MPI: A Message-Passing Interface Standard

Version 3.0

⊤ (Fin2)
⊥ (Fin2)

Message Passing Interface Forum

Draft January 7, 2011

Contents

List of Figures

List of Tables

Chapter 1

Tool Interfaces for MPI

1.1 Introduction

This chapter discusses a set of interfaces that allows tools such as debuggers, performance analyzers, and others to extract information about the operation of MPI processes. Specifically, this chapter defines the PMPI profiling interface (Section **??**) to transparently intercept and inspect any MPI call; and the MPIT tool information interface (Section 1.2) to query 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 as any other MPI function.

1.2 MPIT Performance 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 MPIT interface described in this section provides access to this information.

The purpose of the MPIT interface is to provide a mechanism for the MPI implementation to expose a set of variables that represent a particular property, setting, or performance measurement from within the MPI implementation. The MPIT interface provides the necessary routines to find all variables that exist in the particular MPI implementation, to query their properties, to retrieve descriptions about their meaning, and to 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 MPIT interface allows the implementation to specify which control and performance variables exist. For both types of variables, the interface provides the ability to query the variables offered by the particular MPI implementation, along with additional semantics and descriptions.

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

On success all MPIT routines return MPIT_SUCCESS, otherwise they return an appropriate error code. Details on error codes can be found in Section 1.2.9. However, errors returned by the MPIT interface shall not be fatal nor 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 no longer be portable.
>
> This interface is primarily intended for performance monitoring tools, as well as support tools and libraries controlling the application's environment. Application programmers should either avoid using it or avoid being dependent on the existence of a particular control or performance variable. (*End of advice to users.*)

Since the MPIT interface mostly focuses on tools and support libraries, the MPIT implementations are only required to provide C Bindings. Except where otherwise notes, all conventions and principles governing the C Bindings of the MPI API also apply to the MPIT interface and the MPIT interface shall be defined in the same header or API definition file(s) as the regular MPI routines.

1.2.1 Verbosity Levels

The MPIT interface provides users access to internal configuration and performance information through a set of control and performance variables, which are defined by the MPIT implementation. Since the number of variables can be large for particular implementations, every variable exported by the MPIT interface has to be associated with one of the following verbosity levels.

MPIT_VERBOSITY_USER_BASIC	Basic information of interest for end users
MPIT_VERBOSITY_USER_DETAILED	Detailed information of interest for end users
MPIT_VERBOSITY_USER_VERBOSE	All information of interest for end users
MPIT_VERBOSITY_TUNER_BASIC	Basic information required for tuning
MPIT_VERBOSITY_TUNER_DETAILED	Detailed information required for tuning
MPIT_VERBOSITY_TUNER_VERBOSE	All information required for tuning
MPIT_VERBOSITY_MPIDEV_BASIC	Basic low-level information for MPI developers
MPIT_VERBOSITY_MPIDEV_DETAILED	Detailed low-level information for MPI developers
MPIT_VERBOSITY_MPIDEV_VERBOSE	All low-level information for MPI developers

Table 1.1: MPIT verbosity levels.

Implementations have to assign each variable to one of the verbosity levels. MPI implementations should sort all variables according to the intended target audience (end user, performance optimizers, or MPI developer) and then distinguish three levels of verbosity (basic, detailed, and verbose) within each audience.

> *Advice to implementors.* If an MPIT implementation only uses a single verbosity level for all variables, it is recommended to assign all variables to the level

MPI_VERBOSITY_USER_BASIC. If an MPIT implementation only uses a single verbosity level for all variables intended for each target audience, it is recommended to assign all variables to corresponding basic level. (*End of advice to implementors.*)

1.2.2 Associations between MPIT Variables and MPI Resources

Each variable provides access to a particular control setting or performance property provided by the MPI implementation. The meaning of these variables can refer to the complete MPI library as a global variable or can be associated with a particular MPI resource, such as a communicator, dataytype, or one-sided communication window. In the latter case, the variable is associated with exactly one MPI resource type. Before it can be used, it has to be instantiated with an instance of an MPI resource of that type. Table 1.2 lists all types of MPI resources supported by MPIT along with a corresponding constant used by the MPIT interface to identify that resource type.

Constant	Associated MPI resource
MPIT_MPI_RESOURCE_TYPE_GLOBAL	N/A — global meaning
MPIT_MPI_RESOURCE_TYPE_COMMUNICATOR	MPI communicators
MPIT_MPI_RESOURCE_TYPE_DATATYPE	MPI datatypes
MPIT_MPI_RESOURCE_TYPE_ERRORHANDLER	MPI error handler
MPIT_MPI_RESOURCE_TYPE_FILE	MPI file handles
MPIT_MPI_RESOURCE_TYPE_GROUP	MPI groups
MPIT_MPI_RESOURCE_TYPE_OPERATOR	MPI reduction operators
MPIT_MPI_RESOURCE_TYPE_REQUEST	MPI requests
MPIT_MPI_RESOURCE_TYPE_WINDOW	MPI windows for one-sided communication

Table 1.2: Constant to identify associations of MPIT control variables.

