MPI: A Message-Passing Interface Standard

Version 3.0

⊤ (Fin2)
⊥ (Fin2)

Message Passing Interface Forum

Draft October 8, 2010

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. This includes a profiling interface (Section ??), PMPI, to transparently intercept and inspect any MPI call; and an information interface (Section 1.2), MPIT, 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. Additionally, several other tool interfaces exist that define interfaces that are primarily intended to be used from external processes. An example for the latter is the MPIR process acquisition interface, which is used by debuggers and performance analysis tools to detect and locate all MPI processes belonging to a given job. Currently, these interfaces are not included in MPI standard, but rather described in MPI forum white papers, which are published on the MPI forum's website.

1.2 MPIT Performance Interface

Open questions / ToDos:

- Versioning - should this be part of MPI or MPIT
- Change the get info calls to use structs
- String returns in Taxonomy section
- Iterators in Taxonomy section
- Adding Fortran interface

To optimize MPI applications or their runtime behavior, it is often advantageous to understand the performance switches an MPI library offers to the user as well as to monitor properties and timing information from within the MPI library. The MPIT interface described in this sections provides access to this information.

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 is particularly true for the initialization and finalization of MPIT, which is provided through a separate set of routines can be called before MPI_INIT and after MPI_FINALIZE .

All conventions and principles governing 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 (e.g., *mpi.h* where appropriate).

The interface is split into two parts: the first part provides information about control variables used by the MPI library to fine tune its performance. 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.

On success all MPIT routines return MPIT_SUCCESS, otherwise they return an appropriate error code. Details on error codes can be found in Section 1.2.7. 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 libraries, platforms, and even different builds of the same library 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 and avoid being dependent on the existence of a particular control or performance variable. (*End of advice to users.*)

1.2.1 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()
```

All programs or tools that use the MPIT interface must initialize the MPIT interface before calling any MPIT routine. The only exception to this rule is that the function MPIT_INITCOUNT can be called at any time.

A user can initialize the MPIT interface by calling MPIT_INIT, which can be called multiple times.

MPIT_FINALIZE( )

```
int MPIT_Finalize()
```

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_FINALLIZE 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_FINALLIZE that ends the initialization of MPIT may clean up all MPIT state and invalidate all open sessions (for the concept of Sessions see Section 1.2.5). MPIT can be reinitialized by subsequent calls to MPIT_INIT.

MPIT_INITCOUNT(num)

OUT	num	number of times MPIT is initialized

```
int MPIT_Initcount(int *num)
```

This routine returns the number of times MPIT_INIT has been called minus the times MPIT_FINALIZE has been called up to the current point of execution. It can be used to detect how many components in the current times MPIT has been initialized.

1.2.2 Type System

The MPIT interface provides its own type system. All types are represented by a variable or constant of type MPIT_Datatype. The Table 1.1 lists all available constants that can be used to identify a type for MPIT calls.

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 1.1: MPIT datatypes and their MPI equivalences.

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 variables of these types are equal and that it is possible to send and receive data of a particular MPIT type with regular MPI operations using the equivalent MPI type.

In addition to the predefined datatypes listed in the table, an MPI implementation may provide an additional set of enumeration datatypes to 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_ENUMQUERY.

MPIT_TYPE_ENUMQUERY(datatype,size,name,name_len)

IN	datatype	MPIT datatype to be queried
OUT	size	number of elements representable with this enumeration datatype
OUT	name	buffer to return the name of the type
INOUT	name_len	length of the string and/or buffer for name

```
int MPIT_Type_Enumquery(MPIT_Datatype datatype, int *size, 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 argument name provides a buffer to return the string describing the name of the type. The user has to pass the size of the buffer as the name_len argument. On return, the function deposits at most name_len-1 characters of the requested string into the buffer name followed by a terminating zero character. Additionally, the function writes the length of the returned string (including the terminating zero character) into name_len. If the returned value is smaller than the argument supplied to the function, the string has been truncated due to insufficient buffer resources. If the user passes NULL as the buffer argument or passes 0 as name_len, the function does not return the string and only returns the length of the string in name_len.

