

D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

November 21, 2014

This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

Chapter 14

Tool Support

14.1 Introduction

This chapter discusses interfaces that allow debuggers, performance analyzers, and other tools to extract information about the operation of MPI processes. Specifically, this chapter defines both the MPI profiling interface (Section 14.2), which supports the transparent interception and inspection of MPI calls, and the MPI tool information interface (Section 14.3), which supports the inspection and manipulation of MPI control and performance variables. The interfaces described in this chapter are all defined in the context of an MPI process, i.e., are callable from the same code that invokes other MPI functions.

14.2 Profiling Interface

14.2.1 Requirements

To meet the requirements for the MPI profiling interface, an implementation of the MPI functions *must*

1. provide a mechanism through which all of the MPI defined functions, except those allowed as macros (See Section 2.6.4), may be accessed with a name shift. This requires, in C and Fortran, an alternate entry point name, with the prefix `PMPI_` for each MPI function in each provided language binding and language support method. For routines implemented as macros, it is still required that the `PMPI_` version be supplied and work as expected, but it is not possible to replace at link time the `MPI_` version with a user-defined version.

For Fortran, the different support methods cause several linker names. Therefore, several profiling routines (with these linker names) are needed for each Fortran MPI routine, as described in Section 17.1.5 on page 605.

2. ensure that those MPI functions that are not replaced may still be linked into an executable image without causing name clashes.
3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether she must implement the profile interface for each binding, or can economize by implementing it only for the lowest level routines.

- 1 4. where the implementation of different language bindings is done through a layered
2 approach (e.g., the Fortran binding is a set of “wrapper” functions that call the C
3 implementation), ensure that these wrapper functions are separable from the rest of
4 the library.

5 This separability is necessary to allow a separate profiling library to be correctly
6 implemented, since (at least with Unix linker semantics) the profiling library must
7 contain these wrapper functions if it is to perform as expected. This requirement
8 allows the person who builds the profiling library to extract these functions from the
9 original MPI library and add them into the profiling library without bringing along
10 any other unnecessary code.

- 11 5. provide a no-op routine `MPI_PCONTROL` in the MPI library.

14.2.2 Discussion

15 The objective of the MPI profiling interface is to ensure that it is relatively easy for authors
16 of profiling (and other similar) tools to interface their codes to MPI implementations on
17 different machines.

18 Since MPI is a machine independent standard with many different implementations,
19 it is unreasonable to expect that the authors of profiling tools for MPI will have access to
20 the source code that implements MPI on any particular machine. It is therefore necessary
21 to provide a mechanism by which the implementors of such tools can collect whatever
22 performance information they wish *without* access to the underlying implementation.

23 We believe that having such an interface is important if MPI is to be attractive to end
24 users, since the availability of many different tools will be a significant factor in attracting
25 users to the MPI standard.

26 The profiling interface is just that, an interface. It says *nothing* about the way in which
27 it is used. There is therefore no attempt to lay down what information is collected through
28 the interface, or how the collected information is saved, filtered, or displayed.

29 While the initial impetus for the development of this interface arose from the desire to
30 permit the implementation of profiling tools, it is clear that an interface like that specified
31 may also prove useful for other purposes, such as “internetworking” multiple MPI imple-
32 mentations. Since all that is defined is an interface, there is no objection to its being used
33 wherever it is useful.

34 As the issues being addressed here are intimately tied up with the way in which ex-
35 ecutable images are built, which may differ greatly on different machines, the examples
36 given below should be treated solely as one way of implementing the objective of the MPI
37 profiling interface. The actual requirements made of an implementation are those detailed
38 in the Requirements section above, the whole of the rest of this section is only present as
39 justification and discussion of the logic for those requirements.

40 The examples below show one way in which an implementation could be constructed to
41 meet the requirements on a Unix system (there are doubtless others that would be equally
42 valid).

14.2.3 Logic of the Design

43 Provided that an MPI implementation meets the requirements above, it is possible for
44 the implementor of the profiling system to intercept the MPI calls that are made by the
45 46 47 48

user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

14.2.4 Miscellaneous Control of Profiling

There is a clear requirement for the user code to be able to control the profiler dynamically at run time. This capability is normally used for (at least) the purposes of

- Enabling and disabling profiling depending on the state of the calculation.
- Flushing trace buffers at non-critical points in the calculation.
- Adding user events to a trace file.

These requirements are met by use of `MPI_PCONTROL`.

`MPI_PCONTROL(level, ...)`

IN level Profiling level (integer)

`int MPI_Pcontrol(const int level, ...)`

`MPI_Pcontrol(level) BIND(C)`
`INTEGER, INTENT(IN) :: level`

`MPI_PCONTROL(LEVEL)`
`INTEGER LEVEL`

MPI libraries themselves make no use of this routine, and simply return immediately to the user code. However the presence of calls to this routine allows a profiling package to be explicitly called by the user.

Since MPI has no control of the implementation of the profiling code, we are unable to specify precisely the semantics that will be provided by calls to `MPI_PCONTROL`. This vagueness extends to the number of arguments to the function, and their datatypes.

However to provide some level of portability of user codes to different profiling libraries, we request the following meanings for certain values of `level`.

- `level==0` Profiling is disabled.
- `level==1` Profiling is enabled at a normal default level of detail.
- `level==2` Profile buffers are flushed, which may be a no-op in some profilers.
- All other values of `level` have profile library defined effects and additional arguments.

We also request that the default state after `MPI_INIT` has been called is for profiling to be enabled at the normal default level. (i.e., as if `MPI_PCONTROL` had just been called with the argument 1). This allows users to link with a profiling library and to obtain profile output without having to modify their source code at all.

The provision of `MPI_PCONTROL` as a no-op in the standard MPI library supports the collection of more detailed profiling information with source code that can still link against the standard MPI library.

14.2.5 Profiler Implementation Example

A profiler can accumulate the total amount of data sent by the `MPI_SEND` function, along with the total elapsed time spent in the function as the following example shows:

Example 14.1

```

static int totalBytes = 0;
static double totalTime = 0.0;

int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
{
    double tstart = MPI_Wtime();      /* Pass on all arguments */
    int extent;
    int result = PMPI_Send(buffer, count, datatype, dest, tag, comm);

    totalTime += MPI_Wtime() - tstart; /* and time */

    MPI_Type_size(datatype, &extent); /* Compute size */
    totalBytes += count*extent;

    return result;
}

```

14.2.6 MPI Library Implementation Example

If the MPI library is implemented in C on a Unix system, then there are various options, including the two presented here, for supporting the name-shift requirement. The choice between these two options depends partly on whether the linker and compiler support weak symbols.

Systems with Weak Symbols

If the compiler and linker support weak external symbols (e.g., Solaris 2.x, other System V.4 machines), then only a single library is required as the following example shows:

Example 14.2

```

#pragma weak MPI_Example = PMPI_Example

int PMPI_Example(/* appropriate args */)
{
    /* Useful content */
}

```

The effect of this `#pragma` is to define the external symbol `MPI_Example` as a weak definition. This means that the linker will not complain if there is another definition of the symbol (for instance in the profiling library); however if no other definition exists, then the linker will use the weak definition.

Systems Without Weak Symbols

In the absence of weak symbols then one possible solution would be to use the C macro preprocessor as the following example shows:

Example 14.3

```

#ifdef PROFILELIB
#   ifdef __STDC__
#       define FUNCTION(name) P##name
#   else
#       define FUNCTION(name) P/**/name
#   endif
#else
#   define FUNCTION(name) name
#endif

```

Each of the user visible functions in the library would then be declared thus

```

int FUNCTION(MPI_Example)(/* appropriate args */)
{
    /* Useful content */
}

```

The same source file can then be compiled to produce both versions of the library, depending on the state of the PROFILELIB macro symbol.

It is required that the standard MPI library be built in such a way that the inclusion of MPI functions can be achieved one at a time. This is a somewhat unpleasant requirement, since it may mean that each external function has to be compiled from a separate file. However this is necessary so that the author of the profiling library need only define those MPI functions that she wishes to intercept, references to any others being fulfilled by the normal MPI library. Therefore the link step can look something like this

```
% cc ... -lmyprof -lpmpi -lmpi
```

Here `libmyprof.a` contains the profiler functions that intercept some of the MPI functions, `libpmpi.a` contains the “name shifted” MPI functions, and `libmpi.a` contains the normal definitions of the MPI functions.

14.2.7 Complications**Multiple Counting**

Since parts of the MPI library may themselves be implemented using more basic MPI functions (e.g., a portable implementation of the collective operations implemented using point to point communications), there is potential for profiling functions to be called from within an MPI function that was called from a profiling function. This could lead to “double counting” of the time spent in the inner routine. Since this effect could actually be useful under some circumstances (e.g., it might allow one to answer the question “How much time is spent in the point to point routines when they are called from collective functions?”), we have decided not to enforce any restrictions on the author of the MPI library that would

1 overcome this. Therefore the author of the profiling library should be aware of this problem,
2 and guard against it. In a single-threaded world this is easily achieved through use of a
3 static variable in the profiling code that remembers if you are already inside a profiling
4 routine. It becomes more complex in a multi-threaded environment (as does the meaning
5 of the times recorded).

6 7 Linker Oddities

8
9 The Unix linker traditionally operates in one pass: the effect of this is that functions from
10 libraries are only included in the image if they are needed at the time the library is scanned.
11 When combined with weak symbols, or multiple definitions of the same function, this can
12 cause odd (and unexpected) effects.

13 Consider, for instance, an implementation of MPI in which the Fortran binding is
14 achieved by using wrapper functions on top of the C implementation. The author of the
15 profile library then assumes that it is reasonable only to provide profile functions for the C
16 binding, since Fortran will eventually call these, and the cost of the wrappers is assumed
17 to be small. However, if the wrapper functions are not in the profiling library, then none
18 of the profiled entry points will be undefined when the profiling library is called. Therefore
19 none of the profiling code will be included in the image. When the standard MPI library
20 is scanned, the Fortran wrappers will be resolved, and will also pull in the base versions of
21 the MPI functions. The overall effect is that the code will link successfully, but will not be
22 profiled.

23 To overcome this we must ensure that the Fortran wrapper functions are included in
24 the profiling version of the library. We ensure that this is possible by requiring that these
25 be separable from the rest of the base MPI library. This allows them to be copied out of
26 the base library and into the profiling one using a tool such as `ar`.

27 28 Fortran Support Methods

29 The different Fortran support methods and possible options for the support of subarrays
30 (depending on whether the compiler can support `TYPE(*)`, `DIMENSION(...)` choice buffers)
31 imply different linker names for the same Fortran MPI routine. The rules and implications
32 for the profiling interface are described in Section 17.1.5 on page 605.

34 35 14.2.8 Multiple Levels of Interception

36 The scheme given here does not directly support the nesting of profiling functions, since it
37 provides only a single alternative name for each MPI function. Consideration was given to
38 an implementation that would allow multiple levels of call interception, however we were
39 unable to construct an implementation of this that did not have the following disadvantages

- 40 • assuming a particular implementation language,
- 41 • imposing a run time cost even when no profiling was taking place.

42
43
44 Since one of the objectives of MPI is to permit efficient, low latency implementations, and
45 it is not the business of a standard to require a particular implementation language, we
46 decided to accept the scheme outlined above.

47 Note, however, that it is possible to use the scheme above to implement a multi-level
48 system, since the function called by the user may call many different profiling functions

before calling the underlying MPI function. This capability has been demonstrated in the P^NMPI tool infrastructure [?].

14.3 The MPI Tool Information Interface

MPI implementations often use internal variables to control their operation and performance. Understanding and manipulating these variables can provide a more efficient execution environment or improve performance for many applications. This section describes the MPI tool information interface, which provides a mechanism for MPI implementors to expose variables, each of which represents a particular property, setting, or performance measurement from within the MPI implementation. The interface is split into two parts: the first part provides information about and supports the setting of control variables through which the MPI implementation tunes its configuration. The second part provides access to performance variables that can provide insight into internal performance information of the MPI implementation.