Rationale. Certain variables have meanings that are limited to a particular MPI resource. Examples are 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 separate variable for each MPI resource, e.g., for each communicator, would cause the number of variables to grow unboundedly since they cannot be reused to avoid naming conflicts. By associating variables with MPI resource types, only a single variable has to be created and maintained by the MPI implementation, which can then be reused on as many instances of this MPI resource type as created during the program's execution. (*End of rationale.*)

In order to instantiate a variable with a particular MPI resource instance, the user has to be able to convert a reference to a resource of each supported type to a generic reference, which can then be passed to the MPIT routine responsible for instantiating the MPIT variable. For this purpose, the interface offers the following conversion routines, which each take a reference to an MPI resource and return a reference to a generic MPI resource of type MPIT_MPI_Resource.

MPIT_MPI_RESOURCE_COMMUNICATOR(communicator, resource)

IN	communicator	Reference to an MPI communicator
OUT	resource	Reference to a generic MPI resource

```
int MPIT_MPI_Resource_communicator(MPI_Comm *communicator,
             MPIT_MPI_Resource *resource)
```

MPIT_MPI_RESOURCE_DATATPE(datatype, resource)

IN	datatype	Reference to an MPI datatype
OUT	resource	Reference to a generic MPI resource

```
int MPIT_MPI_Resource_datatype(MPI_Datatype *datatype, MPIT_MPI_Resource
             *resource)
```

MPIT_MPI_RESOURCE_ERRORHANDLER(errorhandler, resource)

IN	errorhandler	Reference to an MPI error handler
OUT	resource	Reference to a generic MPI resource

```
int MPIT_MPI_Resource_errorhandler(MPI_Errorhandler *errorhandler,
             MPIT_MPI_Resource *resource)
```

MPIT_MPI_RESOURCE_GROUP(group, resource)

IN	group	Reference to an MPI group
OUT	resource	Reference to a generic MPI resource

```
int MPIT_MPI_Resource_group(MPI_Group *group, MPIT_MPI_Resource *resource)
```

MPIT_MPI_RESOURCE_FILE(file, resource)

IN	file	Reference to an MPI files
OUT	resource	Reference to a generic MPI resource

```
int MPIT_MPI_Resource_file(MPI_File *file, MPIT_MPI_Resource *resource)
```

MPIT_MPI_RESOURCE_OPERATIONS(operation, resource)

IN	operation	Reference to an MPI reduction operation
OUT	resource	Reference to a generic MPI resource

```
int MPIT_MPI_Resource_operation(MPI_Ops *operation, MPIT_MPI_Resource
              *resource)
```

MPIT_MPI_RESOURCE_REQUEST(request, resource)

IN	request	Reference to an MPI asynchronous communication request
OUT	resource	Reference to a generic MPI resource

```
int MPIT_MPI_Resource_request(MPI_Request *request, MPIT_MPI_Resource
              *resource)
```

MPIT_MPI_RESOURCE_WINDOW(window, resource)

IN	window	Reference to an MPI one-sided communication window
OUT	resource	Reference to a generic MPI resource

```
int MPIT_MPI_Resource_window(MPI_Window *window, MPIT_MPI_Resource
              *resource)
```

Additionally, the MPIT interface provides the constant MPIT_MPI_RESOURCE_GLOBAL of type MPIT_MPI_Resource* that can be used in routines that expect a reference to an MPI resource if the resource type is MPIT_MPI_RESOURCE_TYPE_GLOBAL.

1.2.3 String Arguments

Several MPIT function return one or more strings. These functions have two arguments for each string to be returned, one identifying a pointer to the buffer in which the string will be returned, and one 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 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. The buffer is always null-terminated. 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.

MPIT does not specify the character encoding of strings in the interface. The only requirement is that strings are terminated with a null character. MPI reserves all type, enumeration type item, variables, and category names with the prefixes "MPI_" and "MPIT_" for its own use.

1.2.4 Initialization and Finalization

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

MPIT_INIT()

`int MPIT_Init(void)`

All programs or tools that use the MPIT interface has to initialize the MPIT interface before calling any other MPIT routine. A user can initialize the MPIT interface by calling MPIT_INIT, which can be called multiple times.

MPIT_FINALIZE( )

`int MPIT_Finalize(void)`

This routine finalizes the use of the MPIT interface and may be called as often as the corresponding MPIT_INIT routine up to the current point of execution. Calling it more times is erroneous. As long as the number of calls to MPIT_FINALIZE is smaller than the number of calls to MPIT_INIT up to the current point of execution, the MPIT interface remains initialized and calls to all MPIT routines are permissible. Further, additional calls to MPIT_INIT after one or more calls to MPIT_FINALIZE are permissible.

