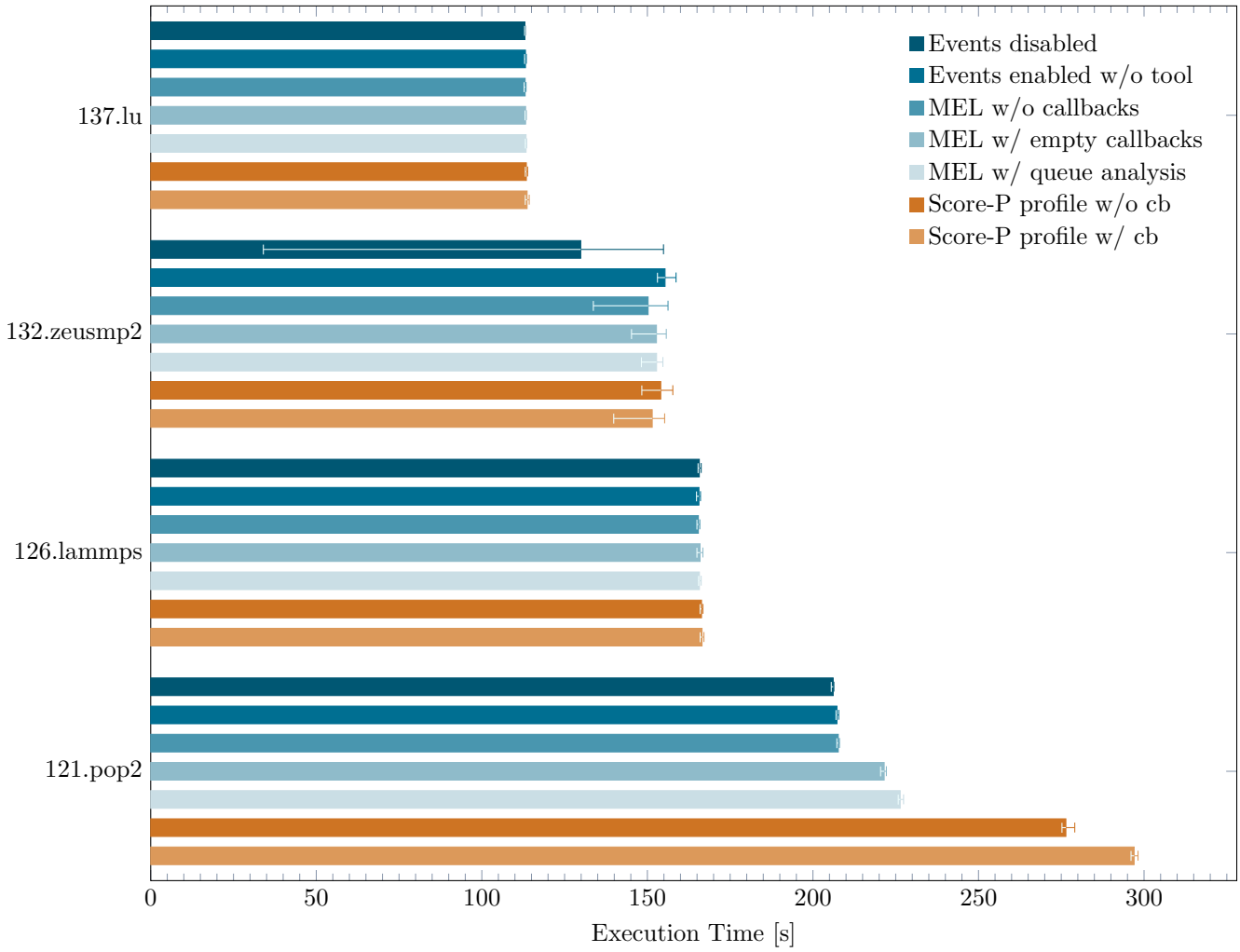


Figure 1: Runtime of selected SPEC MPI2007 v2.0 benchmarks on JURECA [28] with different tool configurations. Bar length indicates the average runtime of 5 measurements. Error bars indicate the minimum and maximum runtime within a measurement set.

query and print the information, without understanding specific elements of the event type and their semantics. While this does not enable automatic processing of events during execution—it relies on the user to interpret the gathered information—it showcases that it is possible for a simple tool to generate useful event information without undue complexity.

