MPI: A Message-Passing Interface Standard

Version 3.0

$\top$ (Fin2)

$\bot$ (Fin2)

Message Passing Interface Forum

Draft March 14th, 2011

Contents

List of Figures

List of Tables

Chapter 1

Tool Interfaces

1.1 Introduction

This chapter discusses a set of interfaces that allows debuggers, performance analyzers, and other tools to extract information about the operation of MPI processes. Specifically, this chapter defines both the PMPI profiling interface (Section 1.2) for transparently intercepting and inspecting any MPI call, and the MPI_T tool information interface (Section 1.3) for querying 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.

1.2 Profiling Interface

1.2.1 Requirements

To meet [the]the requirements for the MPI profiling interface, an implementation of the MPI functions *must* ⊤ (Fin2) ⊥ (Fin2)

1. provide a mechanism through which all of the MPI defined [functions]functions, except those allowed as macros (See Section **??**[)]), 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. The profiling interface in C++ is described in Section **??**. 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. ⊤ (Fin2) ⊥ (Fin2) ⊤ (Fin2) ⊥ (Fin2)

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 [economise]economize by implementing it only for the lowest level routines. ⊤ (Fin2) ⊥ (Fin2)

4. where the implementation of different language bindings is done through a layered approach ([e.g.]e.g., the Fortran binding is a set of "wrapper" functions that call the ⊤ (Fin2) ⊥ (Fin2)

C implementation), ensure that these wrapper functions are separable from the rest of the library.

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

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

1.2.2 Discussion

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

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

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

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

While the initial impetus for the development of this interface arose from the desire to permit the implementation of profiling tools, it is clear that an interface like that specified may also prove useful for other purposes, such as "internetworking" multiple MPI implementations. Since all that is defined is an interface, there is no objection to its being used wherever it is useful.

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

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

1.2.3 Logic of the Design

Provided that an MPI implementation meets the requirements above, it is possible for the implementor of the profiling system to intercept all of the MPI calls that are made by the 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.

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 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]calculation.
- Adding user events to a trace file.

These requirements are met by use of the MPI_PCONTROL.

MPI_PCONTROL(level, ...)

IN	level	Profiling level

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

```
MPI_PCONTROL(LEVEL)
    INTEGER LEVEL
```

{`void MPI::Pcontrol(const int level, ...)` *(binding deprecated, see Section ??)* }

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. (This may be a no-op in some profilers).]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 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 [allows them to modify their source code to obtain]supports the collection of more detailed profiling information[, but still be able to link exactly the]with source [same code]code that can still link against the standard MPI library.

1.2.4 Profiler Implementation Example

[Suppose that the profiler wishes to]A profiler can accumulate the total amount of data sent by the [MPI_SEND]MPI_SEND function, along with the total elapsed time spent in the [function. This could trivially be achieved thus]function, as follows:

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

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

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

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

   return result;
}
```

1.2.5 MPI Library Implementation Example

[On a Unix system, in which the MPI library is implemented in C, then]If the MPI library is implemented in C on a Unix system, then there [there are various possible options, of which two of the most obvious]are various options, including the two presented here, for supporting [are presented here. Which is better depends on whether the linker and]the name-shift requirement. The choice between these two options [compiler support weak symbols.]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.]e.g., Solaris 2.x, other system V.4 machines), then only a single library is required through the use of `#pragma weak` thus