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

MPIT_TYPE_ENUMITEM(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_Enumitem(MPIT_Datatype datatype, int item, char *name, int
              *name_len)
```

The argument name provides a buffer to return the string describing the name of the enumeration item. The user has to pass the size of the buffer as the name_len argument. On return, the function deposits at most name_len-1 characters of the requested string into the buffer name followed by a terminating zero character. Additionally, the function writes the length of the returned string (including the terminating zero character) into name_len. If the returned value is smaller than the argument supplied to the function, the string has been truncated due to insufficient buffer resources. If the user passes NULL as the buffer argument or passes 0 as name_len, the function does not return the string and only returns the length of the string in name_len.

MPIT_TYPE_GETCLASS(datatype,typeclass)

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

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

This routine returns the class of the type for the datatype provided. This allows users of MPIT to distinguish whether a datatype used is an enumeration type or is one of the predefined types listed above. On return, the typeclass argument is set to one of the following constants, if datatype represents a valid type :

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

Table 1.2: MPIT type classes.

1.2.3 Verbosity Levels

The MPIT interface provides users access to internal performance data through a set of control and performance variables, which are defined by the MPI 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 1.3: MPIT verbosity levels.

MPI implementations using verbosity levels should first classify all variables according to the intended target audience (end user, performance optimization, or MPI developer) and then distinguish three level of verbosity (basic, detailed, and verbose) within each class.

Advice to implementors. If an MPIT does not distinguish between different verbosity levels, it is recommended to assign all variables to the level MPI_VERBOSITY_USER_BASIC. (*End of advice to implementors.*)

1.2.4 Control Variables

The first set of routines in the MPIT interface focuses on the ability to list, query, and possibly set all 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 library. On UNIX systems, such variables can often be set using environment variables, although many other configurations mechanisms might be used (e.g., configuration files, 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 MPI implementation does not export any control variables, otherwise the provided control variables are numbered from 1 to N. An MPI 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, MPI implementations are not allowed to change the number of a control variable or delete it once it has been added to the set.

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

MPIT_CTRLVAR_GETNUM(num)

OUT	num	returns number of control variables

```
int MPIT_CTRLVAR_Getum(int *num)
```

The name of individual variables (with numbers between 1 and N acquired by calling MPIT_CTRLVAR_GETNUM) can then be queried with the following function along with any associated information.

MPIT_CTRLVAR_GETINFO(num, name, name_len, verbosity, datatype, count, desc, desc_len, scope, comm)

IN	num	number 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	scope	scope of when changes to this variable are possible
OUT	comm	communicator that collective write operations to this variable have to be executed on

```
int MPIT_Ctrlvar_Getinfo(int num, char *name, int *name_len, int
             *verbosity, MPIT_Datatype *datatype, int *count, char *desc,
             int *desc_len, int *scope, MPI_Comm *comm)
```

The argument name provides a buffer to return the string describing the name of the control variable. The user has to pass the size of the buffer as the name_len argument. On return, the function deposits at most name_len-1 characters of the requested string into the buffer name followed by a terminating zero character. Additionally, the function writes the length of the returned string (including the terminating zero character) into name_len. If the returned value is smaller than the argument supplied to the function, the string has been truncated due to insufficient buffer resources. If the user passes NULL as the buffer argument or passes 0 as name_len, the function does not return the string and only returns the length of the string in name_len.

The argument verbosity returns the verbosity level (see Section 1.2.3) assigned by the MPI implementation to the variable.

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

The argument desc provides a buffer to return the string describing a description of the control variable. The user has to pass the size of the buffer as the desc_len argument. On return, the function deposits at most desc_len-1 characters of the requested string into the buffer desc followed by a terminating zero character. Additionally, the function writes the length of the returned string (including the terminating zero character) into desc_len. If the returned value is smaller than the argument supplied to the function, the string has been truncated due to insufficient buffer resources. If the user passes NULL as the buffer argument or passes 0 as desc_len, the function does not return the string and only returns the length of the string in desc_len.

Returning a description is optional. If an MPI library 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 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 return from MPIT_CTRLVAR_GETINFO it will be set to one of the constants listed in Table 1.4. If setting this variable requires a collective operation, the communicator on which this collective operation has to be executed, is returned as comm. If such an operation is not collective, the implementation should return MPI_COMM_SELF.

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

Table 1.4: Scopes for MPIT control variables.

Note that the scope of a variable only indicates when a variable might be changeable; it is not a guarantee that can be changed at any time. If it can not be changed at a time the user tries to write it, the MPIT implementation is allowed to return an error code as the result of the write operation.

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

Control Variable Access Functions

MPIT_CTRLVAR_READ(num, buf)

IN	num	number of control variable to be read
OUT	buf	initial address of storage location for variable value