For example, by combining the information provided by `MPI_T_event_get_info`, `MPI_T_enum_get_info`, and `MPI_T_enum_get_item`, MEL is capable of providing relevant information, without a specific semantic understanding programmed into the callback itself, as shown by the following partial measurement output of the `ring_c` example provided by Open MPI:

```
[  0.002151416]  'pml_ob1_message_arrived' \
    context id=0 source=0 tag=201 \
    sequence number=10
```

The name in single quotes is the event name and part of the event information. The keys of the key-value pairs and the item names of the provided (optional) enumeration type. The values of the key-value pairs represent the values directly queried within the event callback using `MPI_T_event_read`.

5.2.2. *Profiling the Message Queues*

Open MPI uses two message queues to handle receiving messages efficiently – the posted message queue, containing the message envelope for posted receive operations, and the unexpected message queue for messages without a matching outstanding receive. Understanding the performance characteristics of both queues can help the application developer in a more efficient ordering of send and receive operations.

MEL provides callbacks to profile both the duration of how long individual messages are waiting in the queue for and how much time is spent on searching for messages in the queues. The message queue statistics show the total number of messages entering the queue, the total time the queue was populated, and the maximum length of the queue as well as the average, minimum and maximum time a message stayed in the queue. Events used for the posted queue are *pml_ob1_posted_insert/remove* and *pml_ob1_unex_insert/remove* are used for the unexpected message queue. Output is generated for each rank similar to the following example of a measurement of the ZeusMP/2 (132.zeusmp2) benchmark of the SPEC MPI 2007 [30]