```
#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 pre-processor thus

```
#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. ⊤ (Fin2) ⊥ (Fin2)

1.2.6 Complications

Multiple Counting

Since parts of the MPI library may themselves be implemented using more basic MPI functions ([e.g.]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.]e.g., it might allow one to answer the question "How much time is spent in the point to point routines when they're called from collective functions?"), we have decided not to enforce any restrictions on the author of the MPI library that would overcome this. Therefore the author of the profiling library should be aware of this problem, and guard against it herself. In a single threaded world this is easily achieved through use of a static variable in the profiling code that remembers if you are already inside a profiling routine. It becomes more complex in a multi-threaded environment (as does the meaning of the times recorded[!])[]. ⊤ (Fin2) ⊥ (Fin2) ⊤ (Fin2) ⊥ (Fin2) ⊤ (Fin2) ⊥ (Fin2) ⊤ (Fin2) ⊥ (Fin2)

Linker Oddities

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

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

To overcome this we must ensure that the Fortran wrapper functions are included in the profiling version of the library. We ensure that this is possible by requiring that these be separable from the rest of the base MPI library. This allows them to be `ar`ed out of the base library and into the profiling one.

1.2.7 Multiple Levels of Interception

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

- assuming a particular implementation language[.],
- imposing a run time cost even when no profiling was taking place.

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

[Note, however, that it is possible to use the scheme above to implement a multi-level system, since the function called by the user may call many different profiling functions before calling the underlying MPI function.]

[Unfortunately such an implementation may require more cooperation between the different profiling libraries than is required for the single level implementation detailed above.]Note, however, that it is possible to use the scheme above to implement a multi-level 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 [**?**].

1.3 MPI_T Tool Information Interface

To optimize MPI applications or their runtime behavior, it is often advantageous to understand the performance switches an MPI implementation offers to the user as well as to monitor properties and timing information from within the MPI implementation.

The MPI_T interface described in this section provides a mechanism for the MPI implementation to expose a set of variables, each of which represent a particular property, setting, or performance measurement from within the MPI implementation. The MPI_T interface provides the necessary routines to find all variables that exist in the particular MPI implementation, query their properties, retrieve descriptions about their meaning and access and, if appropriate, alter their values.

The interface is split into two parts: the first part provides information about control variables used by the MPI implementation to fine tune its configuration. The second part provides access to performance variables that can provide insight into internal performance information of the underlying MPI implementation.

To avoid restrictions on the MPI implementation, the MPI_T interface allows the implementation to specify which control and performance variables exist. Additionally, the MPI_T interface can obtain metadata about each available variable, such as its datatype and size, a textual description, etc.

To avoid conflicts between the standard MPI functionality and the tools-oriented functionality introduced with MPI_T , the MPI_T interface is contained in its own name space. All identifiers covered by this interface carry the prefix MPI_T and can be used independently from the MPI functionality. This includes initialization and finalization of MPI_T , which is provided through a separate set of routines. Consequently, MPI_T routines can be called before MPI_INIT and after MPI_FINALIZE.

On success, all MPI_T routines return MPI_T_SUCCESS, otherwise they return an appropriate error code. Details on error codes can be found in Section 1.3.9. However, errors returned by the MPI_T interface are not fatal and do not have any impact on the execution of MPI routines.

> *Advice to users.* The number and type of control variables and performance variables can vary between MPI implementations, platforms, and even different builds of the same implementation on the same platform. Hence, any application relying on a particular variable will not be portable.
>
> This interface is primarily intended for performance monitoring tools, support tools, and libraries controlling the application's environment. Application programmers should either avoid using the MPI_T interface or avoid being dependent on the existence of a particular control or performance variable. (*End of advice to users.*)

Since the MPI_T interface mostly focuses on tools and support libraries, MPI_T implementations are only required to provide C bindings. Except where otherwise noted, all conventions and principles governing the C bindings of the MPI API also apply to the MPI_T interface. The MPI_T interface is available by including the `mpi.h` header file.

1.3.1 Verbosity Levels

The MPI_T interface provides users access to internal configuration and performance information through a set of control and performance variables defined by the MPI_T 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 [complexity]level of⊤ (Fin2) detail (basic, detailed or all). See Table 1.3.1 ⊥ (Fin2)

> *Advice to implementors.* If an MPI_T implementation chooses to use only a single verbosity level for all variables, it is recommended that

MPI_T_VERBOSITY_USER_BASIC	Basic information of interest for end users
MPI_T_VERBOSITY_USER_DETAIL	Detailed information of interest for end users
MPI_T_VERBOSITY_USER_ALL	All information of interest for end users
MPI_T_VERBOSITY_TUNER_BASIC	Basic information required for tuning
MPI_T_VERBOSITY_TUNER_DETAIL	Detailed information required for tuning
MPI_T_VERBOSITY_TUNER_ALL	All information required for tuning
MPI_T_VERBOSITY_MPIDEV_BASIC	Basic low-level information for MPI implementors
MPI_T_VERBOSITY_MPIDEV_DETAIL	Detailed low-level information for MPI implementors
MPI_T_VERBOSITY_MPIDEV_ALL	All low-level information for MPI implementors

Table 1.1: MPI_T verbosity levels.

MPI_T_VERBOSITY_USER_BASIC be used. If an MPI_T implementation only uses a single [complexity]level of detail value for all variables in each target audience, it is recommended that all variables be assigned to corresponding BASIC level. (*End of advice to implementors.*)

1.3.2 Binding of MPI_T Variables to MPI Objects

Each MPI_T variable provides access to a particular control setting or performance property provided by the MPI implementation. [These variables can apply globally to no specific object or can refer to a particular MPI object such as a communicator, dataytype, or one-sided communication window. In the latter case,] A variable may refer to a particular 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. In the first case, the variable must be bound to exactly one MPI object before it can be used. Table 1.2 lists all MPI object types to which an MPI_T variable can be bound, together with matching constant that are used by MPI_T routines to identify the object type.

Constant	MPI object
MPI_T_BIND_NO_OBJECT	N/A; applies globally to entire MPI process
MPI_T_BIND_MPI_COMMUNICATOR	MPI communicators
MPI_T_BIND_MPI_DATATYPE	MPI datatypes
MPI_T_BIND_MPI_ERRORHANDLER	MPI error handlers
MPI_T_BIND_MPI_FILE	MPI file handles
MPI_T_BIND_MPI_GROUP	MPI groups
MPI_T_BIND_MPI_OPERATOR	MPI reduction operators
MPI_T_BIND_MPI_REQUEST	MPI requests
MPI_T_BIND_MPI_WINDOW	MPI windows for one-sided communication

Table 1.2: Constants to identify associations of MPI_T control variables.

Rationale. Some variables have meanings tied to a specific MPI object. Examples include the number of send or receive operations using a particular datatype, the number of times an error handler has been called, or or the communication protocol

and "eager limit" used for a particular communicator. Creating a new MPI_T variable for each MPI object could cause the number of variables to grow without bound since they cannot be reused to avoid naming conflicts. By associating MPI_T variables with a specific MPI object, only a single variable must be specified and maintained by the MPI implementation, which can then be reused on as many MPI objects of the respective type as created during the program's execution. (*End of rationale.*)

1.3.3 String Arguments

Several MPI_T functions return one or more strings. These functions have two arguments for each string to be returned: an OUT parameter that identifies a pointer to the buffer in which the string will be returned, and an IN/OUT parameter to pass the length of the buffer. The user is responsible for the memory allocation of the buffer and must pass [⊤ (Fin2) the size of the buffer as the length argument (n).] the size of the buffer (n) as the length argument. Let n be the length value specified to the function. On return, the function⊥ (Fin2) 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 [users]user⊤ (Fin2) passes the null pointer as the length argument, the buffer argument is ignored and nothing⊥ (Fin2) is returned.

MPI_T does not specify the character encoding of strings in the interface. The only requirement is that strings are terminated with a null character. MPI_T reserves all datatype, enumeration datatype items, variables and category names with the prefix `MPI_T` for its own use.

1.3.4 Initialization and Finalization

Since the MPI_T interface is implemented in a separate name space and is independent of the core MPI functions, it requires a separate set of initialization and finalization routines.

MPI_T_INIT_THREAD(required, provided)

IN	required	desired level of thread support (integer)
OUT	provided	provided level of thread support (integer)