Once MPIT_FINALIZE is called the same number of times as the routine MPIT_INIT up to the current point of execution, the MPIT interface is no longer initialized. Further, the call to MPIT_FINALIZE that ends the initialization of MPIT may clean up all MPIT state, invalidate all open sessions (for the concept of Sessions see Section 1.2.7), and free all handles that have been allocated by MPIT. MPIT can be reinitialized by subsequent calls to MPIT_INIT.

At the end of the program execution, unless MPI_ABORT is called, an application shall have called MPIT_INIT and MPIT_FINALIZE an equal number of times.

1.2.5 Type System

Since the initialization of MPIT is separate from the initialization of MPI, it can not be guaranteed that MPI data types are available at any time during the usage of MPIT. Therefore, the MPIT interface provides a separate type system. All types are represented by a variable or constant of type MPIT_Datatype and are classified into two type classes: predefined and enumeration types. The Table 1.3 lists all available constants that can be used to identify or describe a predefined type for MPIT calls.

MPIT_TYPE_GET_CLASS(datatype, typeclass)

IN	datatype	MPIT datatype to be queried
OUT	typeclass	class of the type passed in

`int MPIT_Type_get_class(MPIT_Datatype datatype, int *typeclass)`

This routine returns the type class for the datatype provided by the argument datatype. This allows users of MPIT to distinguish whether a datatype is an enumeration type, e.g., to represent the state of a resource, or is one of the predefined types listed in Table 1.3. On return, the typeclass argument is set to one of the constants listed in Table 1.4, if datatype represents a valid type.

MPIT Datatype	Equivalent MPI Datatype
MPIT_LOGICAL	MPI_LOGICAL
MPIT_BYTE	MPI_BYTE
MPIT_SHORT	MPI_SHORT
MPIT_INT	MPI_INT
MPIT_LONG	MPI_LONG
MPIT_LONG_LONG	MPI_LONG_LONG
MPIT_CHAR	MPI_CHAR
MPIT_FLOAT	MPI_FLOAT
MPIT_DOUBLE	MPI_DOUBLE

Table 1.3: Predefined MPIT datatypes and their MPI equivalents.

MPIT_TYPECLASS_PREDEFINED	the datatype is a predefined datatype
MPIT_TYPECLASS_ENUMERATION	the datatype is an enumeration datatype

Table 1.4: MPIT type classes.

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

> *Rationale.* The concept of equivalent MPIT and MPI datatypes allows to safely communicate values of MPIT datatypes using regular MPI messages. (*End of rationale.*)

The function MPIT_TYPE_GET_SIZE can be used to query the storage size for each MPIT datatype.

MPIT_TYPE_GET_SIZE(datatype, size)

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

`int MPIT_Type_get_size(MPIT_Datatype datatype, int *size)`

The second type class, enumeration types, describe variables with a fixed set of discrete values. These types are represented through integer variables and have MPI_INT as their equivalent MPI type. Their values range from 0 to $N-1$, with a fixed N that can be queried using MPIT_TYPE_ENUM_QUERY.

MPIT_TYPE_ENUM_GET_INFO(datatype, num, name, name_len)

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

```
int MPIT_Type_enum_get_info (MPIT_Datatype datatype, int *num, char *name,
              int *name_len)
```

This routine returns, if datatype represents a valid enumeration type, the size of the enumeration as well as a name for it.

The arguments name and name_len are used to return the name of the type as described in Section 1.2.3. .

If completed successfully, the routine is required to return a name of at least length one, which is unique with respect to all other names for MPIT datatypes used by the MPI implementation.

Names for the individual items in each enumeration type can be queried using MPIT_TYPE_ENUM_GET_ITEM.

MPIT_TYPE_ENUM_GET_ITEM(datatype, item, name, name_len)

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

```
int MPIT_Type_enum_get_item (MPIT_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 1.2.3.

If completed successfully, the routine is required to return a name of at least length one, which is unique with respect to all other names of items for the same MPIT enumeration type.

1.2.6 Control Variables

The set of routines in this section of the MPIT interface specification focuses on the ability to list, query, and possibly set all exposed control variables used 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 many other configuration mechanisms might be used , like 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 message size that allows the transmission of messages using an eager protocol.

Control Variable Query Functions

Each MPI implementation exports a set of N control variables through MPIT. If N is zero, then the MPIT implementation does not export any control variables, otherwise the provided control variables are indexed from 0 to $N-1$. An MPIT implementation is allowed to increase the number of control variables during the execution of an MPI application, e.g., when new variables become available through dynamic loading. However, MPIT 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:

MPIT_CONTROLVAR_GET_NUM(num)

OUT	num	returns number of control variables