To avoid restrictions on the MPI implementation, the MPI tool information interface allows the implementation to specify which control and performance variables exist. Additionally, the user of the MPI tool information interface can obtain metadata about each available variable, such as its datatype, and a textual description. The MPI tool information interface provides the necessary routines to find all variables that exist in a particular MPI implementation, to query their properties, to retrieve descriptions about their meaning, and to access and, if appropriate, to alter their values. (JEFF: Enumerations, enumeration items with parent enumerations that have equivalent names,) Variables and categories across connected processes with equivalent names are required to have the same meaning (For the definition of equivalent as it relates to strings in this section, please refer to Section 14.3.3). The intent of this requirement is to enforce consistency across connected processes. For example, variables describing the number of packets sent on heterogeneous network devices should have different names to reflect their potentially different meanings. Specification of what must be the same for variables and categories is stated in their respective sections.

The MPI tool information interface can be used independently from the MPI communication functionality. In particular, the routines of this interface can be called before MPI_INIT (or equivalent) and after MPI_FINALIZE. In order to support this behavior cleanly, the MPI tool information interface uses separate initialization and finalization routines. All identifiers used in the MPI tool information interface have the prefix MPI_T_.

On success, all MPI tool information interface routines return MPI_SUCCESS, otherwise they return an appropriate and unique return code indicating the reason why the call was not successfully completed. Details on return codes can be found in Section 14.3.9. However, unsuccessful calls to the MPI tool information interface are not fatal and do not impact the execution of subsequent MPI routines.

Since the MPI tool information interface primarily focuses on tools and support libraries, MPI implementations are only required to provide C bindings for functions and constants introduced in this section. Except where otherwise noted, all conventions and principles governing the C bindings of the MPI API also apply to the MPI tool information interface, which is available by including the mpi.h header file. All routines in this interface have local semantics.

Advice to users. The number and type of control variables and performance variables

can vary between MPI implementations, platforms and different builds of the same implementation on the same platform as well as between runs. Hence, any application relying on a particular variable will not be portable. **Further, there is no guarantee that the number of variables and variable indices are the same across connected processes.**

This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the application's environment. When maximum portability is desired, application programmers should either avoid using the MPI tool information interface or avoid being dependent on the existence of a particular control or performance variable. (*End of advice to users.*)

14.3.1 Verbosity Levels

The MPI tool information interface provides access to internal configuration and performance information through a set of control and performance variables defined by the MPI implementation. Since some implementations may export a large number of variables, variables are classified by a verbosity level that categorizes both their intended audience (end users, performance tuners or MPI implementors) and a relative measure of level of detail (basic, detailed or all). These verbosity levels are described by a single integer. Table 14.1 lists the constants for all possible verbosity levels. The values of the constants are monotonic in the order listed in the table; i.e., `MPI_T_VERBOSITY_USER_BASIC < MPI_T_VERBOSITY_USER_DETAIL < ... < MPI_T_VERBOSITY_MPIDEV_ALL`.

| | |
|--|--|
| <code>MPI_T_VERBOSITY_USER_BASIC</code> | Basic information of interest to users |
| <code>MPI_T_VERBOSITY_USER_DETAIL</code> | Detailed information of interest to users |
| <code>MPI_T_VERBOSITY_USER_ALL</code> | All remaining information of interest to users |
| <code>MPI_T_VERBOSITY_TUNER_BASIC</code> | Basic information required for tuning |
| <code>MPI_T_VERBOSITY_TUNER_DETAIL</code> | Detailed information required for tuning |
| <code>MPI_T_VERBOSITY_TUNER_ALL</code> | All remaining information required for tuning |
| <code>MPI_T_VERBOSITY_MPIDEV_BASIC</code> | Basic information for MPI implementors |
| <code>MPI_T_VERBOSITY_MPIDEV_DETAIL</code> | Detailed information for MPI implementors |
| <code>MPI_T_VERBOSITY_MPIDEV_ALL</code> | All remaining information for MPI implementors |

Table 14.1: MPI tool information interface verbosity levels

14.3.2 Binding MPI Tool Information Interface Variables to MPI Objects

Each MPI tool information interface variable provides access to a particular control setting or performance property of the MPI implementation. A variable may refer to a specific MPI object such as a communicator, datatype, or one-sided communication window, or the variable may refer more generally to the MPI environment of the process. Except for the last case, the variable must be bound to exactly one MPI object before it can be used. Table 14.2 lists all MPI object types to which an MPI tool information interface variable can be bound, together with the matching constant that MPI tool information interface routines return to identify the object type.

Rationale. Some variables have meanings tied to a specific MPI object. Examples include the number of send or receive operations that use a particular datatype, the

| Constant | MPI object |
|---------------------------|---|
| MPI_T_BIND_NO_OBJECT | N/A; applies globally to entire MPI process |
| MPI_T_BIND_MPI_COMM | MPI communicators |
| MPI_T_BIND_MPI_DATATYPE | MPI datatypes |
| MPI_T_BIND_MPI_ERRHANDLER | MPI error handlers |
| MPI_T_BIND_MPI_FILE | MPI file handles |
| MPI_T_BIND_MPI_GROUP | MPI groups |
| MPI_T_BIND_MPI_OP | MPI reduction operators |
| MPI_T_BIND_MPI_REQUEST | MPI requests |
| MPI_T_BIND_MPI_WIN | MPI windows for one-sided communication |
| MPI_T_BIND_MPI_MESSAGE | MPI message object |
| MPI_T_BIND_MPI_INFO | MPI info object |

Table 14.2: Constants to identify associations of variables

number of times a particular error handler has been called, or the communication protocol and “eager limit” used for a particular communicator. Creating a new MPI tool information interface variable for each MPI object would cause the number of variables to grow without bound, since they cannot be reused to avoid naming conflicts. By associating MPI tool information interface variables with a specific MPI object, the MPI implementation only must specify and maintain a single variable, which can then be applied to as many MPI objects of the respective type as created during the program’s execution. (*End of rationale.*)

14.3.3 Convention for Returning Strings

Several MPI tool information interface functions return one or more strings. These functions have two arguments for each string to be returned: an **[OUT]OUT** parameter that identifies a pointer to the buffer in which the string will be returned, and an **[IN/OUT]IN/OUT** parameter to pass the length of the buffer. The user is responsible for the memory allocation of the buffer and must pass the size of the buffer (n) as the length argument. Let n be the length value specified to the function. On return, the function writes at most $n - 1$ of the string’s characters into the buffer, followed by a null terminator. If the returned string’s length is greater than or equal to n , the string will be truncated to $n - 1$ characters. In this case, the length of the string plus one (for the terminating null character) is returned in the length argument. If the user passes the null pointer as the buffer argument or passes 0 as the length argument, the function does not return the string and only returns the length of the string plus one in the length argument. If the user passes the null pointer as the length argument, the buffer argument is ignored and nothing is returned.

Because the string length argument, n , to these routines influences the number of characters returned, and because the value of n could differ across MPI processes, strings returned from query routines that represent the same entity across connected processes must be equivalent according to the following definition. Strings are equivalent if the MPI implementation behaves as if it has an identical character array in each MPI process for the string in question across connected processes. The contents of the returned string will be identical to that of the internal character array, but possibly truncated at any process that supplies an n value to the query routine that is smaller than the length of the internal

1 **character array.**

2 3 14.3.4 Initialization and Finalization

4 The MPI tool information interface requires a separate set of initialization and finalization
5 routines.
6

7
8 `MPI_T_INIT_THREAD(required, provided)`

| | | | |
|----|-----|----------|--|
| 9 | IN | required | desired level of thread support (integer) |
| 10 | | | |
| 11 | OUT | provided | provided level of thread support (integer) |

12
13 `int MPI_T_init_thread(int required, int *provided)`

14
15 All programs or tools that use the MPI tool information interface must initialize the
16 MPI tool information interface in the processes that will use the interface before calling
17 any other of its routines. A user can initialize the MPI tool information interface by calling
18 `MPI_T_INIT_THREAD`, which can be called multiple times. In addition, this routine initial-
19 izes the thread environment for all routines in the MPI tool information interface. Calling
20 this routine when the MPI tool information interface is already initialized has no effect
21 beyond increasing the reference count of how often the interface has been initialized. The
22 argument `required` is used to specify the desired level of thread support. The possible values
23 and their semantics are identical to the ones that can be used with `MPI_INIT_THREAD`
24 listed in Section 12.4. The call returns in `provided` information about the actual level of
25 thread support that will be provided by the MPI implementation for calls to MPI tool
26 information interface routines. It can be one of the four values listed in Section 12.4.

27 The MPI specification does not require all MPI processes to exist before the call to
28 `MPI_INIT`. If the MPI tool information interface is used before `MPI_INIT` has been called,
29 the user is responsible for ensuring that the MPI tool information interface is initialized on
30 all processes it is used in. Processes created by the MPI implementation during `MPI_INIT`
31 inherit the status of the MPI tool information interface (whether it is initialized or not as
32 well as all active sessions and handles) from the process from which they are created.

33 Processes created at runtime as a result of calls to MPI's dynamic process management
34 require their own initialization before they can use the MPI tool information interface.

35 *Advice to users.* If `MPI_T_INIT_THREAD` is called before `MPI_INIT_THREAD`,
36 the requested and granted thread level for `MPI_T_INIT_THREAD` may influence the
37 behavior and return value of `MPI_INIT_THREAD`. The same is true for the reverse
38 order. (*End of advice to users.*)

39
40 *Advice to implementors.* MPI implementations should strive to make as many control
41 or performance variables available before `MPI_INIT` (instead of adding them within
42 `MPI_INIT`) to allow tools the most flexibility. In particular, control variables should
43 be available before `MPI_INIT` if their value cannot be changed after `MPI_INIT`. (*End*
44 *of advice to implementors.*)
45
46
47

48 `MPI_T_FINALIZE()`

```
int MPI_T_finalize(void)
```

This routine finalizes the use of the MPI tool information interface and may be called as often as the corresponding `MPI_T_INIT_THREAD` routine up to the current point of execution. Calling it more times returns a corresponding error code. As long as the number of calls to `MPI_T_FINALIZE` is smaller than the number of calls to `MPI_T_INIT_THREAD` up to the current point of execution, the MPI tool information interface remains initialized and calls to its routines are permissible. Further, additional calls to `MPI_T_INIT_THREAD` after one or more calls to `MPI_T_FINALIZE` are permissible.

Once `MPI_T_FINALIZE` is called the same number of times as the routine `MPI_T_INIT_THREAD` up to the current point of execution, the MPI tool information interface is no longer initialized. The interface can be reinitialized by subsequent calls to `MPI_T_INIT_THREAD`.

At the end of the program execution, unless `MPI_ABORT` is called, an application must have called `MPI_T_INIT_THREAD` and `MPI_T_FINALIZE` an equal number of times.

14.3.5 Datatype System

All variables managed through the MPI tool information interface represent their values through typed buffers of a given length and type using an MPI datatype (similar to regular send/receive buffers). Since the initialization of the MPI tool information interface is separate from the initialization of MPI, MPI tool information interface routines can be called before `MPI_INIT`. Consequently, these routines can also use MPI datatypes before `MPI_INIT`. Therefore, within the context of the MPI tool information interface, it is permissible to use a subset of MPI datatypes as specified below before a call to `MPI_INIT` (or equivalent).

```
MPI_INT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_UNSIGNED_LONG_LONG
MPI_COUNT
MPI_CHAR
MPI_DOUBLE
```

Table 14.3: MPI datatypes that can be used by the MPI tool information interface

Rationale. The MPI tool information interface relies mainly on unsigned datatypes for integer values since most variables are expected to represent counters or resource sizes. `MPI_INT` is provided for additional flexibility and is expected to be used mainly for control variables and enumeration types (see below).