```
int MPI_T_Init_thread(int required, int *provided)
```

All programs or tools that use the MPI_T interface must initialize the MPI_T interface before calling any other MPI_T routine. A user can initialize the MPI_T interface by calling MPI_T_INIT_THREAD, which can be called multiple times. In addition, this routine initializes the thread environment. The argument required is used to specify the desired level of thread support. The possible values and their semantics are identical to the ones that can be used with MPI_INIT_THREAD listed in Section **??**. The call returns in provided information about the actual level of thread support that will be provided by MPI_T . It can be one of the four values listed [above]in Section **??**. ⊤ (Fin2) ⊥ (Fin2)

⊤ (Fin2)

Advice to users. [The MPI specification does not require all MPI ranks to be executed before the call to MPI_INIT. If users use MPI_T before MPI_INIT, they need to call MPI_T_INIT_THREAD on every process that is active as this time.] The MPI specification does not require all MPI processes to exist before the call to MPI_INIT. If MPI_T is used before MPI_INIT has been called, MPI_T_INIT_THREAD must be called on each process that exists. Processes created by the MPI implementation during MPI_INIT inherit the status of MPI_T (whether it is initialized or not as well as all active handles) from the process they are created from. (*End of advice to users.*)

⊥ (Fin2)

Advice to implementors. If MPI_T_INIT_THREAD is called before MPI_INIT_THREAD, it is possible that the requested and granted thread level for MPI_T_INIT_THREAD influences the behavior and return value of MPI_INIT_THREAD. The same is true for the reverse order. (*End of advice to implementors.*)

MPI_T_FINALIZE( )

```
int MPI_T_Finalize(void)
```

This routine finalizes the use of the MPI_T 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 is erroneous. 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_T interface remains initialized and calls to all MPI_T 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_T interface is no longer initialized. Further, the call to MPI_T_FINALIZE that ends the initialization of

⊤ (Fin2)

MPI_T may clean up all MPI_T state, invalidate all open sessions ([for the concept of Sessions] see Section 1.3.7), and all handles that have been allocated by MPI_T .

⊥ (Fin2)

MPI_T 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.

1.3.5 Datatype System

Since the initialization of MPI_T is separate from the initialization of MPI, it can not be guaranteed that MPI datatypes are available at any time during the usage of MPI_T . Therefore, the MPI_T interface provides a separate datatype system.

The MPI_T interface requires a significantly simpler type system than MPI itself. All datatypes are represented by a value of type `MPI_T_Datatype` and are classified into two datatype classes: predefined and enumeration datatypes.

MPI_T_DATATYPE_GET_CLASS(datatype, datatypeclass)

IN	datatype	MPI_T datatype to be queried
OUT	datatypeclass	class of the datatype passed in

```
int MPI_T_Datatype_get_class(MPI_T_Datatype datatype, int *datatypeclass)
```

This routine returns the datatype class for the datatype provided by the argument datatype. This allows users of MPI_T to distinguish whether a datatype is an enumeration datatype, e.g., to represent the state of a resource, or is one of the predefined datatypes listed in Table 1.3.5. On return, the typeclass argument is set to one of the constants listed in Table 1.3.5, if datatype represents a valid datatype.

MPI_T Datatype	Equivalent MPI Datatype
MPI_T_INT	MPI_INT
MPI_T_LONG_LONG	MPI_LONG_LONG
MPI_T_CHAR	MPI_CHAR
MPI_T_DOUBLE	MPI_DOUBLE

Table 1.3: Predefined MPI_T datatypes and their MPI equivalents.

MPI_T_DATATYPE_PREDEFINED	the datatype is a predefined datatype
MPI_T_DATATYPE_ENUMERATION	the datatype is an enumeration datatype

Table 1.4: MPI_T datatype classes.

Conforming implementations of MPI_T must ensure that the MPI_T datatypes are equivalent to the listed MPI datatypes for any section of the code in which both MPI and MPI_T can be used. In particular, this requires that the size of a value represented by an MPI_T datatype and its equivalent MPI datatype are equal and that it is possible to communicate the value of a particular MPI_T datatype using the equivalent MPI datatype through regular MPI operations.

> *Rationale.* The concept of equivalent MPI_T and MPI datatypes allows tools to safely communicate values of MPI_T datatypes using MPI message passing functionality. (*End of rationale.*)

The function MPI_T_DATATYPE_GET_SIZE can be used to query the storage size of a variable for each MPI_T datatype.

MPI_T_DATATYPE_GET_SIZE(datatype, size)

IN	datatype	MPI_T datatype to be queried
OUT	size	Number of bytes required to store a value of datatype datatype

```
int MPI_T_Datatype_get_size(MPI_T_Datatype datatype, int *size)
```

The second datatype class, enumeration datatypes, describes variables with a fixed set of discrete values. These datatypes are represented by integer variables and have MPI_INT as their equivalent MPI datatype. Their values range from 0 to $N-1$, with a fixed N that can be queried using MPI_T_DATATYPE_ENUM_GET_INFO.

MPI_T_DATATYPE_ENUM_GET_INFO(datatype, num, name, name_len)

IN	datatype	MPI_T datatype to be queried
OUT	num	number of discrete values represented by this enumeration datatype
OUT	name	buffer to return the string containing the name of the enumeration datatype
INOUT	name_len	length of the string and/or buffer for name

```
int MPI_T_Datatype_enum_get_info(MPI_T_Datatype datatype, int *num, char
              *name, int *name_len)
```

⊤ (Fin2)

[This routine returns, if datatype represents a valid enumeration datatype, N representing the range of the enumeration 0 to $N-1$ as well as a string with a name for it.] If datatype is a valid enumeration datatype, this routine returns the enumeration range and the name of the datatype. If the range runs from 0 to $N-1$, the value N is returned in num. ⊥ (Fin2) []N has to be at least one, i.e., has to represent at least one item. ⊤ (Fin2) The integer values in this range are used to represent the N items that can be represented by this enumeration type. ⊥ (Fin2)

The arguments name and name_len are used to return the name of the datatype as described in Section 1.3.3.

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 MPI_T datatypes used by the MPI implementation.

Names for the individual items in each enumeration datatype can be queried using MPI_T_DATATYPE_ENUM_GET_ITEM.

MPI_T_DATATYPE_ENUM_GET_ITEM(datatype, item, name, name_len)

IN	datatype	MPI_T datatype to be queried
IN	item	item number in the MPI_T datatype to be queried
OUT	name	buffer to return the string containing the name of the enumeration item
INOUT	name_len	length of the string and/or buffer for name