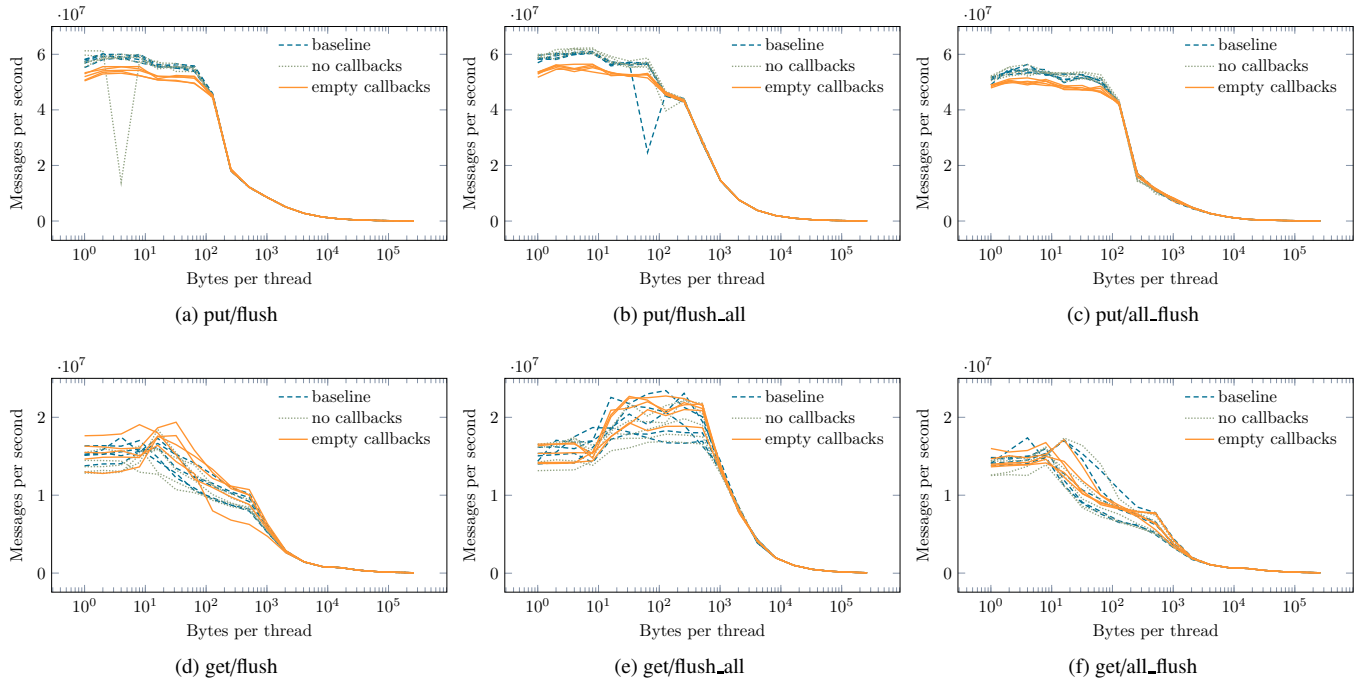


Figure 2: Message rate (Messages per second) over message size (Bytes per thread) for the RMA-MT with 16 threads on JURECA with no tool attached (baseline), MEL attached without callback registration (no callbacks), and MEL attached with empty callbacks registered (empty callbacks).

benchmark suite on 24 processes:

```
[MEL] Posted queue statistics rank: 21 \
Num Messages: 14559 \
Max length of message queue: 14 \
Total time of messages in queue: 625.01 s \
Average time of message in queue: 0.0429294 s \
Min time of message in queue: 5.16e-07 s \
Max time of message in queue: 0.571345 s
```

Figure 3 shows the maximum lifetime of a messages in the posted message queue for the 2 SPEC MPI 2007 benchmarks 121.pop2 (Figure 3 (a-b)) and 137.lu (Figure 3 (c-d)). For each benchmark we show the maximum message lifetime per process on the left and a histogram of the distribution of that time among the processes on the right. We see a very homogenous distribution for POP2 with basically just one bin in the histogram populated and a wave-front pattern in the LU case.

The queue search analysis generates statistics for total search time, the average time per search as well as minimal and maximal search time on each MPI process. It uses the event pairs *pml_ob1_search_posted_begin/end* for the posted queue and *pml_ob1_search_unexpected_begin/end* for the unexpected queue, respectively. Again the output is per MPI process as shown by the output of the analysis of the Zeus-MP/2 benchmark:

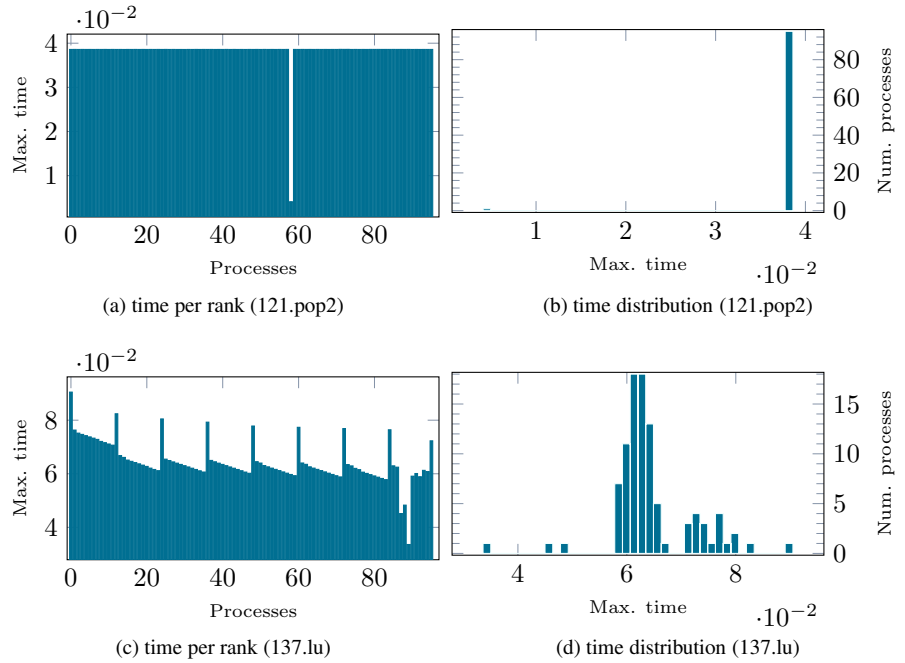
```
[MEL] Unexpected queue search statistics rank: 21 \
Num Searches in queue: 29284 \
Total time searching in queue: 0.0351296 s \
Average time of a search: 1.19962e-06 s \
Min time of a search: 7.8e-08 s \
Max time of a search: 3.0116e-05 s
```

Figure 4 shows the unexpected message queue search results for the two SPEC benchmarks 126.lammps (Figure 4 (a-b)) and 132.zeusmp2 (Figure 4 (c-d)) in a similar way to Figure 3. Here, we cannot determine a specific pattern for any of the benchmarks. In each case the search time varies significantly between the processes with several bins populated in the histogram. In each case the outliers are clearly visible in the histogram. An interesting observation is the significant jump in search time after the first quarter of processes in both benchmarks. This can be a starting point for further investigations.