Providing all basic datatypes, in particular providing all signed and unsigned variants of integer types, would lead to a larger number of types, which tools need to interpret. This would cause unnecessary complexity in the implementation of tools based on the MPI tool information interface. (*End of rationale.*)

The MPI tool information interface only relies on a subset of the basic MPI datatypes and does not use any derived MPI datatypes. Table 14.3 lists all MPI datatypes that can be returned by the MPI tool information interface to represent its variables.

The use of the datatype `MPI_CHAR` in the MPI tool information interface implies a null-terminated character array, i.e., a string in the C language. If a variable has type `MPI_CHAR`, the value of the count parameter returned by `MPI_T_CVAR_HANDLE_ALLOC` and `MPI_T_PVAR_HANDLE_ALLOC` must be large enough to include any valid value, including its terminating null character. The contents of returned `MPI_CHAR` arrays are only defined from index 0 through the location of the first null character.

Rationale. The MPI tool information interface requires a significantly simpler type system than MPI itself. Therefore, only its required subset must be present before `MPI_INIT` (or equivalent) and MPI implementations do not need to initialize the complete MPI datatype system. (*End of rationale.*)

For variables of type `MPI_INT`, an MPI implementation can provide additional information by associating names with a fixed number of values. We refer to this information in the following as an enumeration. In this case, the respective calls that provide additional metadata for each control or performance variable, i.e., `MPI_T_CVAR_GET_INFO` (Section 14.3.6) and `MPI_T_PVAR_GET_INFO` (Section 14.3.7), return a handle of type `MPI_T_enum` that can be passed to the following functions to extract additional information. Thus, the MPI implementation can describe variables with a fixed set of values that each represents a particular state. Each enumeration type can have N different values, with a fixed N that can be queried using `MPI_T_ENUM_GET_INFO`.

```
MPI_T_ENUM_GET_INFO(enumtype, num, name, name_len)
```

| | | |
|-------|-----------------------|---|
| IN | <code>enumtype</code> | enumeration to be queried (handle) |
| OUT | <code>num</code> | number of discrete values represented by this enumeration (integer) |
| OUT | <code>name</code> | buffer to return the string containing the name of the enumeration (string) |
| INOUT | <code>name_len</code> | length of the string and/or buffer for name (integer) |

```
int MPI_T_enum_get_info(MPI_T_enum enumtype, int *num, char *name, int
                        *name_len)
```

If `enumtype` is a valid enumeration, this routine returns the number of items represented by this enumeration type as well as its name. N must be greater than 0, i.e., the enumeration must represent at least one value.

The arguments `name` and `name_len` are used to return the name of the enumeration as described in Section 14.3.3.

The routine is required to return a name of at least length one. This name must be unique with respect to all other names for enumerations that the MPI implementation uses.

Names associated with individual values in each enumeration `enumtype` can be queried using `MPI_T_ENUM_GET_ITEM`.

| | | | |
|---|----------|---|-------------|
| MPI_T_ENUM_GET_ITEM(enumtype, index, value, name, name_len) | | | 1 |
| IN | enumtype | enumeration to be queried (handle) | 2 |
| IN | index | number of the value to be queried in this enumeration (integer) | 3 4 5 |
| OUT | value | variable value (integer) | 6 |
| OUT | name | buffer to return the string containing the name of the enumeration item (string) | 7 8 9 |
| INOUT | name_len | length of the string and/or buffer for name (integer) | 10 11 |
| int MPI_T_enum_get_item(MPI_T_enum enumtype, int index, int *value, char *name, int *name_len) | | | 12 13 |

The arguments name and name_len are used to return the name of the enumeration item as described in Section 14.3.3.

If completed successfully, the routine returns the name/value pair that describes the enumeration at the specified index. The call is further required to return a name of at least length one. This name must be unique with respect to all other names of items for the same enumeration.

14.3.6 Control Variables

The routines described in this section of the MPI tool information interface specification focus on the ability to list, query, and possibly set control variables exposed by the MPI implementation. These variables can typically be used by the user to fine tune properties and configuration settings of the MPI implementation. On many systems, such variables can be set using environment variables, although other configuration mechanisms may be available, such as configuration files or central configuration registries. A typical example that is available in several existing MPI implementations is the ability to specify an “eager limit,” i.e., an upper bound on the size of messages sent or received using an eager protocol.

Control Variable Query Functions

An MPI implementation exports a set of N control variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any control variables, otherwise the provided control variables are indexed from 0 to $N - 1$. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of control variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a control variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

Advice to users. While the MPI tool information interface guarantees that indices or variable properties do not change during a particular run of an MPI program, it does not provide a similar guarantee between runs. (*End of advice to users.*)

The following function can be used to query the number of control variables, *num_cvar*:

```
1
2
3
4 MPI_T_CVAR_GET_NUM(num_cvar)
```

```
5     OUT     num_cvar           returns number of control variables (integer)
```

```
6
7
8 int MPI_T_cvar_get_num(int *num_cvar)
```

The function `MPI_T_CVAR_GET_INFO` provides access to additional information for each variable.

```
9
10
11
12 MPI_T_CVAR_GET_INFO(cvar_index, name, name_len, verbosity, datatype, enumtype, desc,
13                    desc_len, bind, scope)
```

```
14
15     IN     cvar_index         index of the control variable to be queried, value be-
16                                     tween 0 and num_cvar - 1 (integer)
```

```
17     OUT     name             buffer to return the string containing the name of the
18                                     control variable (string)
```

```
19     INOUT   name_len        length of the string and/or buffer for name (integer)
```

```
20     OUT     verbosity       verbosity level of this variable (integer)
```

```
21     OUT     datatype        MPI datatype of the information stored in the control
22                                     variable (handle)
```

```
23
24     OUT     enumtype        optional descriptor for enumeration information (han-
25                                     dle)
```

```
26     OUT     desc            buffer to return the string containing a description of
27                                     the control variable (string)
```

```
28
29     INOUT   desc_len        length of the string and/or buffer for desc (integer)
```

```
30     OUT     bind            type of MPI object to which this variable must be
31                                     bound (integer)
```

```
32     OUT     scope           scope of when changes to this variable are possible
33                                     (integer)
```

```
34
35 int MPI_T_cvar_get_info(int cvar_index, char *name, int *name_len, int
36                        *verbosity, MPI_Datatype *datatype, MPI_T_enum *enumtype, char
37                        *desc, int *desc_len, int *bind, int *scope)
```

After a successful call to `MPI_T_CVAR_GET_INFO` for a particular variable, subsequent calls to this routine that query information about the same variable must return the same information. An MPI implementation is not allowed to alter any of the returned values.

If any OUT parameter to `MPI_T_CVAR_GET_INFO` is a NULL pointer, the implementation will ignore the parameter and not return a value for the parameter.

The arguments `name` and `name_len` are used to return the name of the control variable as described in Section 14.3.3.

If completed successfully, the routine is required to return a name of at least length one. The name must be unique with respect to all other names for control variables used by the MPI implementation.

The argument `verbosity` returns the verbosity level of the variable (see Section 14.3.1).

The argument `datatype` returns the MPI datatype that is used to represent the control variable.

If the variable is of type `MPI_INT`, MPI can optionally specify an enumeration for the values represented by this variable and return it in `enumtype`. In this case, MPI returns an enumeration identifier, which can then be used to gather more information as described in Section 14.3.5. Otherwise, `enumtype` is set to `MPI_T_ENUM_NULL`. If the datatype is not `MPI_INT` or the argument `enumtype` is the null pointer, no enumeration type is returned.

The arguments `desc` and `desc_len` are used to return a description of the control variable as described in Section 14.3.3.

Returning a description is optional. If an MPI implementation does not to return a description, the first character for `desc` must be set to the null character and `desc_len` must be set to one at the return of this call.

The parameter `bind` returns the type of the MPI object to which the variable must be bound or the value `MPI_T_BIND_NO_OBJECT` (see Section 14.3.2).

The scope of a variable determines whether changing a variable's value is either local to the process or must be done by the user across multiple processes. The latter is further split into variables that require changes in a group of processes and those that require collective changes among all connected processes. Both cases can require all processes either to be set to consistent (but potentially different) values or to equal values on every participating process. The description provided with the variable must contain an explanation about the requirements and/or restrictions for setting the particular variable.

On successful return from `MPI_T_CVAR_GET_INFO`, the argument `scope` will be set to one of the constants listed in Table 14.4.

If the name of a control variable is equivalent across connected processes, the following OUT parameters must be identical: `verbosity`, `datatype`, `enumtype`, `bind`, and `scope`. The `desc` parameter must also be equivalent.

| Scope Constant | Description |
|-----------------------------------|---|
| <code>MPI_T_SCOPE_CONSTANT</code> | read-only, value is constant |
| <code>MPI_T_SCOPE_READONLY</code> | read-only, cannot be written, but can change |
| <code>MPI_T_SCOPE_LOCAL</code> | may be writeable, writing is a local operation |
| <code>MPI_T_SCOPE_GROUP</code> | may be writeable, must be done to a group of processes, all processes in a group must be set to consistent values |
| <code>MPI_T_SCOPE_GROUP_EQ</code> | may be writeable, must be done to a group of processes, all processes in a group must be set to the same value |
| <code>MPI_T_SCOPE_ALL</code> | may be writeable, must be done to all processes, all connected processes must be set to consistent values |
| <code>MPI_T_SCOPE_ALL_EQ</code> | may be writeable, must be done to all processes, all connected processes must be set to the same value |

Table 14.4: Scopes for control variables

Advice to users. The `scope` of a variable only indicates if a variable might be changeable; it is not a guarantee that it can be changed at any time. (*End of advice to users.*)

```
1
2
3 MPI_T_CVAR_GET_INDEX(name, cvar_index)
```

```
4
5     IN      name                name of the control variable (string)
```

```
6     OUT     cvar_index          index of the control variable (integer)
```

```
7
8 int MPI_T_cvar_get_index(const char *name, int *cvar_index)
```

```
9
10 MPI_T_CVAR_GET_INDEX is a function for retrieving the index of a control variable
11 given a known variable name. The name parameter is provided by the caller, and cvar_index
12 is returned by the MPI implementation. The name parameter is a string terminated with a
13 null character.
```

```
14 This routine returns MPI_SUCCESS on success and returns MPI_T_ERR_INVALID_NAME
15 if name does not match the name of any control variable provided by the implementation
16 at the time of the call.
```

```
17
18 Rationale. This routine is provided to enable fast retrieval of control variables by
19 a tool, assuming it knows the name of the variable for which it is looking. The
20 number of variables exposed by the implementation can change over time, so it is not
21 possible for the tool to simply iterate over the list of variables once at initialization.
22 Although using MPI implementation specific variable names is not portable across MPI
23 implementations, tool developers may choose to take this route for lower overhead at
24 runtime because the tool will not have to iterate over the entire set of variables to
25 find a specific one. (End of rationale.)
```

```
26
27 Example: Printing All Control Variables
```

30 Example 14.4

```
31 The following example shows how the MPI tool information interface can be used to
32 query and to print the names of all available control variables.
```

```
33
34 #include <stdio.h>
35 #include <stdlib.h>
36 #include <mpi.h>
37
38 int main(int argc, char *argv[]) {
39     int i, err, num, namelen, bind, verbose, scope;
40     int threadsupport;
41     char name[100];
42     MPI_Datatype datatype;
43
44     err=MPI_T_init_thread(MPI_THREAD_SINGLE,&threadsupport);
45     if (err!=MPI_SUCCESS)
46         return err;
47
48     err=MPI_T_cvar_get_num(&num);
```

```

if (err!=MPI_SUCCESS)
    return err;

for (i=0; i<num; i++) {
    namelen=100;
    err=MPI_T_cvar_get_info(i, name, &namelen,
        &verbose, &datatype, NULL,
        NULL, NULL, /*no description */
        &bind, &scope);
    if (err!=MPI_SUCCESS || err!=MPI_T_ERR_INVALID_INDEX) return err;
    printf("Var %i: %s\n", i, name);
}

err=MPI_T_finalize();
if (err!=MPI_SUCCESS)
    return 1;
else
    return 0;
}

```

Handle Allocation and Deallocation

Before reading or writing the value of a variable, a user must first allocate a handle of type `MPI_T_cvar_handle` for the variable by binding it to an MPI object (see also Section 14.3.2).