```
int MPIT_Controlvar_get_num (int *num)
```

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

MPIT_CONTROLVAR_GET_INFO(index, name, name_len, verbosity, datatype, count, desc, desc_len, assoc, attributes)

IN	index	index of the control variable to be queried
OUT	name	buffer to return 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	MPIT type of the information stored in the control variable
OUT	count	number of elements returned
OUT	desc	buffer to return a description of the control variable
INOUT	desc_len	length of the string and/or buffer for desc
OUT	assoc	type of MPI resource this variable is associated with
OUT	attributes	additional attributes defining this variable

```
int MPIT_Controlvar_g et_info(int index, char *name, int *name_len, int
             *verbosity, MPIT_Datatype *datatype, int *count, char *desc,
```

```
              int *desc_len,  int *assoc, MPIT_Controlvar_attributes
              *attributes)
```

After a successful call to MPIT_CONTROLVAR_GET_INFO for a particular variable, subsequent calls to this routine querying information about the same variable has to return the same information. An MPIT implementation is not allowed to alter it at runtime.

The arguments name and name_len are used to return the name of the control variable as described in Section 1.2.3.

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

The argument verbosity returns the verbosity level (see Section 1.2.1) assigned by the MPI implementation to the variable.

The argument datatype returns the MPIT datatype in which the value for this control variable will be returned. The value consists of count elements of this type.

The arguments desc and desc_len are used to return a description of the control variable as described in Section 1.2.3.

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

The parameter assoc returns the type of MPI resource the variable is associated with (see Section 1.2.2).

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

MPIT_CONTROLVAR_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 MPIT_Controlvar_attr_get_scope(MPIT_Controlvar_attributes *attributes,
              int *scope)
```

The scope of a variable determines whether it might be changeable through the MPIT interface and whether changing this variable is a local or a collective operation. On successful return from MPIT_CONTROLVAR_ATTR_GET_SCOPE, the argument scope will be set to one of the constants listed in Table 1.5.

Scope Constant	Description
MPIT_SCOPE_READONLY	read-only, cannot be written
MPIT_SCOPE_LOCAL	may be writeable, writing is a local operation
MPIT_SCOPE_GLOBAL	may be writeable, writing is a global operation

Table 1.5: Scopes for MPIT 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 MPIT implementation is allowed to return an error code as the result of the write operation. (*End of advice to users.*)

Handle Allocation and Deallocation

Before reading or writing the value of a variable, a user has to first allocate a handle for it by instantiating it with an instance of an MPI resource (see also Section 1.2.2). The type of the resource is returned by a previous call to MPIT_CONTROLVAR_GET_INFO.

MPIT_CONTROLVAR_HANDLE_ALLOCATE(index, resource, handle)

IN	index	index of control variable for which handle is to be allocated
IN	resource	reference to an MPI resource
OUT	handle	allocated handle

The reference to the resource instance passed through the argument resource can be generated by converting an MPI resource reference to a generic MPIT resource reference of type MPIT_MPI_Resource using the conversions functions described in Section 1.2.2).

```
int MPIT_Controlvar_handle_allocate(int index, MPIT_MPI_Resource resource,
                   MPIT_Controlvar_handle *handle)
```

A call to this routine , if successfully completed, allocates a handle for the control variable specified by the argument index and associates this variable with the instance of an MPI resource passed in the argument resource. The type of resource passed into this routine has to match the type of resources for this variable as returned by a prior call to MPIT_CONTROLVAR_GET_INFO.

MPIT_CONTROLVAR_HANDLE_FREE(handle)

INOUT	handle	handle to be freed

```
int MPIT_Controlvar_handle_free(MPIT_Controlvar_handle *handle)
```

If a handle is no longer needed, a user of MPIT should call MPIT_CONTROLVAR_HANDLE_FREE to free the handle and the associated resources.

Control Variable Access Functions

MPIT_CONTROLVAR_READ(handle, buf)

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

```
int MPIT_Controlvar_r ead(MPI_Controlvar_handle handle, void* buf)
```

The MPIT_CONTROLVAR_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 that the buffer is of the appropriate size and fits the entire value of the control variable (based on the returned type and count from a prior corresponding call to MPIT_CONTROLVAR_GET_INFO).

MPIT_CONTROLVAR_WRITE(handle, buf)

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

```
int MPIT_Controlvar_w rite(MPI_Controlvar_handle handle, void* buf)
```

The MPIT_CONTROLVAR_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 the entire value of the control variable (based on the returned type and count from a prior corresponding call to MPIT_CONTROLVAR_GET_INFO.)

If the variable has a global scope (as returned by a prior corresponding MPIT_CONTROLVAR_ATTR_GET_SCOPE call), any write call to this variable has to be issued on all connected 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 functions returns either MPIT_ERR_SETNOTNOW, if there may be a later time at which the variable could be set, or MPIT_ERR_SETNEVER, if the variable cannot be set for the remainder of the application's execution.

1.2.7 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 a component is in, aggregated timing data for submodules, or queue sizes and lengths.