5.3. Optimizing RMDA-based Messaging

One early success for MPI_T Events came from debugging a performance problem when using Open MPI with the RMA-MT benchmark suite. This benchmark suite consists of latency, bandwidth and bi-directional measurements between a pair of MPI processes. These benchmarks create a user-specified number of threads each performing a single (for latency measurements) or multiple (for bandwidth measurements) `MPI_Put` or `MPI_Get` operation(s). A master thread handles all synchronization (lock, flush, post-start-complete-wait (PSCW), etc.).

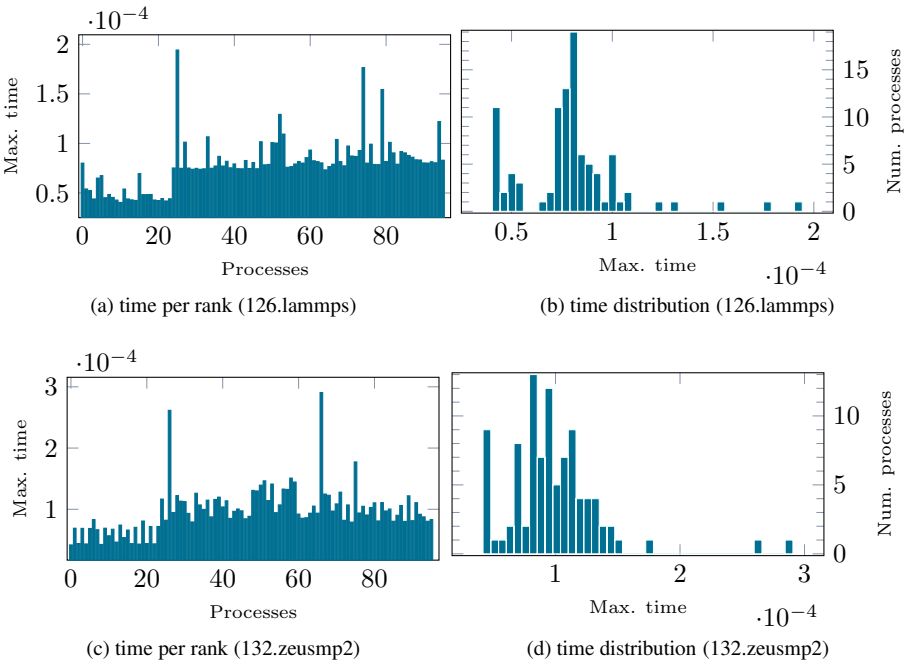
On Cray XC systems we observed a significant drop in the large message ($> 8kB$) bandwidth of `MPI_Put` at higher thread counts (¿ 8 threads). The cause of this drop was unknown and a workaround was added that essentially limits the number of active large put operations. This was working well with the benchmarks. As part of the prototype implementation we added MPI_T events in the *ugni* BTL to trigger when one-sided network operations were started and completed. The RMA-MT benchmarks were updated to create callbacks to print out the size and thread ID when these new event types are triggered. With these event types we were able to determine that without the large message throttling most (in some cases all) of the completion events were being handled by the synchronization thread essentially serializing the completion of network oper-



(a) time per rank (121.pop2) (b) time distribution (121.pop2)

(c) time per rank (137.lu) (d) time distribution (137.lu)

Figure 3: Maximum lifetimes of messages in the posted message queue for 121.pop2 (a-b) and 137.lu (c-d).