Rationale. Handles used in the MPI tool information interface are distinct from handles used in the remaining parts of the MPI standard because they must be usable before `MPI_INIT` and after `MPI_FINALIZE`. Further, accessing handles, in particular for performance variables, can be time critical and having a separate handle space enables optimizations. (*End of rationale.*)

`MPI_T_CVAR_HANDLE_ALLOC(cvar_index, obj_handle, handle, count)`

| | | |
|-----|-------------------------|--|
| IN | <code>cvar_index</code> | index of control variable for which handle is to be allocated (index) |
| IN | <code>obj_handle</code> | reference to a handle of the MPI object to which this variable is supposed to be bound (pointer) |
| OUT | <code>handle</code> | allocated handle (handle) |
| OUT | <code>count</code> | number of elements used to represent this variable (integer) |

```

int MPI_T_cvar_handle_alloc(int cvar_index, void *obj_handle,
    MPI_T_cvar_handle *handle, int *count)

```

This routine binds the control variable specified by the argument `index` to an MPI object. The object is passed in the argument `obj_handle` as an address to a local variable that stores the object's handle. The argument `obj_handle` is ignored if the `MPI_T_CVAR_GET_INFO`

1 call for this control variable returned `MPI_T_BIND_NO_OBJECT` in the argument `bind`. The
 2 handle allocated to reference the variable is returned in the argument
 3 `handle`. Upon successful return, `count` contains the number of elements (of the datatype
 4 returned by a previous `MPI_T_CVAR_GET_INFO` call) used to represent this variable.

5
 6 *Advice to users.* The `count` can be different based on the MPI object to which the
 7 control variable was bound. For example, variables bound to communicators could
 8 have a count that matches the size of the communicator.

9 It is not portable to pass references to predefined MPI object handles, such as
 10 `MPI_COMM_WORLD` to this routine, since their implementation depends on the MPI
 11 library. Instead, such object handles should be stored in a local variable and the
 12 address of this local variable should be passed into `MPI_T_CVAR_HANDLE_ALLOC`.
 13 (*End of advice to users.*)

14
 15 The value of `cvar_index` should be in the range 0 to `num_cvar - 1`, where `num_cvar`
 16 is the number of available control variables as determined from a prior call to
 17 `MPI_T_CVAR_GET_NUM`. The type of the MPI object it references must be consistent
 18 with the type returned in the `bind` argument in a prior call to `MPI_T_CVAR_GET_INFO`.

19 In the case that the `bind` argument returned by `MPI_T_CVAR_GET_INFO` equals
 20 `MPI_T_BIND_NO_OBJECT`, the argument `obj_handle` is ignored.

21
 22
 23 `MPI_T_CVAR_HANDLE_FREE(handle)`

24 INOUT `handle` handle to be freed (`handle`)

25
 26 `int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)`

27
 28 When a handle is no longer needed, a user of the MPI tool information interface should
 29 call `MPI_T_CVAR_HANDLE_FREE` to free the handle and the associated resources in the
 30 MPI implementation. On a successful return, MPI sets the handle to
 31 `MPI_T_CVAR_HANDLE_NULL`.

32
 33 Control Variable Access Functions

34
 35
 36 `MPI_T_CVAR_READ(handle, buf)`

37 IN `handle` handle to the control variable to be read (`handle`)
 38 OUT `buf` initial address of storage location for variable value
 39 (choice)

40
 41
 42 `int MPI_T_cvar_read(MPI_T_cvar_handle handle, void* buf)`

43
 44 This routine queries the value of a control variable identified by the argument `handle` and
 45 stores the result in the buffer identified by the parameter `buf`. The user must ensure that the
 46 buffer is of the appropriate size to hold the entire value of the control variable (based on the
 47 returned datatype and count from prior corresponding calls to `MPI_T_CVAR_GET_INFO`
 48 and `MPI_T_CVAR_HANDLE_ALLOC`, respectively).

MPI_T_CVAR_WRITE(handle, buf)

| | | |
|----|--------|---|
| IN | handle | handle to the control variable to be written (handle) |
| IN | buf | initial address of storage location for variable value (choice) |

```
int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void* buf)
```

This routine sets the value of the control variable identified by the argument `handle` to the data stored in the buffer identified by the parameter `buf`. The user must ensure that the buffer is of the appropriate size to hold the entire value of the control variable (based on the returned datatype and count from prior corresponding calls to `MPI_T_CVAR_GET_INFO` and `MPI_T_CVAR_HANDLE_ALLOC`, respectively).

If the variable has a global scope (as returned by a prior corresponding `MPI_T_CVAR_GET_INFO` call), any write call to this variable must be issued by the user in all connected (as defined in Section 10.5.4) MPI processes. If the variable has group scope, any write call to this variable must be issued by the user in all MPI processes in the group, which must be described by the MPI implementation in the description by the `MPI_T_CVAR_GET_INFO`.

In both cases, the user must ensure that the writes in all processes are consistent. If the scope is either `MPI_T_SCOPE_ALL_EQ` or `MPI_T_SCOPE_GROUP_EQ` this means that the variable in all processes must be set to the same value.

If it is not possible to change the variable at the time the call is made, the function returns either `MPI_T_ERR_CVAR_SET_NOT_NOW`, if there may be a later time at which the variable could be set, or `MPI_T_ERR_CVAR_SET_NEVER`, if the variable cannot be set for the remainder of the application's execution.

Example: Reading the Value of a Control Variable

Example 14.5

The following example shows a routine that can be used to query the value with a control variable with a given index. The example assumes that the variable is intended to be bound to an MPI communicator.

```
int getValue_int_comm(int index, MPI_Comm comm, int *val) {
int err, count;
MPI_T_cvar_handle handle;

/* This example assumes that the variable index */
/* can be bound to a communicator */

err=MPI_T_cvar_handle_alloc(index,&comm,&handle,&count);
if (err!=MPI_SUCCESS) return err;

/* The following assumes that the variable is */
/* represented by a single integer */

err=MPI_T_cvar_read(handle, val);
```

```

1  if (err!=MPI_SUCCESS) return err;
2
3  err=MPI_T_cvar_handle_free(&handle);
4  return err;
5  }
6

```

14.3.7 Performance Variables

The following section focuses on the ability to list and to query performance variables provided by the MPI implementation. Performance variables provide insight into MPI implementation specific internals and can represent information such as the state of the MPI implementation (e.g., waiting blocked, receiving, not active), aggregated timing data for submodules, or queue sizes and lengths.

Rationale. The interface for performance variables is separate from the interface for control variables, since performance variables have different requirements and parameters. By keeping them separate, the interface provides cleaner semantics and allows for more performance optimization opportunities. (*End of rationale.*)

Performance Variable Classes

Each performance variable is associated with a class that describes its basic semantics, possible datatypes, basic behavior, its starting value, whether it can overflow, and when and how an MPI implementation can change the variable's value. The starting value is the value that is assigned to the variable the first time that it is used or whenever it is reset.

Advice to users. If a performance variable belongs to a class that can overflow, it is up to the user to protect against this overflow, e.g., by frequently reading and resetting the variable value. (*End of advice to users.*)

Advice to implementors. MPI implementations should use large enough datatypes for each performance variable to avoid overflows under normal circumstances. (*End of advice to implementors.*)

The classes are defined by the following constants:

- MPI_T_PVAR_CLASS_STATE

A performance variable in this class represents a set of discrete states. Variables of this class are represented by MPI_INT and can be set by the MPI implementation at any time. Variables of this type should be described further using an enumeration, as discussed in Section 14.3.5. The starting value is the current state of the implementation at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

- MPI_T_PVAR_CLASS_LEVEL

A performance variable in this class represents a value that describes the utilization level of a resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_DOUBLE. The starting value

is the current utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

- **MPI_T_PVAR_CLASS_SIZE**

A performance variable in this class represents a value that is the **[fixed]size** of a resource. Values returned from variables in this class are non-negative and represented by one of the following datatypes: `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_UNSIGNED_LONG_LONG`, `MPI_DOUBLE`. The starting value is the current **[utilization level]size** of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

- **MPI_T_PVAR_CLASS_PERCENTAGE**

The value of a performance variable in this class represents the percentage utilization of a finite resource. The value of a variable of this class can change at any time to match the current utilization level of the resource. It will be returned as an `MPI_DOUBLE` datatype. The value must always be between 0.0 (resource not used at all) and 1.0 (resource completely used). The starting value is the current percentage utilization level of the resource at the time that the starting value is set. MPI implementations must ensure that variables of this class cannot overflow.

- **MPI_T_PVAR_CLASS_HIGHWATERMARK**

A performance variable in this class represents a value that describes the high watermark utilization of a resource. The value of a variable of this class is non-negative and grows monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_UNSIGNED_LONG_LONG`, `MPI_DOUBLE`. The starting value is the current utilization level of the resource at the time that the **[starting value is set]variable is started or reset**. MPI implementations must ensure that variables of this class cannot overflow.

- **MPI_T_PVAR_CLASS_LOWWATERMARK**

A performance variable in this class represents a value that describes the low watermark utilization of a resource. The value of a variable of this class is non-negative and decreases monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_UNSIGNED_LONG_LONG`, `MPI_DOUBLE`. The starting value is the current utilization level of the resource at the time that the **[starting value is set]variable is started or reset**. MPI implementations must ensure that variables of this class cannot overflow.

- **MPI_T_PVAR_CLASS_COUNTER**

A performance variable in this class counts the number of occurrences of a specific event (e.g., the number of memory allocations within an MPI library). The value of a variable of this class increases monotonically from the initialization or reset of the performance variable by one for each specific event that is observed. Values must be non-negative and represented by one of the following datatypes: `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_UNSIGNED_LONG_LONG`. The starting value for variables of this class is 0. Variables of this class can overflow.

- **MPI_T_PVAR_CLASS_AGGREGATE**

The value of a performance variable in this class is an aggregated value that represents a sum of arguments processed during a specific event (e.g., the amount

of memory allocated by all memory allocations). This class is similar to the counter class, but instead of counting individual events, the value can be incremented by arbitrary amounts. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_UNSIGNED_LONG_LONG`, `MPI_DOUBLE`. The starting value for variables of this class is 0. Variables of this class can overflow.

- `MPI_T_PVAR_CLASS_TIMER`

The value of a performance variable in this class represents the aggregated time that the MPI implementation spends executing a particular event, type of event, or section of the MPI library. This class has the same basic semantics as `MPI_T_PVAR_CLASS_AGGREGATE`, but explicitly records a timing value. The value of a variable of this class increases monotonically from the initialization or reset of the performance variable. It must be non-negative and represented by one of the following datatypes: `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_UNSIGNED_LONG_LONG`, `MPI_DOUBLE`. The starting value for variables of this class is 0. If the type `MPI_DOUBLE` is used, the units that represent time in this datatype must match the units used by `MPI_WTIME`. Otherwise, the time units should be documented, e.g., in the description returned by `MPI_T_PVAR_GET_INFO`. Variables of this class can overflow.

- `MPI_T_PVAR_CLASS_GENERIC`

This class can be used to describe a variable that does not fit into any of the other classes. For variables in this class, the starting value is variable-specific and implementation-defined.