Performance Variable Classes

Each reported performance variable is associated with a class of performance variables describing its the basic semantics. The class of a variable also defines its basic behavior, when and how an MPI implementation can change its value and what the initial or starting value of this variable is when it is either used for the first time or reset. Further, it also defines which types can be used to represent it. These classes are defined by the following constants:

- MPIT_PERFVAR_CLASS_STATE
  A performance variable in this class represents a set of discrete states the MPI implementation or a component of the MPI implementation is in. Variables of this class are expected to be represented by an enumeration type and can be set by the MPI implementation at any time. The default starting value is the current state of the implementation.

- MPIT_PERFVAR_CLASS_RESOURCE_LEVEL
  A performance variable in this class represents a value that describes the utilization level of a resource within the MPI implementation. 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 types: MPIT_BYTE, MPIT_SHORT, MPIT_INT, MPIT_LONG, MPIT_LONG_LONG, MPIT_FLOAT or MPIT_DOUBLE. The default starting value is the current utilization level of the resource.

- MPIT_PERFVAR_CLASS_RESOURCE_PERCENTAGE
  The value of a performance variable in this class represent the percentage utilization of a finite resource in the MPI implementation. The value of a variable of this class can change at any time to match the current utilization level of the resource. It should be returned as an MPIT_FLOAT or MPIT_DOUBLE type. The value has to be always between 0.0 (resource not used at all) and 1.0 (resource completely used). The default starting value is the current percentage utilization level of the resource.

- MPIT_PERFVAR_CLASS_RESOURCE_HIGHWATERMARK
  A performance variable in this class represents a value that describes the high watermark utilization of a resource within the MPI implementation. The value of a variable of this class is monotonically growing (from the initialization or reset of the variable). It can be represented by one of the following types: MPIT_BYTE, MPIT_SHORT, MPIT_INT, MPIT_LONG, MPIT_LONG_LONG, MPIT_FLOAT or MPIT_DOUBLE. The default starting value is the current utilization level of the resource.

- MPIT_PERFVAR_CLASS_RESOURCE_LOWWATERMARK
  A performance variable in this class represents a value that describes the low watermark utilization of a resource within the MPI implementation. The value of a variable of this class is monotonically decreasing (from the initialization or reset of the variable). It can be represented by one of the following types: MPIT_BYTE, MPIT_SHORT, MPIT_INT, MPIT_LONG, MPIT_LONG_LONG, MPIT_FLOAT or MPIT_DOUBLE. The default starting value is the current utilization level of the resource.

- MPIT_PERFVAR_CLASS_EVENT_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 is monotonically increasing (from the initialization or reset of the performance variable) by one for each specific event that is observed. Values have to be non-negative and represented by one of the following types: MPIT_SHORT, MPIT_INT, MPIT_LONG, MPIT_LONG_LONG. The default starting value for variables of this class is 0.

- MPIT_PERFVAR_CLASS_EVENT_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 is monotonically increasing (from

the initialization or reset of the performance variable). It has to be non-negative and represented by one of the following types: MPIT_SHORT, MPIT_INT, MPIT_LONG, MPIT_LONG_LONG, MPIT_FLOAT, MPI_DOUBLE. The default starting value for variables of this class is 0.

- MPIT_PERFVAR_CLASS_EVENT_TIMER
  The value of a performance variable in this class represents the aggregated time that the MPI implementation spends executing a particular event. This class has the same basic semantics as MPIT_PERFVAR_CLASS_EVENT_AGGREGATE, but explicitly records a timing value. The value of a variable of this class is monotonically increasing (from the initialization or reset of the performance variable). It has to be non-negative and represented by one of the following types: MPIT_INT, MPIT_LONG, MPIT_LONG_LONG, MPIT_FLOAT, MPIT_DOUBLE. The default starting value for variables if this class is 0.

Performance Variable Query Functions

Each MPI implementation exports a set of N performance variables through MPIT. If N is zero, then the MPIT implementation does not export any performance variables, otherwise the provided performance variables are indexed from 0 to $N-1$. An MPIT implementation is allowed to increase the number of performance variables during the execution of an MPI application, e.g., when new variables become available through dynamic loading. However, MPIT 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:

MPIT_PERFVAR_GET_NUM(num)

OUT	num	returns number of performance variables

`int MPIT_Perfvar_get_num(int *num)`

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

MPIT_PERFVAR_GET_INFO(index, name, name_len, verbosity, varclass, datatype, count, desc, desc_len, assoc, attributes)

IN	index	index of the performance variable to be queried
OUT	name	buffer to return 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	varclass	class of performance variable
OUT	datatype	MPIT type of the information stored in the performance variable
OUT	count	number of elements returned
OUT	desc	buffer to return a description of the performance variable
INOUT	desc_len	length of the string and/or buffer for desc
OUT	assoc	type of MPI resource this variable is associated with
OUT	attributes	additional attributes defining this variable

```
int MPIT_Perfvar_get_info(int num, char *name, int *name_len, int
             *verbosity, int *varclass, MPIT_Datatype *datatype, int
             *count, char *desc, int *desc_len, int *assoc,
             MPIT_Perfvar_attributes *attributes)
```

After a successful call to MPIT_PERFVAR_GET_INFO for a particular variable, subsequent calls to this routine querying information about the same variable has to return the same information. An MPIT implementation is not allowed to alter it at runtime.