```
int MPI_T_Datatype_enum_get_item(MPI_T_Datatype datatype, int item, char
              *name, int *name_len)
```

The arguments name and name_len are used to return the name of the enumeration item as described in Section 1.3.3.

If completed successfully, the routine is 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 MPI_T enumeration datatype.

1.3.6 Control Variables

The routines described in this section of the MPI_T 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 MPI_T . If N is zero, then the MPI_T 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 MPI_T calls to identify the individual variables.

An MPI_T 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_T implementations are not allowed to change the index of a control variable or delete a variable once it has been added to the set.

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

MPI_T_CVAR_GET_NUM(num)

OUT	num	returns number of control variables

```
int MPI_T_Cvar_get_num(int *num)
```

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

MPI_T_CVAR_GET_INFO(index, name, name_len, verbosity, datatype, count, desc, desc_len, bind, attributes)

IN	index	index of the control variable to be queried
OUT	name	buffer to return the string containing the name of the control variable
INOUT	name_len	length of the string and/or buffer for name
OUT	verbosity	verbosity level of this variable
OUT	datatype	MPI_T datatype of the information stored in the control variable
OUT	count	number of elements returned
OUT	desc	buffer to return the string containing a description of the control variable
INOUT	desc_len	length of the string and/or buffer for desc
OUT	bind	type of MPI object to which this variable must be bound
OUT	attributes	additional attributes defining this variable

```
int MPI_T_Cvar_get_info(int index, char *name, int *name_len, int
              *verbosity, MPI_T_Datatype *datatype, int *count, char *desc,
              int *desc_len,  int *bind, MPI_T_Cvar_attributes *attributes)
```

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

The arguments name and name_len are used to return the name of the control variable as described in Section 1.3.3.

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 MPI_T control variables used by the MPI implementation.

⊤ (Fin2) The argument verbosity returns the verbosity level [(see Section 1.3.1) assigned by the MPI implementation to the variable.] of the variable (see Section 1.3.1).

⊤ (Fin2) ⊥ (Fin2) The argument datatype returns the MPI_T datatype [in which the value for this control variable is returned.] that is used to represent the control variable.

The arguments desc and desc_len are used to return a description of the control variable as described in Section 1.3.3.

Returning a description is optional. If an MPI implementation decides 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 1.3.1).

Additional information about the variable is returned through the attributes argument using an opaque structure of type MPI_T_Cvar_attributes and can be queried using the following accessor function.

⊤ (Fin2) [

Rationale. The use of opaque attributes enables extensions of the MPI_T specification in subsequent versions of the MPI standard without having to redefine or alter the query function. Instead new information can be added by adding new accessor functions. (*End of rationale.*)

]ONLY MOVED

⊥ (Fin2)

MPI_T_CVAR_ATTR_GET_SCOPE(attributes, scope)

IN	attributes	attributes returned by a previous query call
OUT	scope	scope of when changes to this variable are possible

```
int MPI_T_Cvar_attr_get_scope(MPI_T_Cvar_attributes attributes, int *scope)
```

The scope of a variable determines whether an operation either local to the [processor]process ⊤ (Fin2) or collective across multiple processes can change a variable through the MPI_T interface. ⊥ (Fin2) On successful return from MPI_T_CVAR_ATTR_GET_SCOPE , the argument scope will be set to one of the constants listed in Table 1.3.6.

Scope Constant	Description
MPI_T_SCOPE_READONLY	read-only, cannot be written
MPI_T_SCOPE_LOCAL	may be writeable, writing is a local operation
MPI_T_SCOPE_GLOBAL	may be writeable, writing is a global operation

Table 1.5: Scopes for MPI_T 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. If it cannot be changed at a time the user tries to write to it, the MPI_T implementation is allowed to return an error code as the result of the write operation. (*End of advice to users.*)

⊤ (Fin2)

[ONLY MOVED]

Rationale. The use of opaque attributes enables extensions of the MPI_T specification in subsequent versions of the MPI standard without having to redefine or alter the query function. Instead new information can be added by adding new accessor functions. (*End of rationale.*)

⊥ (Fin2)

Handle Allocation and Deallocation

Before reading or writing the value of a variable, a user must first allocate a handle for it by binding it to an MPI object (see also Section 1.3.1).

Rationale. MPI_T handles are distinct from MPI handles because they must be usable before [MPI_Init]MPI_INIT and after [MPI_Finalize]MPI_FINALIZE. Further, ⊤ (Fin2) accessing handles, in particular for performance variables, can be time critical and ⊥ (Fin2) having a separate handle space enables optimizations. (*End of rationale.*) ⊤ (Fin2)

⊥ (Fin2)

MPI_T_CVAR_HANDLE_ALLOC(index, object, handle)

IN	index	index of control variable for which handle is to be allocated
IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound
OUT	handle	allocated handle

```
int MPI_T_Cvar_handle_alloc(int index, void *obj_handle, MPI_T_Cvar_handle
              *handle)
```

⊤ (Fin2)

[This routine allocates a handle for the control variable specified by the argument index and binds this variable to the MPI object referenced by the pointer to its handle passed in the argument obj_handle . The type of the MPI handle passed into this routine must match the type returned by the bind prameter in a prior call to MPI_T_CVAR_GET_INFO. If the type of the object is identified as MPI_T_BIND_NO_OBJECT, i.e., the variable refers to the entire MPI process, the argument object is ignored.] This routine binds the control variable specified by the argument index to the MPI object referenced by the handle passed in argument obj_handle and returns an allocated variable handle in the argument handle. The value of index should be in the range 0 to $N - 1$, where N is the number of available control variables as determined from a prior call to MPI_T_CVAR_GET_NUM. The value of obj_handle must be the memory address of the object's MPI handle, and the type of MPI object it references must be consistent with the type returned in the bind argument in a prior call to MPI_T_CVAR_GET_INFO.

⊥ (Fin2)

MPI_T_CVAR_HANDLE_FREE(handle)

INOUT	handle	handle to be freed

```
int MPI_T_Cvar_handle_free(MPI_T_Cvar_handle *handle)
```

When a handle is no longer needed, a user of MPI_T should call MPI_T_CVAR_HANDLE_FREE to free the handle and the associated resources in the MPI_T implementation. On a successful return, MPI_T sets the handle to MPI_T_CVAR_HANDLE_NULL.

Control Variable Access Functions

MPI_T_CVAR_READ(handle, buf)

IN	handle	handle to the control variable to be read
OUT	buf	initial address of storage location for variable value