Performance Variable Query Functions

An MPI implementation exports a set of N performance variables through the MPI tool information interface. If N is zero, then the MPI implementation does not export any performance variables; otherwise the provided performance variables are indexed from 0 to $N - 1$. This index number is used in subsequent calls to identify the individual variables.

An MPI implementation is allowed to increase the number of performance variables during the execution of an MPI application when new variables become available through dynamic loading. However, MPI implementations are not allowed to change the index of a performance variable or to delete a variable once it has been added to the set. When a variable becomes inactive, e.g., through dynamic unloading, accessing its value should return a corresponding error code.

The following function can be used to query the number of performance variables, N :

```
MPI_T_PVAR_GET_NUM(num_pvar)
```

```
OUT    num_pvar           returns number of performance variables (integer)
```

```
int MPI_T_pvar_get_num(int *num_pvar)
```

The function `MPI_T_PVAR_GET_INFO` provides access to additional information for each variable.

| | | | |
|-------|------------|--|----|
| | | MPI_T_PVAR_GET_INFO(pvar_index, name, name_len, verbosity, varclass, datatype, enumtype, desc, desc_len, bind, readonly, continuous, atomic) | 1 |
| | | | 2 |
| IN | pvar_index | index of the performance variable to be queried be- tween 0 and <i>num_pvar</i> - 1 (integer) | 3 |
| | | | 4 |
| OUT | name | buffer to return the string containing the name of the performance variable (string) | 5 |
| | | | 6 |
| INOUT | name_len | length of the string and/or buffer for name (integer) | 7 |
| | | | 8 |
| OUT | verbosity | verbosity level of this variable (integer) | 9 |
| | | | 10 |
| OUT | var_class | class of performance variable (integer) | 11 |
| | | | 12 |
| OUT | datatype | MPI datatype of the information stored in the perfor- mance variable (handle) | 13 |
| | | | 14 |
| OUT | enumtype | optional descriptor for enumeration information (han- dle) | 15 |
| | | | 16 |
| OUT | desc | buffer to return the string containing a description of the performance variable (string) | 17 |
| | | | 18 |
| INOUT | desc_len | length of the string and/or buffer for desc (integer) | 19 |
| | | | 20 |
| OUT | bind | type of MPI object to which this variable must be bound (integer) | 21 |
| | | | 22 |
| OUT | readonly | flag indicating whether the variable can be written/reset (integer) | 23 |
| | | | 24 |
| OUT | continuous | flag indicating whether the variable can be started and stopped or is continuously active (integer) | 25 |
| | | | 26 |
| OUT | atomic | flag indicating whether the variable can be atomically read and reset (integer) | 27 |
| | | | 28 |
| | | | 29 |
| | | | 30 |
| | | int MPI_T_pvar_get_info(int pvar_index, char *name, int *name_len, int *verbosity, int *var_class, MPI_Datatype *datatype, MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind, int *readonly, int *continuous, int *atomic) | 31 |
| | | | 32 |
| | | | 33 |
| | | | 34 |

After a successful call to MPI_T_PVAR_GET_INFO for a particular variable, subsequent calls to this routine that query information about the same variable must return the same information. An MPI implementation is not allowed to alter any of the returned values. ticket378.

If any OUT parameter to MPI_T_PVAR_GET_INFO is a NULL pointer, the implementation will ignore the parameter and not return a value for the parameter.

The arguments name and name_len are used to return the name of the performance variable as described in Section 14.3.3. If completed successfully, the routine is required to return a name of at least length one.

The argument verbosity returns the verbosity level of the variable (see Section 14.3.1).

The class of the performance variable is returned in the parameter var_class. The class must be one of the constants defined in Section 14.3.7.

The combination of the name and the class of the performance variable must be unique with respect to all other names for performance variables used by the MPI implementation.

1 *Advice to implementors.* Groups of variables that belong closely together, but have
 2 different classes, can have the same name. This choice is useful, e.g., to refer to
 3 multiple variables that describe a single resource (like the level, the total size, as well
 4 as high and low watermarks). (*End of advice to implementors.*)

5
 6 The argument `datatype` returns the MPI datatype that is used to represent the perfor-
 7 mance variable.

8 If the variable is of type `MPI_INT`, MPI can optionally specify an enumeration for the
 9 values represented by this variable and return it in `enumtype`. In this case, MPI returns an
 10 enumeration identifier, which can then be used to gather more information as described in
 11 Section 14.3.5. Otherwise, `enumtype` is set to `MPI_T_ENUM_NULL`. If the datatype is not
 ticket0. 12 `MPI_INT` or the argument `enumtype` is the null pointer, no `[enumeration]enumeration` type
 13 is returned.

ticket0. 14 Returning a description is optional. If an MPI implementation does not `[to]` return a
 15 description, the first character for `desc` must be set to the null character and `desc_len` must
 16 be set to one at the return from this function.

17 The parameter `bind` returns the type of the MPI object to which the variable must be
 18 bound or the value `MPI_T_BIND_NO_OBJECT` (see Section 14.3.2).

19 Upon return, the argument `readonly` is set to zero if the variable can be written or reset
 20 by the user. It is set to one if the variable can only be read.

21 Upon return, the argument `continuous` is set to zero if the variable can be started and
 22 stopped by the user, i.e., it is possible for the user to control if and when the value of a
 23 variable is updated. It is set to one if the variable is always active and cannot be controlled
 24 by the user.

25 Upon return, the argument `atomic` is set to zero if the variable cannot be read and
 26 reset atomically. Only variables for which the call sets `atomic` to one can be used in a call
 ticket383. 27 to `MPI_T_PVAR_READRESET`.

28 If a performance variable has an equivalent name and has the same class across con-
 29 nected processes, the following OUT parameters must be identical: `verbosity`, `varclass`,
 30 `datatype`, `enumtype`, `bind`, `readonly`, `continuous`, and `atomic`. The `desc` parameter must also
 ticket377. 31 be equivalent.

32
 33 `MPI_T_PVAR_GET_INDEX(name, var_class, pvar_index)`

| | | | |
|----|-----|-------------------------|---|
| 35 | IN | <code>name</code> | the name of the performance variable (string) |
| 36 | IN | <code>var_class</code> | the class of the performance variable (integer) |
| 37 | OUT | <code>pvar_index</code> | the index of the performance variable (integer) |

39
 40 `int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)`

41 `MPI_T_PVAR_GET_INDEX` is a function for retrieving the index of a performance
 42 variable given a known variable name and class. The `name` and `var_class` parameters are
 43 provided by the caller, and `pvar_index` is returned by the MPI implementation. The `name`
 44 parameter is string terminated with a null character.

45 This routine returns `MPI_SUCCESS` on success and returns `MPI_T_ERR_INVALID_NAME` if
 46 `name` does not match the name of any performance variable provided by the implementation
 47 at the time of the call.
 48


```

1 MPI_T_PVAR_HANDLE_ALLOC(session, pvar_index, obj_handle, handle, count)
2     IN      session          identifier of performance experiment session (handle)
3     IN      pvar_index      index of performance variable for which handle is to
4                               be allocated (integer)
5
6     IN      obj_handle      reference to a handle of the MPI object to which this
7                               variable is supposed to be bound (pointer)
8
9     OUT     handle          allocated handle (handle)
10    OUT     count           number of elements used to represent this variable (in-
11                               teger)

```

```

12
13 int MPI_T_pvar_handle_alloc(MPI_T_pvar_session session, int pvar_index,
14                             void *obj_handle, MPI_T_pvar_handle *handle, int *count)

```

This routine binds the performance variable specified by the argument `index` to an MPI object in the session identified by the parameter `session`. The object is passed in the argument `obj_handle` as an address to a local variable that stores the object's handle. The argument `obj_handle` is ignored if the `MPI_T_PVAR_GET_INFO` call for this performance variable returned `MPI_T_BIND_NO_OBJECT` in the argument `bind`. The handle allocated to reference the variable is returned in the argument `handle`. Upon successful return, `count` contains the number of elements (of the datatype returned by a previous `MPI_T_PVAR_GET_INFO` call) used to represent this variable.

Advice to users. The `count` can be different based on the MPI object to which the performance variable was bound. For example, variables bound to communicators could have a count that matches the size of the communicator.

It is not portable to pass references to predefined MPI object handles, such as `MPI_COMM_WORLD`, to this routine, since their implementation depends on the MPI library. Instead, such an object handle should be stored in a local variable and the address of this local variable should be passed into `MPI_T_PVAR_HANDLE_ALLOC`. (*End of advice to users.*)

The value of `index` should be in the range 0 to `num_pvar - 1`, where `num_pvar` is the number of available performance variables as determined from a prior call to `MPI_T_PVAR_GET_NUM`. The type of the MPI object it references must be consistent with the type returned in the `bind` argument in a prior call to `MPI_T_PVAR_GET_INFO`.

[In the case the `bind` argument equals `MPI_T_BIND_NO_OBJECT`, the argument `obj_handle` is ignored.]

For all routines in the rest of this section that take both `handle` and `session` as IN arguments, if the `handle` argument passed in is not associated with the `session` argument, `MPI_T_ERR_INVALID_HANDLE` is returned.

```

43
44 MPI_T_PVAR_HANDLE_FREE(session, handle)
45     IN      session          identifier of performance experiment session (handle)
46     INOUT   handle          handle to be freed (handle)
47
48

```

```
int MPI_T_pvar_handle_free(MPI_T_pvar_session session, MPI_T_pvar_handle
                          *handle)
```

When a handle is no longer needed, a user of the MPI tool information interface should call `MPI_T_PVAR_HANDLE_FREE` to free the handle in the session identified by the parameter `session` and the associated resources in the MPI implementation. On a successful return, MPI sets the handle to `MPI_T_PVAR_HANDLE_NULL`.

Starting and Stopping of Performance Variables

Performance variables that have the continuous flag set during the query operation are continuously operating once a handle has been allocated. Such variables may be queried at any time, but they cannot be started or stopped by the user. All other variables are in a stopped state after their handle has been allocated; their values are not updated until they have been started by the user.

```
MPI_T_PVAR_START(session, handle)
```

| | | |
|----|---------|---|
| IN | session | identifier of performance experiment session (handle) |
| IN | handle | handle of a performance variable (handle) |

```
int MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
```

This function starts the performance variable with the handle identified by the parameter `handle` in the session identified by the parameter `session`.

If the constant `MPI_T_PVAR_ALL_HANDLES` is passed in `handle`, the MPI implementation attempts to start all variables within the session identified by the parameter `session` for which handles have been allocated. In this case, the routine returns `MPI_SUCCESS` if all variables are started successfully (even if there are no non-continuous variables to be started), otherwise `MPI_T_ERR_PVAR_NO_STARTSTOP` is returned. Continuous variables and variables that are already started are ignored when `MPI_T_PVAR_ALL_HANDLES` is specified.

```
MPI_T_PVAR_STOP(session, handle)
```

| | | |
|----|---------|---|
| IN | session | identifier of performance experiment session (handle) |
| IN | handle | handle of a performance variable (handle) |

```
int MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
```

This function stops the performance variable with the handle identified by the parameter `handle` in the session identified by the parameter `session`.

If the constant `MPI_T_PVAR_ALL_HANDLES` is passed in `handle`, the MPI implementation attempts to stop all variables within the session identified by the parameter `session` for which handles have been allocated. In this case, the routine returns `MPI_SUCCESS` if all variables are stopped successfully (even if there are no non-continuous variables to be started), otherwise `MPI_T_ERR_PVAR_NO_STARTSTOP` is returned. Continuous variables and variables that are already stopped are ignored when `MPI_T_PVAR_ALL_HANDLES` is specified.