```
int MPIT_Ctrlvar_Read(int num, void* buf)
```

The MPIT_CTRLVAR_READ queries the value of the control variable with the number num 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 during the MPIT_CTRLVAR_GETINFO call) .

MPIT_CTRLVAR_WRITE(num, buf, comm)

IN	num	number of control variable to be read
IN	buf	initial address of storage location for variable value
IN	comm	communicator for which this operation is collective on

```
int MPIT_Ctrlvar_Write(int num, void* buf, MPI_Comm comm)
```

The MPIT_CTRLVAR_WRITE sets the value of the control variable with the number num 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 during the query MPIT_CTRLVAR_GETINFO call).

The operation is collective with respect to the communicator comm. The user is responsible that the right communicator, i.e., the one returned by MPIT_CTRLVAR_GETINFO, is passed as the comm argument and that this operation is called as a collective operation on all processes in the communicator. The same ordering constraints as for MPI collectives apply. If this operation is local and not collective, the user is required to pass MPI_COMM_SELF.

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 time.

1.2.5 Performance Variables

The second set of functions included in the MPIT interface 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 like 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, which describes its the basic semantics. 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 library or a component of the MPI library is in. The value of this kind of variable can change at any time to any value within the type definition. Variables of this class are expected to be represented by an enumeration type. Variables of this class don't have a default starting value, since the variable reflects a current state of the library.

- MPIT_PERFVAR_CLASS_UTILIZATION
  The value of a performance variable in this class represent the percentage utilization of a finite resource in the MPI library. The value of this kind of variable can change at any time and should be returned as an MPIT_FLOAT or MPIT_DOUBLE type. The value must always be between 0.0 (resource not used at all) and 1.0 (resource completely used). Variables of this class don't have a default starting value, since the variable reflects a current state of the library.

- MPIT_PERFVAR_CLASS_RESOURCE
  A performance variable in this class represents a value that describes the absolute utilization level of a resource within the MPI library. The value of this kind of variable can change at any time and values returned from variables in this class must be non-negative and are represented by one of the following types: MPIT_BYTE, MPIT_SHORT, MPIT_INT, MPIT_LONG, MPIT_LONG_LONG, MPIT_FLOAT or MPIT_DOUBLE. Variables of this class don't have a default starting value, since the variable reflects a current state of the library.

- MPIT_PERFVAR_CLASS_HIGHWATERMARK
A performance variable in this class represents a value that describes the high watermark absolute utilization of a resource within the MPI library. The value of this kind of variable is monotonically growing (from the initialization or reset of the variable). It must be non-negative and 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 for variables of this class is the current absolute utilization of the resource.

- MPIT_PERFVAR_CLASS_LOWWATERMARK
A performance variable in this class represents a value that describes the low watermark absolute utilization of a resource within the MPI library. The value of this kind of variable is monotonically shrinking (from the initialization or reset of the variable). It must be non-negative and 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 for variables of this class is the current absolute utilization of the resource.

- MPIT_PERFVAR_CLASS_COUNTER
A performance variable in this class counts the number of occurrences of a specific event during the execution time of an application. The value of this kind of variable is monotonically increasing (from the initialization or reset of the performance variable). It must 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_AGGREGATE
The value of a performance variable in this class is an an aggregated value of over time. 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 this kind of variable is monotonically increasing (from the initialization or reset of the performance variable). It must 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_TIMER
The value of a performance variable in this class represents the aggregated time that the MPI library spends executing a particular event. The value of this kind of variable is monotonically increasing (from the initialization or reset of the performance variable). It must 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 MPI implementation does not export any performance variables, otherwise the provided performance variables are numbered from 1 to N. An MPI 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, MPI implementations are not allowed to change the number of a performance variable or delete it once it has been added to the set.

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

MPIT_PERFVAR_GETNUM(num)

OUT	num	returns number of performance variables