```
int MPI_T_Cvar_read(MPI_T_Cvar_handle handle, void* buf)
```

The MPI_T_CVAR_READ queries the value of the control variable identified by the argument handle and stores the result in the buffer buf. The user is responsible to ensure

⊤ (Fin2)
⊥ (Fin2)

that the buffer is of the appropriate size [and fits]to hold the entire value of the control variable (based on the returned datatype and count from a prior corresponding call to MPI_T_CVAR_GET_INFO).

MPI_T_CVAR_WRITE(handle, buf)

IN	handle	handle to the control variable to be written
IN	buf	initial address of storage location for variable value

```
int MPI_T_Cvar_write(MPI_T_Cvar_handle handle, void* buf)
```

The MPI_T_CVAR_WRITE sets the value of the control variable identified by the argument handle to the data stored in the buffer buf. The user is responsible to ensure that the buffer is of the appropriate size [and fits]to hold the entire value of the control ⊤ (Fin2)
variable (based on the returned datatype and count from a prior corresponding call to ⊥ (Fin2)
MPI_T_CVAR_GET_INFO).

If the variable has a global scope (as returned by a prior corresponding MPI_T_CVAR_ATTR_GET_SCOPE call), any write call to this variable must be issued consistently in all connected (as defined in Section ??) MPI processes. The user is responsible to ensure that the writes in all processes are consistent.

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

1.3.7 Performance Variables

The following section focuses on the ability to list and 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. Performance variables are always local to [a single]an MPI process. ⊤ (Fin2)
⊥ (Fin2)

> *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 [has a]provides cleaner semantics and ⊤ (Fin2)
> allows for more performance optimization opportunities. (*End of rationale.*) ⊥ (Fin2)

Performance Variable Classes

⊤ (Fin2)

[Each performance variable is associated with a class describing its [the] basic semantics. ⊤ (Fin2)
The class of a variable also defines its basic behavior, when and how an MPI implementation ⊥ (Fin2)
can change its value, and what the initial value of this variable is at the time it is either used for the first time or reset. Each performance variable is associated with a class that describes its basic semantics, basic behavior, its starting value, and when and how an MPI implementation can changes its value. The starting value is the value the variable assumes when it is used for the first time or whenever it is reset. In the following this referred to as the starting value. Further, [it]the class of the variable also defines which datatypes can ⊤ (Fin2)
⊥ (Fin2)

be used to represent it.]These 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 expected to be]are represented by an enumeration datatype and can be set by the MPI implementation at any time. The [default] starting value is the current state [(at the time the starting value is set) of the implementation.] of the implementation at the time the starting value is set .

- 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 represented by one of the following datatypes: MPI_T_INT, MPI_T_LONG_LONG, MPI_T_DOUBLE. The [default] starting value is the current utilization level [(at the time the starting value is set) of the resource.] of the resource at the time the starting value is set.

- 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_T_DOUBLE datatype. The value must always be between 0.0 (resource not used at all) and 1.0 (resource completely used). The [default] starting value is the current percentage utilization level [(at the time the starting value is set) of the resource.] of the resource at the time the starting value is set.

- 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 grows monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: MPI_T_INT, MPI_T_LONG_LONG, MPI_T_DOUBLE. The [default] starting value is the current utilization level [(at the time the starting value is set) of the resource.] of the resource at the time the starting value is set.

- 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 decreases monotonically from the initialization or reset of the variable. It can be represented by one of the following datatypes: MPI_T_INT, MPI_T_LONG_LONG, MPI_T_DOUBLE. The [default] starting value is the current utilization level [(at the time the starting value is set) of the resource.] of the resource at the time the starting value is set.

- MPI_T_PVAR_CLASS_COUNTER
  A performance variable in this class counts the number of occurrences of a specific event [during the execution time of an application] (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_T_INT, MPI_T_LONG_LONG. The [default]

starting value for variables of this class is 0.

- MPI_T_PVAR_CLASS_AGGREGATE
  The value of a performance variable in this class is an 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_T_INT, MPI_T_LONG_LONG, MPI_T_DOUBLE. The [default] starting value for variables of this class is 0. ⊤ (Fin2) ⊥ (Fin2)

- 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 or type of event. 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_T_INT, MPI_T_LONG_LONG, MPI_T_DOUBLE. The [default] starting value for variables ⊤ (Fin2) [if]of this class is 0. If the type MPI_T_DOUBLE ⊥ (Fin2) is used, the units representing time ⊤ (Fin2) in this datatype must match the units used by MPI_WTIME. ⊥ (Fin2)

Performance Variable Query Functions

An MPI implementation exports a set of N performance variables through MPI_T . If N is zero, then the MPI_T 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 MPI_T calls to identify the individual variables.

An MPI_T 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_T implementations are not allowed to change the index of a performance variable or delete a variable once it has been added to the set.

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

MPI_T_PVAR_GET_NUM(num)

OUT	num	returns number of performance variables

```
int MPI_T_Pvar_get_num(int *num)
```

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

MPI_T_PVAR_GET_INFO(index, name, name_len, verbosity, varclass, datatype, count, desc, desc_len, bind, attributes)

IN	index	index of the performance variable to be queried
OUT	name	buffer to return the string containing the name of the performance variable
INOUT	name_len	length of the string and/or buffer for name
OUT	verbosity	verbosity level of this variable
OUT	var_class	class of performance variable
OUT	datatype	MPI_T datatype of the information stored in the performance variable
OUT	count	number of elements returned
OUT	desc	buffer to return the string containing a description of the performance variable
INOUT	desc_len	length of the string and/or buffer for desc
OUT	bind	type of MPI object to which this variable must be bound
OUT	attributes	additional attributes defining this variable