1 Performance Variable Access Functions

2
3
4 MPI_T_PVAR_READ(session, handle, buf)

5 IN session identifier of performance experiment session (handle)
6
7 IN handle handle of a performance variable (handle)
8 OUT buf initial address of storage location for variable value
9 (choice)

10
11 `int MPI_T_pvar_read(MPI_T_pvar_session session, MPI_T_pvar_handle handle,`
12 `void* buf)`
13

14 The MPI_T_PVAR_READ call queries the value of the performance variable with the
15 handle `handle` in the session identified by the parameter `session` and stores the result in the
16 buffer identified by the parameter `buf`. The user is responsible to ensure that the buffer
17 is of the appropriate size to hold the entire value of the performance variable (based on
18 the datatype and count returned by the corresponding previous calls to
19 MPI_T_PVAR_GET_INFO and MPI_T_PVAR_HANDLE_ALLOC, respectively).

20 The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the func-
21 tion MPI_T_PVAR_READ.

22
23
24 MPI_T_PVAR_WRITE(session, handle, buf)

25 IN session identifier of performance experiment session (handle)
26 IN handle handle of a performance variable (handle)
27 IN buf initial address of storage location for variable value
28 (choice)

29
30
31 `int MPI_T_pvar_write(MPI_T_pvar_session session, MPI_T_pvar_handle handle,`
32 `const void* buf)`
33

34 The MPI_T_PVAR_WRITE call attempts to write the value of the performance variable
35 with the handle identified by the parameter `handle` in the session identified by the parameter
36 `session`. The value to be written is passed in the buffer identified by the parameter `buf`. The
37 user must ensure that the buffer is of the appropriate size to hold the entire value of the per-
38 formance variable (based on the datatype and count returned by the corresponding previous
39 calls to MPI_T_PVAR_GET_INFO and MPI_T_PVAR_HANDLE_ALLOC, respectively).

40 If it is not possible to change the variable, the function returns
41 MPI_T_ERR_PVAR_NO_WRITE.

42 The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the func-
43 tion MPI_T_PVAR_WRITE.

MPI_T_PVAR_RESET(session, handle) 1

IN session identifier of performance experiment session (handle) 2

IN handle handle of a performance variable (handle) 3

int MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle) 4

The MPI_T_PVAR_RESET call sets the performance variable with the handle identified by the parameter handle to its starting value specified in Section 14.3.7. If it is not possible to change the variable, the function returns MPI_T_ERR_PVAR_NO_WRITE. 5

If the constant MPI_T_PVAR_ALL_HANDLES is passed in handle, the MPI implementation attempts to reset all variables within the session identified by the parameter session for which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all variables are reset successfully (even if there are no valid handles or all are read-only), otherwise MPI_T_ERR_PVAR_NO_WRITE is returned. Read-only variables are ignored when MPI_T_PVAR_ALL_HANDLES is specified. 6

MPI_T_PVAR_READRESET(session, handle, buf) 7

IN session identifier of performance experiment session (handle) 8

IN handle handle of a performance variable (handle) 9

OUT buf initial address of storage location for variable value (choice) 10

int MPI_T_pvar_readreset(MPI_T_pvar_session session, MPI_T_pvar_handle handle, void* buf) 11

This call atomically combines the functionality of MPI_T_PVAR_READ and MPI_T_PVAR_RESET with the same semantics as if these two calls were called separately. If atomic operations on this variable are not supported, this routine returns MPI_T_ERR_PVAR_NO_ATOMIC. 12

The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the function MPI_T_PVAR_READRESET. 13

Advice to implementors. Sampling-based tools rely on the ability to call the MPI tool information interface, in particular routines to start, stop, read, write and reset performance variables, from any program context, including asynchronous contexts such as signal handlers. MPI implementations should strive, if possible in their particular environment, to enable these usage scenarios for all or a subset of the routines mentioned above. If implementing only a subset, the read, write, and reset routines are typically the most critical for sampling based tools. An MPI implementation should clearly document any restrictions on the program contexts in which the MPI tool information interface can be used. Restrictions might include guaranteeing usage outside of all signals or outside a specific set of signals. Any restrictions could be documented, for example, through the description returned by MPI_T_PVAR_GET_INFO. (End of advice to implementors.) 14

Rationale. All routines to read, to write or to reset performance variables require the session argument. This requirement keeps the interface consistent and allows the use 15

ticket391.

1 of MPI_T_PVAR_ALL_HANDLES where appropriate. Further, this opens up additional
 2 performance optimizations for the implementation of handles. (*End of rationale.*)

3
 4 Example: Tool to Detect Receives with Long Unexpected Message Queues

5
 6 **Example 14.6**

7 The following example shows a sample tool to identify receive operations that occur
 8 during times with long message queues. This examples assumes that the MPI implementa-
 9 tion exports a variable with the name “MPI_T_UMQ_LENGTH” to represent the current length
 10 of the unexpected message queue. The tool is implemented as a PMPI tool using the MPI
 11 profiling interface.

12 The tool consists of three parts: (1) the initialization (by intercepting the call to
 13 MPI_INIT), (2) the test for long unexpected message queues (by intercepting calls to
 14 MPI_RECV), and (3) the clean-up phase (by intercepting the call to MPI_FINALIZE). To
 15 capture all receives, the example would have to be extended to have similar wrappers for
 16 all receive operations.

17
 18 Part 1— Initialization: During initialization, the tool searches for the variable and, once
 19 the right index is found, allocates a session and a handle for the variable with the found
 20 index, and starts the performance variable.

```

21
22 #include <stdio.h>
23 #include <stdlib.h>
24 #include <string.h>
25 #include <assert.h>
26 #include <mpi.h>
27
28 /* Global variables for the tool */
29 static MPI_T_pvar_session session;
30 static MPI_T_pvar_handle handle;
31
32 int MPI_Init(int *argc, char ***argv ) {
33     int err, num, i, index, namelen, verbosity;
34     int var_class, bind, threadsup;
35     int readonly, continuous, atomic, count;
36     char name[18];
37     MPI_Comm comm;
38     MPI_Datatype datatype;
39     MPI_T_enum enumtype;
40
41     err=PMPI_Init(argc,argv);
42     if (err!=MPI_SUCCESS) return err;
43
44     err=PMPI_T_init_thread(MPI_THREAD_SINGLE,&threadsup);
45     if (err!=MPI_SUCCESS) return err;
46
47     err=PMPI_T_pvar_get_num(&num);
48

```



```

if (err!=MPI_SUCCESS) return err; 1
index=-1; 2
i=0; 3
while ((i<num) && (index<0) && (err==MPI_SUCCESS)) { 4
/* Pass a buffer that is at least one character longer than */ 5
/* the name of the variable being searched for to avoid */ 6
/* finding variables that have a name that has a prefix */ 7
/* equal to the name of the variable being searched. */ 8
namelen=18; 9
err=PMPI_T_pvar_get_info(i, name, &namelen, &verbosity, 10
&var_class, &datatype, &enumtype, NULL, NULL, &bind, 11
&readonly, &continuous, &atomic); 12
if (strcmp(name,"MPI_T_UMQ_LENGTH")==0) index=i; 13
i++; } 14
if (err!=MPI_SUCCESS) return err; 15
16

/* this could be handled in a more flexible way for a generic tool */ 17
assert(index>=0); 18
assert(var_class==MPI_T_PVAR_CLASS_LEVEL); 19
assert(datatype==MPI_INT); 20
assert(bind==MPI_T_BIND_MPI_COMM); 21
22

/* Create a session */ 23
err=PMPI_T_pvar_session_create(&session); 24
if (err!=MPI_SUCCESS) return err; 25
26

/* Get a handle and bind to MPI_COMM_WORLD */ 27
comm=MPI_COMM_WORLD; 28
err=PMPI_T_pvar_handle_alloc(session, index, &comm, &handle, &count); 29
if (err!=MPI_SUCCESS) return err; 30
31

/* this could be handled in a more flexible way for a generic tool */ 32
assert(count==1); 33
34

/* Start variable */ 35
err=PMPI_T_pvar_start(session, handle); 36
if (err!=MPI_SUCCESS) return err; 37
38

return MPI_SUCCESS; 39
} 40
41

Part 2 — Testing the Queue Lengths During Receives: During every receive operation, the 42
tool reads the unexpected queue length through the matching performance variable and 43
compares it against a predefined threshold. 44
45

#define THRESHOLD 5 46
47

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, 48

```

```

1             MPI_Comm comm, MPI_Status *status)
2     {
3     int value, err;
4
5     if (comm==MPI_COMM_WORLD) {
6     err=PMPI_T_pvar_read(session, handle, &value);
7     if ((err==MPI_SUCCESS) && (value>THRESHOLD))
8     {
9             /* tool identified receive called with long UMQ */
10    /* execute tool functionality, */
11    /* e.g., gather and print call stack */
12    }
13    }
14
15    return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
16    }

```

18 Part 3 — Termination: In the wrapper for MPI_FINALIZE, the MPI tool information inter-
19 face is finalized.

```

20
21 int MPI_Finalize()
22 {
23 int err;
24 err=PMPI_T_pvar_handle_free(session, &handle);
25 err=PMPI_T_pvar_session_free(&session);
26 err=PMPI_T_finalize();
27 return PMPI_Finalize();
28 }

```

30 14.3.8 Variable Categorization

31 MPI implementations can optionally group performance and control variables into categories
32 to express logical relationships between various variables. For example, an MPI implemen-
33 tation could group all control and performance variables that refer to message transfers in
34 the MPI implementation and thereby distinguish them from variables that refer to local
35 resources such as memory allocations or other interactions with the operating system.

36 Categories can also contain other categories to form a hierarchical grouping. Categories
37 can never include themselves, either directly or transitively within other included categories.
38 Expanding on the example above, this allows MPI to refine the grouping of variables referring
39 to message transfers into variables to control and to monitor message queues, message
40 matching activities and communication protocols. Each of these groups of variables would
41 be represented by a separate category and these categories would then be listed in a single
42 category representing variables for message transfers.

43 The category information may be queried in a fashion similar to the mechanism for
44 querying variable information. The MPI implementation exports a set of N categories via
45 the MPI tool information interface. If $N = 0$, then the MPI implementation does not export
46 any categories, otherwise the provided categories are indexed from 0 to $N - 1$. This index
47

48

number is used in subsequent calls to functions of the MPI tool information interface to identify the individual categories.

An MPI implementation is permitted to increase the number of categories during the execution of an MPI program when new categories become available through dynamic loading. However, MPI implementations are not allowed to change the index of a category or delete it once it has been added to the set.

Similarly, MPI implementations are allowed to add variables to categories, but they are not allowed to remove variables from categories or change the order in which they are returned.

The following function can be used to query the number of **[control variables]categories**, *N*.

```
MPI_T_CATEGORY_GET_NUM(num_cat)
```

| | | |
|-----|---------|--|
| OUT | num_cat | current number of categories (integer) |
|-----|---------|--|

```
int MPI_T_category_get_num(int *num_cat)
```

Individual category information can then be queried by calling the following function:

```
MPI_T_CATEGORY_GET_INFO(cat_index, name, name_len, desc, desc_len, num_cvars,
                        num_pvars, num_categories)
```

| | | |
|-------|----------------|---|
| IN | cat_index | index of the category to be queried (integer) |
| OUT | name | buffer to return the string containing the name of the category (string) |
| INOUT | name_len | length of the string and/or buffer for name (integer) |
| OUT | desc | buffer to return the string containing the description of the category (string) |
| INOUT | desc_len | length of the string and/or buffer for desc (integer) |
| OUT | num_cvars | number of control variables in the category (integer) |
| OUT | num_pvars | number of performance variables in the category (integer) |
| OUT | num_categories | number of categories contained in the category (integer) |

```
int MPI_T_category_get_info(int cat_index, char *name, int *name_len,
                          char *desc, int *desc_len, int *num_cvars, int *num_pvars,
                          int *num_categories)
```

The arguments *name* and *name_len* are used to return the name of the category as described in Section 14.3.3.