```
int MPIT_PERFVAR_Getum(int *num)
```

The name of individual variables (with numbers between 1 and N acquired by calling MPIT_PERFVAR_GETNUM) can then be queried with the following function along with any associated information.

MPIT_PERFVAR_GETINFO(num, name, name_len, verbosity, varclass, datatype, count, desc, desc_len, readonly, continuous)

IN	num	number 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 control variable
INOUT	desc_len	length of the string and/or buffer for desc
OUT	readonly	flags indicating whether variable can be written/reset
OUT	continuous	flags indicating whether variable can be started/stopped or is continuously activated

```
int MPIT_Perfvar_Getinfo(int num, char *name, int *name_len, int
             *verbosity, int *varclass, MPIT_Datatype *datatype, int
             *count, char *desc, int *desc_len, int *readonly, int
             *continuous)
```

The argument name provides a buffer to return the string describing the name of the control variable. The user has to pass the size of the buffer as the name_len argument. On return, the function deposits at most name_len-1 characters of the requested string into the buffer name followed by a terminating zero character. Additionally, the function writes the length of the returned string (including the terminating zero character) into name_len. If the returned value is smaller than the argument supplied to the function, the string has been truncated due to insufficient buffer resources. If the user passes NULL as the buffer argument or passes 0 as name_len, the function does not return the string and only returns the length of the string in name_len.

The argument verbosity returns the verbosity level (see Section 1.2.3) 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 1.2.5.

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

The argument desc provides a buffer to return the string describing a description of the control variable. The user has to pass the size of the buffer as the desc_len argument. On return, the function deposits at most desc_len-1 characters of the requested string into the buffer desc followed by a terminating zero character. Additionally, the function writes the length of the returned string (including the terminating zero character) into desc_len. If the returned value is smaller than the argument supplied to the function, the string has been truncated due to insufficient buffer resources. If the user passes NULL as the buffer argument or passes 0 as desc_len, the function does not return the string and only returns the length of the string in desc_len.

Returning a description is optional. If an MPI library 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.

Upon return, the argument readonly will be set to zero 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.

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

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

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 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 shall not influence the results in any other session.

MPIT_PERFVAR_SESSIONCREATE(session)

OUT	session	identifier of performance experiment session

```
int MPIT_Perfvar_Sessioncreate(int *session)
```

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

MPIT_PERFVAR_SESSIONFREE(session)

IN	session	identifier of performance experiment session

```
int MPIT_Perfvar_Sessionfree(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. After the call, all active performance variables in this context are deactivated.

Performance Variable Activation

Before a performance variable can be used, i.e., started, stopped, read, written, or reset, it must first be activated. Only activated performance variables can be passed to the start, stop or access functions discussed in the next sections.

MPIT_PERFVAR_ACTIVATE(session,num)

IN	session	identifier of performance experiment session
IN	num	number of the performance variable

```
int MPIT_Perfvar_Activate(int session, int num)
```

This routine activates the performance variable num with respect to session session. If this variable is not yet activated, the variable will be reset to its default value. Calling this function on already activated variables (within the same session) has no affect.

MPIT_PERFVAR_DEACTIVATE(session,num)

IN	session	identifier of performance experiment session
IN	num	number of the performance variable

```
int MPIT_Perfvar_Deactivate(int session, int num)
```

This routine deactivates the performance variable num with respect to session session.

> *Advice to implementors.* The extra step of activating performance variables allows MPIT implementations to selectively enable counters and only monitor activated events. This can be used to minimize the overhead of performance monitors when not used. (*End of advice to implementors.*)

Starting and Stopping of Performance Variables

Performance variables that have the continuous flag set during the query operation are continuously operating after a call to MPIT_PERFVAR_ACTIVATE and can not be stopped or paused by the user. All other variables are in a stopped state after their first activation within a session, i.e., they are not updated as the program executes, and have to be started by the user.