```
int MPI_T_Pvar_get_info(int num, char *name, int *name_len, int *verbosity,
              int *var_class, MPI_T_Datatype *datatype, int *count, char
              *desc, int *desc_len, int *bind, MPI_T_Pvar_attributes
              *attributes)
```

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

The arguments name and name_len are used to return the name of the performance variable as described in Section 1.3.3.

If completed successfully, 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 MPI_T performance variables used by the MPI implementation.

⊤ (Fin2) ⊥ (Fin2) The argument verbosity returns the verbosity level [(see Section 1.3.1) assigned by the MPI implementation to the variable.] of the variable (see Section 1.3.1).

⊤ (Fin2) ⊥ (Fin2) The class of the performance variable is returned in the parameter var_class [and can]. The class must be one of the constants defined in Section 1.3.7.

⊤ (Fin2) ⊥ (Fin2) The argument datatype returns the MPI_T datatype [in which the value for this performance variable is returned.] that is used to represent the performance variable. The value consists of count elements of this datatype.

⊤ (Fin2) ⊥ (Fin2) The arguments desc and desc_len are used to return a description of the [control]performance variable as described in Section 1.3.3.

Returning a description is optional. If an MPI implementation decides 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 from this function.

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 1.3.1).

Additional information about the variable is returned through the attributes argument using an opaque structure of type MPI_T_Pvar_attributes and can be queried using the following accessor functions.

MPI_T_PVAR_ATTR_GET_READONLY(attributes, readonly)

IN	attributes	attributes returned by a previous query call
OUT	readonly	flag indicating whether a variable can be written/reset

```
int MPI_T_Pvar_attr_get_readonly(MPI_T_Pvar_attributes attributes, int
              *readonly)
```

Upon return, the argument readonly [will be]is set to zero if the variable can be written or reset by the user[,]or. It is set to one if the variable can only be read. ⊤ (Fin2) ⊥ (Fin2) ⊤ (Fin2) ⊥ (Fin2)

MPI_T_PVAR_ATTR_GET_CONTINUOUS(attributes, continuous)

IN	attributes	attributes returned by a previous query call
OUT	continuous	flag indicating whether a variable can be started and stopped or is continuously active

```
int MPI_T_Pvar_attr_get_continuous(MPI_T_Pvar_attributes attributes, int
              *continuous)
```

Upon return, the argument continuous [will be]is set to zero if the variable can be started and stopped by the user, i.e, it is possible for the user to control if and when the value of a variable is updated [, or]. It is set to one if the variable is [automatically]always active and [can not by]cannot be controlled by the user. ⊤ (Fin2) ⊥ (Fin2) ⊤ (Fin2) ⊥ (Fin2) ⊤ (Fin2) ⊥ (Fin2) ⊤ (Fin2) ⊥ (Fin2)

Performance Experiment Sessions

Within a single program, multiple components can use the MPI_T interface. To avoid collisions with respect to accesses to performance variables, users of the MPI_T interface must first create a session. All subsequent calls accessing performance variables are then within the context of this session. Any call executed in a session must not influence the results in any other session.

MPI_T_PVAR_SESSION_CREATE(session)

OUT	session	identifier of performance session

```
int MPI_T_Pvar_session_create(MPI_T_Pvar_session *session)
```

This call creates a new session for accessing performance variables and returns an identifier for this session in the argument session.

MPI_T_PVAR_SESSION_FREE(session)

INOUT	session	identifier of performance experiment session

```
int MPI_T_Pvar_session_free(MPI_T_Pvar_session *session)
```

This call frees an existing session[, i.e., calls]Calls to MPI_T can no longer be made within the [context of the MPI freed session.]context of a session after it is freed. This call also frees all handles that have been allocated within the specified session (see below for handle allocation and freeing). On a successful return, MPI_T sets the session identifier to MPI_T_PVAR_SESSION_NULL.

Handle Allocation and Deallocation

Before using a performance variable, a user must first allocate a handle for it by binding it to an MPI object (see also Section 1.3.1). [The type of the MPI object is returned by a previous call to MPI_T_PVAR_GET_INFO in the bind argument.]

MPI_T_PVAR_HANDLE_ALLOC(session, index, objhandle, handle)

IN	session	identifier of performance experiment session
IN	index	index of performance variable for which handle is to be allocated
IN	obj_handle	reference to a handle of the MPI object to which this variable is supposed to be bound
OUT	handle	allocated handle

```
int MPI_T_Pvar_handle_alloc(MPI_T_Pvar_session session, int index, void
              *obj_handle, MPI_T_Pvar_handle *handle)
```

[A call to this routine allocates a handle for the performance variable specified by the argument index and binds this variable to the MPI object referenced by the pointer to its handle passed in the argument obj_handle . The type of the MPI object passed into this routine must match the type of the MPI object for this variable as returned by a prior call to MPI_T_PVAR_GET_INFO. If the type of the object is identified as MPI_T_BIND_NO_OBJECT, i.e., the variable refers to the entire MPI implementation the argument object is ignored.] This routine binds the performance variable specified by the argument index to the MPI object referenced by the handle passed in argument obj_handle and returns an allocated variable handle in the argument handle. The value of index should be in the range 0 to $N-1$, where N is the number of available control variables as determined from a prior call to MPI_T_PVAR_GET_NUM. The value of obj_handle must be the memory address of the object's MPI handle, and the type of 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.

MPI_T_PVAR_HANDLE_FREE(session, handle)

IN	session	identifier of performance experiment session
INOUT	handle	handle to be freed

```
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 MPI_T should call MPI_T_PVAR_HANDLE_FREE to free the handle and the associated resources in the MPI_T implementation. On a successful return, MPI_T sets the handle to MPI_T_PVAR_HANDLE_NULL.

Starting and Stopping of Performance Variables

⊤ (Fin2)

[Performance variables that have the continuous flag set during the query operation are continuously operating once a handle has been allocated and can be queried any time. They cannot be stopped or paused by the user. All other variables are in a stopped state after their handle has been allocated; their values are not updated as the program executes, and must be started by the user.] 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 stopped or paused 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.