The routine is required to return a name of at least length one. This name must be unique with respect to all other names for categories used by the MPI implementation.

If any OUT parameter to MPI_T_CATEGORY_GET_INFO is a NULL pointer, the implementation will ignore the parameter and not return a value for the parameter.

1 The arguments `desc` and `desc_len` are used to return the description of the category as
 2 described in Section 14.3.3.

3 Returning a description is optional. If an MPI implementation decides not to return a
 4 description, the first character for `desc` must be set to the null character and `desc_len` must
 5 be set to one at the return of this call.

6 The function returns the number of control variables, performance variables and other
 7 categories contained in the queried category in the arguments `num_cvars`, `num_pvars`, and
 ticket383. 8 `num_categories`, respectively.

9 If the name of a category is equivalent across connected processes, then `desc` must also
 ticket377. 10 be equivalent.

11
 12 `MPI_T_CATEGORY_GET_INDEX(name, cat_index)`

13
 14 IN name the name of the category (string)
 15 OUT cat_index the index of the category (integer)
 16

17 `int MPI_T_category_get_index(const char *name, int *cat_index)`
 18

19 `MPI_T_CATEGORY_GET_INDEX` is a function for retrieving the index of a category
 20 given a known category name. The `name` parameter is provided by the caller, and `cat_index`
 21 is returned by the MPI implementation. The `name` parameter is a string terminated with a
 22 null character.

23 This routine returns `MPI_SUCCESS` on success and returns `MPI_T_ERR_INVALID_NAME`
 24 if `name` does not match the name of any category provided by the implementation at the
 25 time of the call.

26
 27 *Rationale.* This routine is provided to enable fast retrieval of a category index
 28 by a tool, assuming it knows the name of the category for which it is looking. The
 29 number of categories exposed by the implementation can change over time, so it is not
 30 possible for the tool to simply iterate over the list of categories once at initialization.
 31 Although using MPI implementation specific category names is not portable across
 32 MPI implementations, tool developers may choose to take this route for lower overhead
 33 at runtime because the tool will not have to iterate over the entire set of categories
 34 to find a specific one. (*End of rationale.*)
 35
 36

37 `MPI_T_CATEGORY_GET_CVARS(cat_index, len, indices)`

38 IN cat_index index of the category to be queried, in the range $[0, N-1]$ (integer)
 39
 40 IN len the length of the indices array (integer)
 41 OUT indices an integer array of size `len`, indicating control variable
 42 indices (array of integers)
 43
 44

45 `int MPI_T_category_get_cvars(int cat_index, int len, int indices[])`
 46

47 `MPI_T_CATEGORY_GET_CVARS` can be used to query which control variables are
 48 contained in a particular category. A category contains zero or more control variables.

```

MPI_T_CATEGORY_GET_PVARS(cat_index,len,indices) 1
  IN      cat_index      index of the category to be queried, in the range [0, N- 2
                        1] (integer) 3
  IN      len            the length of the indices array (integer) 4
  OUT     indices        an integer array of size len, indicating performance 5
                        variable indices (array of integers) 6

```

```

int MPI_T_category_get_pvars(int cat_index, int len, int indices[]) 7

```

MPI_T_CATEGORY_GET_PVARS can be used to query which performance variables are contained in a particular category. A category contains zero or more performance variables.

```

MPI_T_CATEGORY_GET_CATEGORIES(cat_index,len,indices) 8
  IN      cat_index      index of the category to be queried, in the range [0, N- 9
                        1] (integer) 10
  IN      len            the length of the indices array (integer) 11
  OUT     indices        an integer array of size len, indicating category indices 12
                        (array of integers) 13

```

```

int MPI_T_category_get_categories(int cat_index, int len, int indices[]) 14

```

MPI_T_CATEGORY_GET_CATEGORIES can be used to query which other categories are contained in a particular category. A category contains zero or more other categories.

As mentioned above, MPI implementations can grow the number of categories as well as the number of variables or other categories within a category. In order to allow users of the MPI tool information interface to check quickly whether new categories have been added or new variables or categories have been added to a category, MPI maintains a virtual timestamp. This timestamp is monotonically increasing during the execution and is returned by the following function:

```

MPI_T_CATEGORY_CHANGED(stamp) 15
  OUT     stamp          a virtual time stamp to indicate the last change to the 16
                        categories (integer) 17

```

```

int MPI_T_category_changed(int *stamp) 18

```

If two subsequent calls to this routine return the same timestamp, it is guaranteed that the category information has not changed between the two calls. If the timestamp retrieved from the second call is higher, then some categories have been added or expanded.

Advice to users. The timestamp value is purely virtual and only intended to check for changes in the category information. It should not be used for any other purpose. (*End of advice to users.*)

1 The index values returned in indices by `MPI_T_CATEGORY_GET_CVARS`,
 2 `MPI_T_CATEGORY_GET_PVARS` and `MPI_T_CATEGORY_GET_CATEGORIES` can be used
 3 as input to `MPI_T_CVAR_GET_INFO`, `MPI_T_PVAR_GET_INFO` and
 4 `MPI_T_CATEGORY_GET_INFO`, respectively.

5 The user is responsible for allocating the arrays passed into the functions
 6 `MPI_T_CATEGORY_GET_CVARS`, `MPI_T_CATEGORY_GET_PVARS` and
 7 `MPI_T_CATEGORY_GET_CATEGORIES`. Starting from array index 0, each function writes
 8 up to `len` elements into the array. If the category contains more than `len` elements, the
 9 function returns an arbitrary subset of size `len`. Otherwise, the entire set of elements is
 10 returned in the beginning entries of the array, and any remaining array entries are not
 11 modified.

13 14.3.9 Return Codes for the MPI Tool Information Interface

14 All functions defined as part of the MPI tool information interface return an integer error
 15 code (see Table 14.5) to indicate whether the function was completed successfully or was
 16 aborted. In the latter case the error code indicates the reason for not completing the routine.
 17 Such errors neither impact the execution of the MPI process nor invoke MPI error handlers.
 18 The MPI process continues executing regardless of the return code from the call. The MPI
 19 implementation is not required to check all user-provided parameters; if a user passes invalid
 20 parameter values to any routine the behavior of the implementation is undefined.

21 All error codes with the prefix `MPI_T_` must be unique values and cannot overlap with
 22 any other error codes or error classes returned by the MPI implementation. Further, they
 23 shall be treated as MPI error classes as defined in Section 8.4 on page 347 and follow the
 24 same rules and restrictions. In particular, they must satisfy:

$$25 \quad 0 = \text{MPI_SUCCESS} < \text{MPI_T_ERR_...} \leq \text{MPI_ERR_LASTCODE}.$$

26
 27
 28 *Rationale.* All MPI tool information interface functions must return error classes,
 29 because applications cannot portably call `MPI_ERROR_CLASS` before
 30 `MPI_INIT` or `MPI_INIT_THREAD` to map an arbitrary error code to an error class.
 31 (*End of rationale.*)
 32

34 14.3.10 Profiling Interface

35
 36 All requirements for the profiling interfaces, as described in Section 14.2, also apply to
 37 the MPI tool information interface. All rules, guidelines, and recommendations from Sec-
 38 tion 14.2 apply equally to calls defined as part of the MPI tool information interface.
 39
 40
 41
 42
 43
 44
 45
 46
 47
 48

| Return Code | Description | |
|---|--|----|
| | | 1 |
| | | 2 |
| Return Codes for All Functions in the MPI Tool Information Interface | | 3 |
| MPI_SUCCESS | Call completed successfully | 4 |
| [ticket400.] MPI_T_ERR_INVALID | [ticket400.]Invalid use of the interface or bad parameter values | 5 |
| MPI_T_ERR_MEMORY | Out of memory | 7 |
| MPI_T_ERR_NOT_INITIALIZED | Interface not initialized | 8 |
| MPI_T_ERR_CANNOT_INIT | Interface not in the state to be initialized | 9 |
| Return Codes for Datatype Functions: MPI_T_ENUM_* | | 10 |
| MPI_T_ERR_INVALID_INDEX | The enumeration index is invalid | 11 |
| | [ticket406.][or has been deleted.] | 12 |
| MPI_T_ERR_INVALID_ITEM | The item index queried is out of range | 13 |
| | (for MPI_T_ENUM_GET_ITEM only) | 14 |
| Return Codes for variable and category query functions: [ticket377.][MPI_T_*_GET_INFO]MPI_T_*_GET_* | | 16 |
| MPI_T_ERR_INVALID_INDEX | The variable or category index is invalid | 17 |
| [ticket377.]MPI_T_ERR_INVALID_NAME | [ticket377.]The variable or category name is invalid | 18 |
| Return Codes for Handle Functions: MPI_T_*_{ALLOC FREE} | | 19 |
| MPI_T_ERR_INVALID_INDEX | The variable index is invalid [ticket406.][or has been deleted] | 20 |
| MPI_T_ERR_INVALID_HANDLE | The handle is invalid | 21 |
| MPI_T_ERR_OUT_OF_HANDLES | No more handles available | 22 |
| Return Codes for Session Functions: MPI_T_PVAR_SESSION_* | | 23 |
| MPI_T_ERR_OUT_OF_SESSIONS | No more sessions available | 24 |
| MPI_T_ERR_INVALID_SESSION | Session argument is not a valid session | 25 |
| Return Codes for Control Variable Access Functions: | | 26 |
| MPI_T_CVAR_READ, WRITE | | 27 |
| MPI_T_ERR_CVAR_SET_NOT_NOW | Variable cannot be set at this moment | 28 |
| MPI_T_ERR_CVAR_SET_NEVER | Variable cannot be set until end of execution | 29 |
| MPI_T_ERR_INVALID_HANDLE | The handle is invalid | 30 |
| Return Codes for Performance Variable Access and Control: | | 31 |
| MPI_T_PVAR_{START STOP READ WRITE RESET READREST} | | 32 |
| MPI_T_ERR_INVALID_HANDLE | The handle is invalid | 33 |
| MPI_T_ERR_INVALID_SESSION | Session argument is not a valid session | 34 |
| MPI_T_ERR_PVAR_NO_STARTSTOP | Variable cannot be started or stopped | 35 |
| | (for MPI_T_PVAR_START and | 36 |
| | MPI_T_PVAR_STOP) | 37 |
| MPI_T_ERR_PVAR_NO_WRITE | Variable cannot be written or reset | 38 |
| | (for MPI_T_PVAR_WRITE and | 39 |
| | MPI_T_PVAR_RESET) | 40 |
| MPI_T_ERR_PVAR_NO_ATOMIC | Variable cannot be read and written atomically | 41 |
| | (for MPI_T_PVAR_READRESET) | 42 |
| Return Codes for Category Functions: MPI_T_CATEGORY_* | | 43 |
| MPI_T_ERR_INVALID_INDEX | The category index is invalid | 44 |