The arguments name and name_len are used to return the name of the performance variable as described in Section 1.2.3.

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

The argument verbosity returns the verbosity level (see Section 1.2.1) assigned by the MPI implementation to the variable.

The class of the performance variable is returned in the parameter varclass and can be one of the constants defined in Section 1.2.7.

The argument datatype returns the MPIT datatype in which the value for this performance variable will be returned. The value consists of count elements of this type.

The arguments desc and desc_len are used to return a description of the control variable as described in Section 1.2.3.

Returning a description is optional. If an MPI implementation decides not to return a description, the first character for desc has to be set to the null character and desc_len has to be set to one at the return from this function.

The parameter assoc returns the type of MPI resource the variable is associated with (see Section 1.2.2).

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

MPIT_PERFVAR_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 MPIT_Perfvar_attr_get_readonly(MPIT_Perfvar_attributes *attributes, int
             *readonly)
```

Upon return, the argument readonly will be set to null if the variable can be written or reset by the user, or one if the variable is only initialized at MPIT_INIT and can only be read after that.

MPIT_PERFVAR_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 MPIT_Perfvar_attr_get_continuous(MPIT_Perfvar_attributes *attributes,
             int *continuous)
```

Upon return, the argument continuous will be set to null if the variable can be started and stopped by the user, or one if the variable is automatically active and can not by stopped by the user.

Performance Experiment Sessions

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

MPIT_PERFVAR_SESSION_CREATE(session)

OUT	session	identifier of performance experiment session

```
int MPIT_Perfvar_session_create(int *session)
```

This call creates a new session for accessing performance variables. An identifier of the current section is returned in session.

MPIT_PERFVAR_SESSION_FREE(session)

IN	session	identifier of performance experiment session

```
int MPIT_Perfvar_session_free(int session)
```

This call frees an existing session, i.e., calls to MPIT can no longer be made within the context of the freed session. This call also frees all handles that have been allocated within the specified session — see below for handle allocation and freeing.

Handle Allocation and Deallocation

Before using a performance variable, a user first has to allocate a handle for it by instantiating it with an instance of an MPI resource (see also Section 1.2.2). The type of the resource is returned by a previous call to MPIT_PERFVAR_GET_INFO.

MPIT_PERFVAR_HANDLE_ALLOCATE(session, index, resource, handle)

IN	session	identifier of performance experiment session
IN	index	index of performance variable for which handle is to be allocated
IN	resource	reference to an MPI resource
OUT	handle	allocated handle

```
int MPIT_Perfvar_handle_allocate(int session, int index, MPIT_MPI_Resource
             resource, MPIT_Perfvar_handle *handle)
```

A call to this routine, if successfully completed, allocates a handle for the performance variable specified by the argument index, associates this variable with the instance of an MPI resource passed in the argument resource, and resets the value of the variable to its default value (as specified in Section 1.2.7). The type of resource passed into this routine has to match the type of resources for this variable as returned by a prior call to MPIT_PERFVAR_GET_INFO.

The reference to the resource instance passed through the argument resource can be generated by converting an MPI resource reference to a generic MPIT resource reference of type MPIT_MPI_Resource using the conversions functions described in Section 1.2.2).

MPIT_PERFVAR_HANDLE_FREE(session,handle)

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

```
int MPIT_Perfvar_handle_free(int session, MPIT_Perfvar_handle *handle)
```

If a handle is no longer needed, a user of MPIT should call MPIT_PERFVAR_HANDLE_FREE to free the handle and the associated resources.

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 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, i.e., their values are not updated as the program executes, and have to be started by the user.

MPIT_PERFVAR_START(session, handle)

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

```
int MPIT_Perfvar_start(int session, MPIT_Perfvar_handle handle)
```

This functions starts the performance variable with the handle handle in the session session.

If the constant MPIT_PERFVAR_ALL_HANDLES is passed in handle, the MPI implementation attempts to start all variables within the session identified by session for which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all variables are started successfully, otherwise MPIT_ERR_NOSTARTSTOP is returned. Continuous variables and variables that are already started are ignored when used with MPIT_PERFVAR_ALL_HANDLES .

MPIT_PERFVAR_STOP(session, handle)

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

```
int MPIT_Perfvar_stop(int session, MPIT_Perfvar_handle handle)
```

This functions stops the performance variable with the handle handle in the session session.