⊥ (Fin2)

MPI_T_PVAR_START(session, handle)

IN	session	identifier of performance experiment session
IN	handle	handle of a performance variable

```
int MPI_T_Pvar_start(MPI_T_Pvar_session session, MPI_T_Pvar_handle handle)
```

This functions starts the performance variable with the handle handle in the session 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_T_SUCCESS if all variables are started successfully, otherwise MPI_T_ERR_NOSTARTSTOP is returned. Continuous variables and variables that are already started are ignored when [used with ⊤ (Fin2) MPI_T_PVAR_ALL_HANDLES.] MPI_T_PVAR_ALL_HANDLES is specified. ⊥ (Fin2)

MPI_T_PVAR_STOP(session, handle)

IN	session	identifier of performance experiment session
IN	handle	handle of a performance variable

```
int MPI_T_Pvar_stop(MPI_T_Pvar_session session, MPI_T_Pvar_handle handle)
```

This functions stops the performance variable with the handle handle in the session 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, otherwise MPI_T_ERR_NOSTARTSTOP is returned. Continuous variables and variables that are already stopped are ignored when [(]used with MPI_T_PVAR_ALL_HANDLES. MPI_T_PVAR_ALL_HANDLES is specified.

Performance Variable Access Functions

MPI_T_PVAR_READ(session, handle, buf)

IN	session	identifier of performance experiment session
IN	handle	handle of a performance variable
OUT	buf	initial address of storage location for variable value

```
int MPI_T_Pvar_read(MPI_T_Pvar_session session, MPI_T_Pvar_handle  handle,
              void* buf)
```

The MPI_T_PVAR_READ call queries the value of the performance variable with the handle handle in the session identified by the parameter session and stores the result in the buffer buf. The user is responsible to ensure that the buffer is of the appropriate size [and fits]to hold the entire value of the performance variable (based on the returned datatype and count during the MPI_T_PVAR_GET_INFO call).

[Note that the constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the MPI_T function MPI_T_PVAR_READ, since this would require the function to return a set of variable values instead of just one.] The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the MPI_T function MPI_T_PVAR_READ.

MPI_T_PVAR_WRITE(session,handle, buf)

IN	session	identifier of performance experiment session
IN	handle	handle of a performance variable
IN	buf	initial address of storage location for variable value

```
int MPI_T_Pvar_write(MPI_T_Pvar_session session, MPI_T_Pvar_handle handle,
              void* buf)
```

The MPI_T_PVAR_WRITE call attempts to write the value of the performance variable with the handle handle in the session identified by the parameter session. The value to be written is passed in the buffer buf. The user is responsible to ensure that the buffer is of the appropriate size [and fits]to hold the entire value of the performance variable (based on the returned datatype and count during the MPI_T_PVAR_GET_INFO call).

If it is not possible to change the variable, the function returns MPI_T_ERR_PVAR_WRITE.

[Note that the constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the MPI_T function MPI_T_PVAR_WRITE, since this would require the function to accept a set of variable values instead of just one.] The constant MPI_T_PVAR_ALL_HANDLES cannot be used as an argument for the MPI_T function MPI_T_PVAR_WRITE. ⊥ (Fin2)

MPI_T_PVAR_RESET(session, handle)

IN	session	identifier of performance experiment session
IN	handle	handle of a performance variable

```
int MPI_T_Pvar_reset(MPI_T_Pvar_session session, MPI_T_Pvar_handle  handle)
```

The MPI_T_PVAR_RESET call sets [of] the performance variable with the handle handle ⊤ (Fin2) to its starting value specified in Section 1.3.7. If it is not possible to change the variable, ⊥ (Fin2) the function returns MPI_T_ERR_PVAR_WRITE.

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_T_SUCCESS if all variables are reset successfully, otherwise MPI_T_ERR_NOWRITE is returned. Readonly variables are ignored when [used with MPI_T_PVAR_ALL_HANDLES.] ⊤ (Fin2) MPI_T_PVAR_ALL_HANDLES is specified. ⊥ (Fin2)

MPI_T_PVAR_READRESET(session, handle, buf)

IN	session	identifier of performance experiment session
IN	handle	handle of a performance variable
OUT	buf	initial address of storage location for variable value

```
int MPI_T_Pvar_readreset(MPI_T_Pvar_session session, MPI_T_Pvar_handle
              handle, void* buf)
```

This call 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. ⊤ (Fin2)

[Note that the constant MPI_T_PVAR_ALL_HANDLES can not be used as an argument for the MPI_T function MPI_T_PVAR_READRESET, since this would require the function to return a set of variable values instead of just one.] The constant MPI_T_PVAR_ALL_HANDLES can not be used as an argument for the MPI_T function MPI_T_PVAR_READRESET. ⊥ (Fin2)

Advice to implementors. Although MPI places no requirements on the interaction with external mechanisms such as signal handlers, it is strongly recommended that all routines to start, stop, read, write, and reset performance variables should be safe to call in asynchronous contexts. Examples of asynchronous contexts include signal handlers and interrupt handlers. Such safety permits the development of sampling-based tools. High quality implementations should strive to make the results of any such interactions intuitive to users, and document known restrictions. (*End of advice to implementors.*)

1.3.8 Variable Categorization

MPI implementations can optionally group performance and control variables into categories to express logical relationships between various variables. For example, an MPI_T implementation could group all control and performance variables that refer to message transfers in the MPI implementation and thereby [distinguishing]distinguish them from variables that refer to local resources such as memory allocations or other interactions with the operating system.

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

The category information may be queried in a fashion similar to the mechanism for querying variable information. The MPI implementation exports a set of N categories via the MPI_T interface. If $N = 0$, then the MPI implementation does not export any categories, otherwise the provided performance variables are indexed from 0 to $N - 1$. This index number is used in subsequent MPI_T calls 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.

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

MPI_T_CATEGORY_GET_NUM(num)

OUT	num	current number of categories

```
int MPI_T_Category_get_num(int *num)
```

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

MPI_T_CATEGORY_GET_INFO(index, name, name_len, desc, desc_len, num_controlvars, num_perfvars, num_categories)

IN	index	index of the category to be queried
OUT	name	buffer to return the string containing the name of the category
INOUT	name_len	length of the string and/or buffer for name
OUT	desc	buffer to return the string containing the description of the category
INOUT	desc_len	length of the string and/or buffer for desc
OUT	num_controlvars	number of control variables in the category
OUT	num_perfvars	number of performance variables in the category
OUT	num_categories	number of MPI_T categories contained in the category

```
int MPI_T_Category_get_info(int index, char *name, int *name_len, char
              *desc, int *desc_len, int *num_controlvar s, int
              *num_perfvars, int *num_categories)
```

The arguments name and name_len are used to return the name of the category as described in Section 1.3.3.