(a) time per rank (126.lammps) (b) time distribution (126.lammps)

(c) time per rank (132.zeusmp2) (d) time distribution (132.zeusmp2)

Figure 4: Maximum search times in the unexpected message queue for 126.lammps (a-b) and 132.zeusmp2 (c-d).

ations. With the throttling enabled the handling of the completion events was more balanced between all the benchmark threads. This information will be used to guide future development in the multi-threaded RMA code paths in Open MPI.

This result would not have been possible without the flexibility provided by the MPI_T Events interface. By not specifying events and their semantics it allows MPI implementors to expose the information that is relevant to their implementation and platform.

5.4. Score-P Integration

We also integrate MPI_T Events into Score-P [9] to show its applicability to complex and established tool infrastructures. Score-P is an event-based performance measurement and analysis tool and processes information based on event relationships defined in an event model that enables portable performance analysis across MPI implementations. The MPI_T approach to not define and mandate specific events posed difficulties for the Score-P event model. However, some event types mapped to

Figure 5: Zoomed timeline of an execution of Zeus-MP/2 on 24 processes. Solid blue lines of the Master thread shows execution of an `MPI_Waitall`; Magenta blocks on the location stream below show searches in the posted message queue.

events in the model. Identifying similarity of events within and across MPI implementations and how to handle them in event models such as that of Score-P, OTF2 [25], and Scalasca Trace Analyzer [31] are left as future work.

We implemented a Score-P prototype that records searches in the posted-message queue and the unexpected-message queue via the event pair *pml_ob1_search_posted_begin/end* and the event pair *pml_ob1_search_unexpected_begin/-end* by modeling them as code regions with enter and exit records. Events are recorded on a separate location stream. Figure 5 shows how Vampir [32] displays the event information of a measurement of the Zeus-MP/2 benchmark. The information obtained through the MPI_T events interface reveals where the implementation searches the respective queues during a call of `MPI_Waitall`. Score-P attaches the event information passed to the begin callbacks to the corresponding enter event, which Vampir displays as region attributes (shown on the right).

6. Conclusions

Asynchronous event information can greatly aid in the performance analysis of MPI applications. It enables the detection of causal and temporal relationships within a program's execution, which are not available through synchronous event information or through summarized profiles. However, the current tool interfaces in the MPI Standard do not provide asynchronous data leaving such information unexplored, subject to approximation or heuristics or dependent on implementation or vendor specific extensions — portable tools leveraging event data are not possible.

To close this gap, we propose the MPI_T Events API. It extends and cleanly integrates with the existing MPI_T interface with functions for tools to register asynchronous callbacks for events of interest generated by the MPI implementation. Our proposed API follows the design philosophy of MPI_T and does not prescribe any particular event, but rather lets the MPI implementation decide which events to offer and in what form. Tools can then query the MPI implementation for the events offered as well as their semantic information and with that gain access to the events. Our proposed API addresses many issues surrounding the use of callback APIs in MPI, including the ability to reason about event order, restrictions imposed on callbacks in certain execution contexts as well as the use of extendable type information and callback signatures. Further, a prototype in Open MPI, one of the leading open source MPI implementations, shows that the approach is both feasible and can provide novel and helpful performance data to tools.

In summary, our MPI_T Events proposal closes a clear gap in the current tool interfaces of MPI and can enable a new generation of portable tools. It complements and completes the existing tool APIs and hence equips MPI with new monitoring capabilities already present in other programming models, such as GASNet and OpenMP. This proposal is currently under discussion in the MPI Forum for inclusion in the MPI Standard. We hope that this paper helps further this discussion, as well as spurs the development of new, event-based tools for MPI applications.

Acknowledgment

We thank our colleagues at the MPI Forum and specifically the MPI Forum Tools Working Group for their valuable feedback during the discussion of this interface. This work was partly funded by the Excellence Initiative of the German federal and state governments. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-JRNL-765281. The authors gratefully acknowledge the computing time granted through JARA-HPC on the supercomputer JURECA at Forschungszentrum Jülich.

[1] The Message Passing Interface Forum, MPI: A Message Passing Interface Standard, Version 3.1, 2015.

[2] I. Compres, On-line Application-specific Tuning with the Periscope Tuning Framework and the MPI Tools Interface, Presentation at the 2014 Petascale Tools Workshop, Madison, WI, August 2014.