If the constant MPIT_PERFVAR_ALL_HANDLES is passed in handle, the MPI implementation attempts to stop all variables within the session identified by session for which handles have been allocated. In this case, the routine returns MPI_SUCCESS if all variables are stopped successfully, otherwise MPIT_ERR_NOSTARTSTOP is returned. Continuous variables and variables that are already stopped are ignored when used with MPIT_PERFVAR_ALL_HANDLES .

Performance Variable Access Functions

MPIT_PERFVAR_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 MPIT_Perfvar_read(int session, MPIT_Perfvar_handle  handle, void* buf)
```

The MPIT_PERFVAR_READ call queries the value of the performance variable with the handle handle in the session 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 the entire value of the performance variable (based on the returned type and count during the MPIT_PERFVAR_GET_INFO call).

Note that the constant MPIT_PERFVAR_ALL_HANDLES can not be used as an argument for the MPIT function MPIT_PERFVAR_READ, since this would require the function to return a set of variable values instead of just one.

MPIT_PERFVAR_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 MPIT_Perfvar_write(int session, MPIT_Perfvar_handle handle, void* buf)
```

The MPIT_PERFVAR_WRITE call attempts to write the value of the performance variable with the handle handle in the session 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 the entire value of the performance variable (based on the returned type and count during the MPIT_PERFVAR_GET_INFO call).

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

Note that the constant MPIT_PERFVAR_ALL_HANDLES can not be used as an argument for the MPIT function MPIT_PERFVAR_WRITE, since this would require the function to accept a set of variable values instead of just one.

MPIT_PERFVAR_RESET(session, handle)

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

```
int MPIT_Perfvar_reset(int session, MPIT_Perfvar_handle  handle)
```

The MPIT_PERFVAR_RESET call sets of the performance variable with the handle handle to its default starting value (as specified in Section 1.2.7). If it is not possible to change the variable the function returns MPIT_ERR_PERFVAR_WRITE.

If the constant MPIT_PERFVAR_ALL_HANDLES is passed in handle, the MPI implementation attempts to reset all variables within the session identified by session for which handles have been allocated. In this case, the routine returns MPIT_SUCCESS if all variables are reset successfully, otherwise MPIT_ERR_NOWRITE is returned. Readonly variables are ignored when used with MPIT_PERFVAR_ALL_HANDLES .

MPIT_PERFVAR_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 MPIT_Perfvar_readreset(int session, MPIT_Perfvar_handle handle, void*
              buf)
```

The MPIT_PERFVAR_READRESET call atomically queries the value of the performance variable, stores the result in the buffer buf, and then sets the value of the performance variable to its default starting value (as specified in Section 1.2.7). The user is responsible to ensure that the buffer is of the appropriate size and fits the entire value of the performance variable (based on the returned type and count during the query call). If it is not possible to change the variable the function returns MPIT_ERR_PERFVAR_WRITE. In this case, the value returned in buf is the same as if the variable would have been read by the MPIT_PERFVAR_READ call.

Note that the constant MPIT_PERFVAR_ALL_HANDLES can not be used as an argument for the MPIT function MPIT_PERFVAR_READRESET, since this would require the function to return a set of variable values instead of just one.

> *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 attempt to document restrictions where deemed necessary. (*End of advice to implementors.*)

1.2.8 Variable Categorization

MPI implementations can optionally group performance and control variables into categories to express logical relationships between various variables. Categories can also contain other categories to form a hierarchical grouping. Categories can never include themselves either directly or transitively within other included categories.

> *Rationale.* The ability to include categories in other categories enables the creation of a hierarchical grouping of variables. The restriction that categories can not include

themselves directly or transitively guarantees that this structure is strictly hierarchical and does not contain any loops. (*End of rationale.*)

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.*)

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 MPIT interface. If $N = 0$, then the MPI implementation does not export any categories. An MPI implementation is permitted to increase the number of categories during the execution of an MPI program, such as 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:

MPIT_CATEGORY_GET_NUM(num)

OUT	num	current number of categories

```
int MPIT_Category_get_num(int *num)
```

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

MPIT_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, in the range $[0, N-1]$
OUT	name	buffer to return the name of the category
INOUT	name_len	length of the string and/or buffer for name
OUT	desc	buffer to return 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 MPIT categories contained in the category

```
int MPIT_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 1.2.3.

If completed successfully, the routine is required to return a name of at least length one, which is unique with respect to all other names for MPIT categories used by the MPIT implementation.

The arguments desc and desc_len are used to return the description of the category as described in Section 1.2.3.

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

On successful completion, the function returns the number of control variables (num_controlvars), performance variables (num_perfvars) and other categories (num_categories) contained in the queried category.