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 MPI_T categories used by the MPI_T implementation.

The arguments desc and desc_len are used to return the description of the category as described in Section 1.3.3.

Returning a description is optional. If an MPI implementation decides 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 function returns the number of control variables, performance variables and other categories contained in the queried category in the arguments num_controlvars, num_perfvars, and num_categories respectively.

> *Advice to implementors.* To avoid confusion and to simplify the interpretation of the categories provided by a particular implementation, it is recommended that categories should either only contain other categories or only control and performance variables. Mixing categories and control and performance variables within a single category is not recommended. (*End of advice to implementors.*)

MPI_T_CATEGORY_GET_CVARS(cat_index, len, indices)

IN	cat_index	index of the category to be queried, in the range $[0, N-1]$
IN	len	the length of the indices array
OUT	indices	an integer array of size len, indicating control variable indices

```
int MPI_T_Category_get_cvars(int cat_index, int len, int indices[])
```

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

MPI_T_CATEGORY_GET_PVARS(cat_index,len,indices)

IN	cat_index	index of the category to be queried, in the range $[0, N-1]$
IN	len	the length of the indices array
OUT	indices	an integer array of size len, indicating performance variable indices

```
int MPI_T_Category_get_pvars(int cat_index, int len, int indices[])
```

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)

IN	cat_index	index of the category to be queried, in the range $[0, N-1]$
IN	len	the length of the indices array
OUT	indices	an integer array of size len, indicating category indices

```
int MPI_T_Category_get_categories(int cat_index, int len, int indices[])
```

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.

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

The user is responsible for allocating the arrays passed into the functions MPI_T_CATEGORY_GET_CVARS, MPI_T_CATEGORY_GET_PVARS and MPI_T_CATEGORY_GET_CATEGORIES. [The functions will only write up to len elements into the respective array. If the category contains more than len variables or other categories respectively, the function returns an arbitrary subset; if it contains less than len variables

⊤ (Fin2)

or other categories respectively, all will be returned and the remaining array entries will not be modified.] Starting from array index 0, each function writes up to len elements into the array. If the category contains more than len elements, the function returns an arbitrary subset of size len. Otherwise, the entire set of elements is returned in the beginning entries of the array, and any remaining array entries are not modified."

1.3.9 Return and Error Codes

All MPI_T functions return an integer error code (See Table 1.3.9). [None of the error codes returned by an MPI_T routine are fatal to the MPI process or invoke an MPI error handler.] errors returned by MPI_T routines are not fatal and do not invoke MPI error handlers. The execution of the MPI process continues as if the MPI_T call would have succeeded. However, the MPI_T implementation is not required to check all user provided parameters; if a user passes invalid parameter values to any MPI_T routine the behavior of the implementation is undefined.

1.3.10 Profiling Interface

All requirements for the profiling interfaces, as described in Section 1.2, also apply to the MPI_T interface. In particular, this means that [a complying]compliant MPI_T implementation[s] must provide matching PMPI_T calls for every MPI_T call. All rules, guidelines, and recommendations from Section 1.2 apply equally to PMPI_T calls.

Return Code	Description
Return Codes for all MPI_T Functions	
MPI_T_SUCCESS	No error, call completed
MPI_T_ERR_MEMORY	Out of memory
MPI_T_ERR_NOTINITIALIZED	MPI_T not initialized
MPI_T_ERR_CANTINIT	MPI_T not in the state to be initialized
Return Codes for Datatype Functions: MPI_T_DATATYPE_*	
MPI_T_ERR_PREDEFINED	Datatype is a predefined datatype and not an enumeration
MPI_T_ERR_INVALIDDATATYPE	Datatype is not a valid datatype
MPI_T_ERR_INVALIDITEM	The item index queried is out of range (for MPI_T_DATATYPE_ENUMITEM only)
Return Codes for variable and category query functions: MPI_T_*_GET_INFO	
MPI_T_ERR_INVALIDINDEX	The variable or category index is invalid
Return Codes for Handle Functions: MPI_T_*_ALLOCATE,FREE	
MPI_T_ERR_INVALIDINDEX	The variable index is invalid
MPI_T_ERR_INVALIDHANDLE	The handle is invalid
MPI_T_ERR_OUTOFHANDLES	No more handles available
Return Codes for Session Functions: MPI_T_PVAR_SESSION_*	
MPI_T_ERR_OUTOFSESSIONS	No more sessions available
MPI_T_ERR_INVALIDSESSION	Session argument is not a valid session
Return Codes for Control Variable Access Functions: MPI_T_CVAR_READ, WRITE	
MPI_T_ERR_SETNOTNOW	Variable cannot be set at this moment
MPI_T_ERR_SETNEVER	Variable cannot be set until end of execution
MPI_T_ERR_INVALIDVAR	Control variable does not exist
MPI_T_ERR_INVALIDHANDLE	The handle is invalid
Return Codes for Performance Variable Access and Control: MPI_T_PVAR_START, STOP, READ, WRITE, RESET, READRESET	
MPI_T_ERR_INVALIDHANDLE	The handle is invalid
MPI_T_ERR_INVALIDSESSION	Session argument is not a valid session
MPI_T_ERR_NOSTARTSTOP	Variable can not be started or stopped for MPI_T_PVAR_START and MPI_T_PVAR_STOP
MPI_T_ERR_NOWRITE	Variable can not be written or reset for MPI_T_PVAR_WRITE and MPI_T_PVAR_RESET
Return Codes for Category Functions: MPI_T_CATEGORY_*	
MPI_T_ERR_INVALIDCATEGORY	The specified category index does not exist

Table 1.6: Return and error codes used MPI_T functions.

Bibliography

[1] mpi-debug: Finding Processes. http://www-unix.mcs.anl.gov/mpi/mpi-debug/.

[2] James Cownie and William Gropp. A Standard Interface for Debugger Access to Message Queue Information in MPI. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machin e and Message Passing Interface*, pages 51–58, Barcelona, Spain, September 1999.

Examples Index

This index lists code examples throughout the text. Some examples are referred to by content; others are listed by the major MPI_T function that they are demonstrating. MPI_T functions listed in all capital letter are Fortran examples; MPI_T functions listed in mixed case are C/C++ examples.