[3] E. Gallardo, J. Vienne, L. Fialho, P. Teller, J. Browne, MPI Advisor: A Minimal Overhead Tool for MPI Library Performance Tuning, in: Proc. 22nd Eur. MPI Users' Gr. Meet., EuroMPI '15, ACM, New York, NY, USA, 2015, pp. 6:1—-6:10. doi:10.1145/2802658.2802667.

[4] E. Gallardo, J. Vienne, L. Fialho, P. Teller, J. Browne, Employing MPI_T in MPI Advisor to optimize application performance, The International Journal of High Performance Computing Applications 0 (0). doi:10.1177/1094342016684005.

[5] T. Islam, K. Mohror, M. Schulz, Exploring the Capabilities of the New MPI_T Interface, in: Proceedings of the 21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, 2014.

[6] R. Rajachandrasekar, J. Perkins, K. Hamidouche, M. Arnold, D. K. Panda, Understanding the Memory-Utilization of MPI Libraries: Challenges and Designs in Implementing the MPI_T Interface, in: Proc. of the 21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, 2014.

[7] S. Ramesh, A. Mahéo, S. Shende, A. D. Malony, H. Subramoni, D. K. Panda, MPI Performance Engineering with the MPI Tool Interface: The Integration of MVAPICH and TAU, in: Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI '17, 2017.

[8] M.-A. Hermanns, N. T. Hjlem, M. Knobloch, K. Mohror, M. Schulz, Enabling callback-driven runtime introspection via MPI_T, in: Proceedings of the 25th European MPI Users' Group Meeting, EuroMPI'18, ACM, New York, NY, USA, 2018, pp. 8:1–8:10. doi:10.1145/3236367.3236370.

[9] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. S. Shende, R. Tschüter, M. Wagner, B. Wesarg, F. Wolf, Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir, in: H. Brunst, M. S. Müller, W. E. Nagel, M. M. Resch (Eds.), Tools High Perform. Comput. 2011, Springer Berlin Heidelberg, 2012, pp. 79–91. doi:10.1007/978-3-642-31476-6_7.

[10] R. Brightwell, S. Goudy, K. Underwood, A Preliminary Analysis of the MPI Queue Characteristics of Several Applications, in: Proceedings of the 2005 International Conference on Parallel Processing, ICPP '05, IEEE, 2005, pp. 175–183.

[11] J. M. Kunkel, Y. Tsujita, O. Mordvinova, T. Ludwig, Tracing Internal Communication in MPI and MPI-I/O, in: Int. Conf. on Parallel and Distrib. Comp., Applications and Technologies, IEEE, 2009, pp. 280–286.

[12] UPC Consortium, UPC language specifications (Nov. 2013).

[13] D. Bonachea, GASNet specification, Tech. Rep. UCB/CSD-02-1207, Lawrence Berkeley National Laboratory (Nov. 2006).

[14] A. Leko, D. Bonachea, H.-H. Su, A. D. George, GASP : A performance analysis tool interface for global address space programming models, Tech. Rep. LBNL-61606, Lawrence Berkeley National Lab (Sep. 2006).

[15] A. Leko, H.-H. Su, D. Bonachea, B. Golden, M. Billingsley III., A. D. George, Parallel performance wizard: a performance analysis tool for partitioned global-address-space programming models, in: SC '06 Proc. 2006 ACM/IEEE Conf. Supercomput., ACM, New York, NY, USA, 2006, p. 186. doi:10.1145/1188455.1188647.

[16] H.-H. Su, Parallel Performance Wizard: Framework and Techniques for Parallel Application Optimization, Ph.D. thesis, University of Florida (2010).

[17] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, D. Lorenz, OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis, in: A. P. Rendell, B. M. Chapman, M. S. Müller (Eds.), OpenMP Era Low Power Devices Accel., Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 171–185.

[18] OpenMP Architecture Review Board, OpenMP 4.5 Specification, 2015.

[19] R. Dimitrov, A. Skjellum, T. Jones, B. de Supinski, R. Brightwell, C. Janssen, M. Nochumson, PERUSE: An MPI Performance Revealing Extensions Interface, Sixth IBM System Scientific Computing User Group.