Table 14.5: Return codes used in functions of the MPI tool information interface

Index

- CONST:MPI_CHAR, [11](#), [12](#)
- CONST:MPI_COMM_WORLD, [18](#), [26](#)
- CONST:MPI_COUNT, [11](#)
- CONST:MPI_DOUBLE, [11](#), [20–22](#)
- CONST:MPI_ERR_LASTCODE, [36](#)
- CONST:MPI_INT, [11](#), [12](#), [15](#), [20](#), [24](#)
- CONST:MPI_SUCCESS, [7](#), [16](#), [24](#), [27](#), [29](#), [34](#), [36](#), [37](#)
- CONST:MPI_T_, [36](#)
- CONST:MPI_T_BIND_MPI_COMM, [9](#)
- CONST:MPI_T_BIND_MPI_DATATYPE, [9](#)
- CONST:MPI_T_BIND_MPI_ERRHANDLER, [9](#)
- CONST:MPI_T_BIND_MPI_FILE, [9](#)
- CONST:MPI_T_BIND_MPI_GROUP, [9](#)
- CONST:MPI_T_BIND_MPI_INFO, [9](#)
- CONST:MPI_T_BIND_MPI_MESSAGE, [9](#)
- CONST:MPI_T_BIND_MPI_OP, [9](#)
- CONST:MPI_T_BIND_MPI_REQUEST, [9](#)
- CONST:MPI_T_BIND_MPI_WIN, [9](#)
- CONST:MPI_T_BIND_NO_OBJECT, [9](#), [15](#), [18](#), [24](#), [26](#)
- CONST:MPI_T_cvar_handle, [17](#), [17–19](#)
- CONST:MPI_T_CVAR_HANDLE_NULL, [18](#)
- CONST:MPI_T_CVAR_READ, WRITE, [37](#)
- CONST:MPI_T_enum, [12](#), [12–14](#), [23](#)
- CONST:MPI_T_ENUM_NULL, [15](#), [24](#)
- CONST:MPI_T_ERR_..., [36](#)
- CONST:MPI_T_ERR_CANNOT_INIT, [37](#)
- CONST:MPI_T_ERR_CVAR_SET_NEVER, [19](#), [37](#)
- CONST:MPI_T_ERR_CVAR_SET_NOT_NOW, [19](#), [37](#)
- CONST:MPI_T_ERR_INVALID, [37](#)
- CONST:MPI_T_ERR_INVALID_HANDLE, [26](#), [37](#)
- CONST:MPI_T_ERR_INVALID_INDEX, [37](#)
- CONST:MPI_T_ERR_INVALID_ITEM, [37](#)
- CONST:MPI_T_ERR_INVALID_NAME, [16](#), [24](#), [34](#), [37](#)
- CONST:MPI_T_ERR_INVALID_SESSION, [37](#)
- CONST:MPI_T_ERR_MEMORY, [37](#)
- CONST:MPI_T_ERR_NOT_INITIALIZED, [37](#)
- CONST:MPI_T_ERR_OUT_OF_HANDLES, [37](#)
- CONST:MPI_T_ERR_OUT_OF_SESSIONS, [37](#)
- CONST:MPI_T_ERR_PVAR_NO_ATOMIC, [29](#), [37](#)
- CONST:MPI_T_ERR_PVAR_NO_STARTSTOP, [27](#), [37](#)
- CONST:MPI_T_ERR_PVAR_NO_WRITE, [28](#), [29](#), [37](#)
- CONST:MPI_T_PVAR_ALL_HANDLES, [27–30](#)
- CONST:MPI_T_PVAR_CLASS_AGGREGATE, [21](#), [22](#)
- CONST:MPI_T_PVAR_CLASS_COUNTER, [21](#)
- CONST:MPI_T_PVAR_CLASS_GENERIC, [22](#)
- CONST:MPI_T_PVAR_CLASS_HIGHWATERMARK, [21](#)
- CONST:MPI_T_PVAR_CLASS_LEVEL, [20](#)
- CONST:MPI_T_PVAR_CLASS_LOWWATERMARK, [21](#)
- CONST:MPI_T_PVAR_CLASS_PERCENTAGE, [21](#)
- CONST:MPI_T_PVAR_CLASS_SIZE, [21](#)
- CONST:MPI_T_PVAR_CLASS_STATE, [20](#)
- CONST:MPI_T_PVAR_CLASS_TIMER, [22](#)
- CONST:MPI_T_pvar_handle, [25](#), [25–29](#)
- CONST:MPI_T_PVAR_HANDLE_NULL, [27](#)
- CONST:MPI_T_pvar_session, [25](#), [25–29](#)
- CONST:MPI_T_PVAR_SESSION_NULL, [25](#)
- CONST:MPI_T_SCOPE_ALL, [15](#)
- CONST:MPI_T_SCOPE_ALL_EQ, [15](#), [19](#)
- CONST:MPI_T_SCOPE_CONSTANT, [15](#)

| | | |
|--|---|----|
| CONST:MPI_T_SCOPE_GROUP, 15 | MPI_T_CATEGORY_CHANGED(stamp), 35 | 1 |
| CONST:MPI_T_SCOPE_GROUP_EQ, 15 , 19 | MPI_T_CATEGORY_GET_CATEGORIES, | 2 |
| CONST:MPI_T_SCOPE_LOCAL, 15 | 35–37 | 3 |
| CONST:MPI_T_SCOPE_READONLY, 15 | MPI_T_CATEGORY_GET_CATEGORIES(cat_index, len, indices), | 4 |
| CONST:MPI_T_VERBOSITY_MPIDEV_ALL, | 35 | 5 |
| 8 | MPI_T_CATEGORY_GET_CVARS, 34 , 36 , | 6 |
| CONST:MPI_T_VERBOSITY_MPIDEV_BASIC, | 37 | 7 |
| 8 | MPI_T_CATEGORY_GET_CVARS(cat_index, | 8 |
| CONST:MPI_T_VERBOSITY_MPIDEV_DETAIL, | len, indices), 34 | 9 |
| 8 | MPI_T_CATEGORY_GET_INDEX, 34 | 10 |
| CONST:MPI_T_VERBOSITY_TUNER_ALL, | MPI_T_CATEGORY_GET_INDEX(name, cat_index), | 11 |
| 8 | 34 | 12 |
| CONST:MPI_T_VERBOSITY_TUNER_BASIC, | MPI_T_CATEGORY_GET_INFO, 33 , 36 , 37 | 13 |
| 8 | MPI_T_CATEGORY_GET_INFO(cat_index, | 14 |
| CONST:MPI_T_VERBOSITY_TUNER_DETAIL, | name, name_len, desc, desc_len, num_cvars, num_pvars, | 15 |
| 8 | num_categories), 33 | 16 |
| CONST:MPI_T_VERBOSITY_USER_ALL, | MPI_T_CATEGORY_GET_NUM(num_cat), | 17 |
| 8 | 33 | 18 |
| CONST:MPI_T_VERBOSITY_USER_BASIC, | MPI_T_CATEGORY_GET_PVARS, 35–37 | 19 |
| 8 | MPI_T_CATEGORY_GET_PVARS(cat_index, len, indices), | 20 |
| CONST:MPI_T_VERBOSITY_USER_DETAIL, | 35 | 21 |
| 8 | MPI_T_CVAR_GET_INDEX, 16 | 22 |
| CONST:MPI_UNSIGNED, 11 , 20–22 | MPI_T_CVAR_GET_INDEX(name, cvar_index), | 23 |
| CONST:MPI_UNSIGNED_LONG, 11 , 20–22 | 16 | 24 |
| CONST:MPI_UNSIGNED_LONG_LONG, 11 , 20–22 | MPI_T_CVAR_GET_INFO, 12 , 14 , 15 , 17– | 25 |
| | 19 , 36 , 37 | 26 |
| CONST:NULL, 14 , 23 , 33 | MPI_T_CVAR_GET_INFO(cvar_index, name, | 27 |
| | name_len, verbosity, datatype, enum- | 28 |
| EXAMPLES:Basic tool using performance vari- | type, desc, desc_len, bind, scope), | 29 |
| ables in the MPI tool information in- | 14 | 30 |
| terface, 30 | MPI_T_CVAR_GET_NUM, 18 | 31 |
| EXAMPLES:Profiling interface, 4 | MPI_T_CVAR_GET_NUM(num_cvar), 14 | 32 |
| EXAMPLES:Reading the value of a control | MPI_T_CVAR_HANDLE_ALLOC, 12 , 18 , | 33 |
| variable in the MPI tool information | 19 , 37 | 34 |
| interface, 19 | MPI_T_CVAR_HANDLE_ALLOC(cvar_index, | 35 |
| EXAMPLES:Using MPI_T_CVAR_GET_INFO | obj_handle, handle, count), 17 | 36 |
| to list all names of control variables., | MPI_T_CVAR_HANDLE_FREE, 18 , 37 | 37 |
| 16 | MPI_T_CVAR_HANDLE_FREE(handle), 18 | 38 |
| | MPI_T_CVAR_READ(handle, buf), 18 | 39 |
| MPI_ABORT, 11 | MPI_T_CVAR_WRITE(handle, buf), 19 | 40 |
| MPI_ERROR_CLASS, 36 | MPI_T_ENUM_GET_INFO, 12 , 37 | 41 |
| MPI_FINALIZE, 7 , 17 , 30 , 32 | MPI_T_ENUM_GET_INFO(enumtype, num, | 42 |
| MPI_INIT, 3 , 7 , 10–12 , 17 , 30 , 36 | name, name_len), 12 | 43 |
| MPI_INIT_THREAD, 10 , 36 | MPI_T_ENUM_GET_ITEM, 12 , 37 | 44 |
| MPI_PCONTROL, 2 , 3 | MPI_T_ENUM_GET_ITEM(enumtype, in- | 45 |
| MPI_PCONTROL(level, ...), 3 | dex, value, name, name_len), 13 | 46 |
| MPI_RECV, 30 | MPI_T_FINALIZE, 11 | 47 |
| MPI_SEND, 4 | MPI_T_FINALIZE(), 10 | 48 |

1 MPI_T_INIT_THREAD, [10](#), [11](#)
2 MPI_T_INIT_THREAD(required, provided),
3 [10](#)
4 MPI_T_PVAR_GET_INDEX, [24](#)
5 MPI_T_PVAR_GET_INDEX(name, var_class,
6 pvar_index), [24](#)
7 MPI_T_PVAR_GET_INFO, [12](#), [22](#), [23](#), [26](#),
8 [28](#), [29](#), [36](#), [37](#)
9 MPI_T_PVAR_GET_INFO(pvar_index, name,
10 name_len, verbosity, varclass, datatype,enumtype,
11 desc, desc_len, bind, readonly, con-
12 tinuous, atomic), [23](#)
13 MPI_T_PVAR_GET_NUM, [26](#)
14 MPI_T_PVAR_GET_NUM(num_pvar), [22](#)
15 MPI_T_PVAR_HANDLE_ALLOC, [12](#), [26](#),
16 [28](#), [37](#)
17 MPI_T_PVAR_HANDLE_ALLOC(session, pvar_index,
18 obj_handle, handle, count), [26](#)
19 MPI_T_PVAR_HANDLE_FREE, [27](#), [37](#)
20 MPI_T_PVAR_HANDLE_FREE(session, han-
21 dle), [26](#)
22 MPI_T_PVAR_READ, [28](#), [29](#), [37](#)
23 MPI_T_PVAR_READ(session, handle, buf),
24 [28](#)
25 MPI_T_PVAR_READRESET, [24](#), [29](#), [37](#)
26 MPI_T_PVAR_READRESET(session, han-
27 dle, buf), [29](#)
28 MPI_T_PVAR_RESET, [29](#), [37](#)
29 MPI_T_PVAR_RESET(session, handle), [29](#)
30 MPI_T_PVAR_SESSION_CREATE, [37](#)
31 MPI_T_PVAR_SESSION_CREATE(session),
32 [25](#)
33 MPI_T_PVAR_SESSION_FREE, [37](#)
34 MPI_T_PVAR_SESSION_FREE(session), [25](#)
35 MPI_T_PVAR_START, [37](#)
36 MPI_T_PVAR_START(session, handle), [27](#)
37 MPI_T_PVAR_STOP, [37](#)
38 MPI_T_PVAR_STOP(session, handle), [27](#)
39 MPI_T_PVAR_WRITE, [28](#), [37](#)
40 MPI_T_PVAR_WRITE(session,handle, buf),
41 [28](#)
42 MPI_TYPE_SIZE, [4](#)
43 MPI_WTIME, [4](#), [22](#)
44
45 PMPI_, [1](#)
46
47
48