MPIT_CATEGORY_GET_CONTENTS(cat_index,len,kinds,indices)

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

```
int MPIT_Category_get_contents(int cat_index, int len, int kinds[],
             indices[])
```

MPIT_CATEGORY_GET_CONTENTS can be used to query which variables and other categories are contained in a particular category. A category may contain zero or more variables or categories and variables may be control variables, performance variables, or a mixture of the two kinds.

The index values returned in indices an be used as input to MPIT_CONTROLVAR_GET_INFO, MPIT_PERFVAR_GET_INFO or MPIT_CATEGORY_GET_INFO to get more information about the variables in the category designated by cat_index. The values in the kinds array indicate whether the corresponding entries in the indices array represent indices of a control variable, a performance variable, or another category. The constants used for this are listed in Table 1.6.

Constant	Variable or category type
MPIT_CATEGORY_CONTROLVAR	Element represents a control variable
MPIT_CATEGORY_PERFVAR	Element represents a performance variable
MPIT_CATEGORY_CATEGORY	Element represents another category

Table 1.6: Constants describing a variable or category type.

The user is responsible for allocating the arrays passed into the MPIT_CATEGORY_GET_CONTENTS function. The function will only write up to len elements into the arrays. If the category contains more than len variables and other categories the function returns an arbitrary subset; if it contains less then len variables and other categories, all will be returned and the remaining array entries will not be modified.

1.2.9 Return and Error Codes

All MPIT functions return a return or error code. The constants in Table 1.7 are defined for this purpose. None of the error codes returned by an MPIT routine shall be considered fatal to the overall MPI implementation or shall invoke an MPI error handler. In any case, the execution of the MPI program shall continue as if the call would have succeeded. However, the MPIT implementation is not required to check all user provided parameters; if a user passes illegal parameter values to any MPIT routine that are not caught by the implementation, the behavior of the implementation is undefined.

1.2.10 Profiling Interface

All requirements for the profiling interfaces, as described in Section **??**, also apply to the MPIT interface. In particular, this means that a complying MPI implementation has to provide matching PMPIT calls for every MPIT call. All rules, guidelines, and recommendations from Section **??** apply equally to PMPIT calls.

Return Code	Description
Return Codes for all MPIT Functions	
MPIT_SUCCESS	No error, call completed
MPIT_ERR_MEMORY	Out of memory
MPIT_ERR_NOTINITIALIZED	MPIT not initialized
MPIT_ERR_CANTINIT	MPIT not in the state to be initialized
Return Codes for Type Functions: MPIT_TYPE_*	
MPIT_ERR_PREDEFINED	Datatype is a predefined type and not an enumaration
MPIT_ERR_INVALIDTYPE	Datatype is not a valid datatype
MPIT_ERR_INVALIDITEM	The item index queried is out of range (for MPIT_TYPE_ENUMITEM only)
Return Codes for variable and category query functions: MPIT_*_GET_INFO	
MPIT_ERR_INVALIDINDEX	The variable or category index is invalid
Return Codes for Handle Functions: MPIT_*_ALLOCATE,FREE	
MPIT_ERR_INVALIDINDEX	The variable index is invalid
MPIT_ERR_INVALIDHANDLE	The handle is invalid
MPIT_ERR_OUTOFHANDLES	No more handles available
Return Codes for Session Functions: MPIT_PERFVAR_SESSION_*	
MPIT_ERR_OUTOFSESSIONS	No more sessions available
MPIT_ERR_INVALIDSESSION	Session argument is not a valid session
Return Codes for Control Variable Access Functions: MPIT_CONTROLVAR_READ,WRITE	
MPIT_ERR_SETNOTNOW	Variable cannot be set at this moment
MPIT_ERR_SETNEVER	Variable cannot be set until end of execution
MPIT_ERR_INVALIDVAR	Control variable does not exist
MPIT_ERR_INVALIDHANDLE	The handle is invalid
Return Codes for Performance Variable Access and Control: MPIT_PERFVAR_START,STOP,READ,WRITE,RESET,READRESET	
MPIT_ERR_INVALIDHANDLE	The handle is invalid
MPIT_ERR_INVALIDSESSION	Session argument is not a valid session
MPIT_ERR_NOSTARTSTOP	Variable can not be started or stopped for MPIT_PERFVAR_START and MPIT_PERFVAR_STOP
MPIT_ERR_NOWRITE	Variable can not be written or reset for MPIT_PERFVAR_WRITE and MPIT_PERFVAR_RESET
Return Codes for Category Functions: MPIT_CATEGORY_*	
MPIT_ERR_INVALIDCATEGORY	The specified category index does not exist

Table 1.7: Return and error codes used MPIT 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.

MPI Constant and Predefined Handle Index

This index lists predefined MPI constants and handles.

MPI Declarations Index

This index refers to declarations needed in C/C++, such as address kind integers, handles, etc. The underlined page numbers is the "main" reference (sometimes there are more than one when key concepts are discussed in multiple areas).

MPI Function Index

The underlined page numbers refer to the function definitions.