[20] T. Jones, B. W. Barrett, D. E. Bernholdt, R. Brightwell, L. A. Bongo, G. Bosilca, A. Cortés, T. Cortés, J. Coyle, B. R. de Supinski, R. Dimitrov, S. Erdogon, H.-C. Hoppe, G. Fagg, F. Geier, J. Gimenez, R. L. Graham, D. Gunter, S. T. Healey, C. Janssen, K. L. Karavanic, R. Keller, B. King-Smith, D. J. Kerbyson, J. Labarta, B. LePore, A. Lumsdaine, C. W. Lee, E. L. Lusk, D. Merril, B. Mohr, K. Mohror, M. S. Müller, B. Noble, R. W. Numrich, P. Ohly, D. K. Panda, K. Pinnow, K. Pajaram, H. Ritzdorf, P. C. Roth, M. Schulz, M. Senar, A. Skjellum, J. Squyres, R. Treumann, T. Woodall, MPI PERUSE: An MPI Extension for Revealing Unexposed Implementation Information, Tech. rep., LLNL (2006).

[21] R. Keller, G. Bosilca, G. Fagg, M. Resch, J. J. Dongarra, Implementation and Usage of the PERUSE-Interface in Open MPI, in: B. Mohr, J. L. Träff, J. Worringen, J. Dongarra (Eds.), Recent Adv. Parallel Virtual Mach. Messag. Passing Interface, Vol. 4192 of LNCS, Springer Berlin Heidelberg, 2006, pp. 347–355. doi:10.1007/11846802_48.

[22] R. Keller, R. L. Graham, Characteristics of the Unexpected Message Queue of MPI Applications, in: R. Keller, E. Gabriel, M. Resch, J. Dongarra (Eds.), Recent Adv. Messag. Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 179–188.

[23] K. A. Brown, J. Domke, S. Matsuoka, Tracing Data Movements Within MPI Collectives, in: Proc. 21st Eur. MPI Users' Gr. Meet., EuroMPI/ASIA '14, ACM, New York, NY, USA, 2014, pp. 117:117—-117:118. doi:10.1145/2642769.2642789.

[24] K. A. Brown, J. Domke, S. Matsuoka, Hardware-Centric Analysis of Network Performance for MPI Applications, in: 2015 IEEE 21st Int. Conf. Parallel Distrib. Syst., 2015, pp. 692–699. doi:10.1109/ICPADS.2015.92.

[25] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, F. Wolf, Open Trace Format 2: The next generation of scalable trace formats and support libraries, Adv. Parallel Comput. 22 (2012) 481–490. doi:10.3233/978-1-61499-041-3-481.

[26] S. K. Gutierrez, N. T. Hjelm, M. G. Venkata, R. L. Graham, Performance evaluation of open mpi on cray xe/xk systems, in: 2012 IEEE 20th Annual Symposium on High-Performance Interconnects, 2012, pp. 40–47. doi:10.1109/HOTI.2012.11.

[27] M. G. F. Dosanjh, T. Groves, R. E. Grant, R. Brightwell, P. G. Bridges, Rma-mt: A benchmark suite for assessing mpi multi-threaded rma performance, in: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016, pp. 550–559. doi:10.1109/CCGrid.2016.84.

[28] Jülich Supercomputing Centre, JURECA: General-purpose supercomputer at Jülich Supercomputing Centre, Journal of large-scale research facilities 2 (A62). doi:10.17815/jlsrf-2-121.

[29] N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. L. Groves, P. G. Bridges, D. C. Arnold, Improving MPI multi-threaded RMA communication performance, in: Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018, 2018, pp. 58:1–58:11. doi:10.1145/3225058.3225114.

[30] M. S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, H. Feng, C. Ponder, SPEC MPI2007 – an application benchmark suite for parallel systems using MPI, Concurrency and Computation: Practice and Experience 22 (2) (2007) 191–205. doi:10.1002/cpe.1535.

[31] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, B. Mohr, The Scalasca performance toolset architecture, Concurrency and Computation: Practice and Experience 22 (6) (2010) 702–719. doi:10.1002/cpe.1556.

[32] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, W. E. Nagel, The Vampir performance analysis tool-set, in: Tools for High Perf. Comp., Springer, 2008, pp. 139–155.