MPIT_PERFVAR_START(session,num)

IN	session	Identifier of performance experiment session
IN	num	number of the performance variable

```
int MPIT_Perfvar_Start(int session, int num)
```

This functions starts the performance variable with the number num in the session session. The variable has to be activated before making this call using the function MPIT_PERFVAR_ACTIVATE.

If the constant MPIT_PERFVAR_ALL is passed in num, the MPI library attempts to start all activated variables within the session identified by session. In this case, the routine returns MPI_SUCCESS if all variables are started successfully; continuous variables, variables that are already started, and not activated variables are ignored when used with MPIT_PERFVAR_ALL .

MPIT_PERFVAR_STOP(session, num)

IN	session	Identifier of performance experiment session
IN	num	number of the performance variable

```
int MPIT_Perfvar_Stop(int session, int num)
```

This functions stops the performance variable with the number num in the session session. The variable has to be activated before making this call using the function MPIT_PERFVAR_ACTIVATE.

If the constant MPIT_PERFVAR_ALL is passed passed in num, the MPI library attempts to stop all activated variables within the session identified by session. In this case, the routine returns MPI_SUCCESS if all variables are stopped successfully; continuous variables, variables that are already stopped, and not activated variables are ignored when used with MPIT_PERFVAR_ALL .

Advice to implementors. Although MPI places no requirements on the interaction with external mechanisms such as signal handlers, it is strongly recommended that the routines in this section to start and stop 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.*)

Performance Variable Access Functions

MPIT_PERFVAR_READ(session, num, buf)

IN	session	Identifier of performance experiment session
IN	num	number of the performance variable
OUT	buf	initial address of storage location for variable value

```
int MPIT_Perfvar_Read(int session, int num, void* buf)
```

The MPIT_PERFVAR_READ call queries the value of the performance variable with the number num 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_GETINFO call). The variable has to be activated before making this call using the function MPIT_PERFVAR_ACTIVATE.

MPIT_PERFVAR_WRITE(session, num, buf)

IN	session	Identifier of performance experiment session
IN	num	number of the performance variable
IN	buf	initial address of storage location for variable value

```
int MPIT_Perfvar_write(int session, int num, void* buf)
```

The MPIT_PERFVAR_WRITE call attempts to write the value of the performance variable with the number num 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_GETINFO call). The variable has to be activated before making this call using the function MPIT_PERFVAR_ACTIVATE.

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

MPIT_PERFVAR_RESET(session,num)

IN	session	Identifier of performance experiment session
IN	num	number of the performance variable

```
int MPIT_Perfvar_reset(int session, int num)
```

The MPIT_PERFVAR_RESET call sets the value of the performance variable to its default starting value. If it is not possible to change the variable the function returns MPIT_ERR_PERFVAR_WRITE.

If the constant MPIT_PERFVAR_ALL is passed in num, the MPI library attempts to reset all activated variables within the session identified by session. In this case, the routine returns MPIT_SUCCESS if all variables are reset successfully; readonly variables, and not activated variables are ignored when used with MPIT_PERFVAR_ALL .

MPIT_PERFVAR_READRESET(session,num, buf)

IN	session	Identifier of performance experiment session
IN	num	number of the performance variable
OUT	buf	initial address of storage location for variable value

```
int MPIT_Perfvar_Readreset(int session, int num, 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. 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.

Advice to implementors. Although MPI places no requirements on the interaction with external mechanisms such as signal handlers, it is strongly recommended that the routines in this section to 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.*)

1.2.6 Performance and Control Variable Taxonomic Information

MPI implementations can optionally provide information that describes the relationship of performance and control variables to each other. For this, an MPI implementation can define names that represent sets of variables and then associate each performance/control variable with zero or more sets. Sets may contain zero or more performance/control variables and zero or more other sets. Sets may not contain themselves either directly or indirectly. More formally, these sets and performance/control variables form a directed acyclic graph (DAG). This information is accessible via several interrogative routines.

MPIT_TAXON_QUERY_SET_SETS(iterator, setname, name, namelen, type)

INOUT	iterator	iterator variable passed in by user (iterator)
IN	setname	name of the set to be queried (string)
OUT	name	name of the set returned on this iteration (string)
OUT	namelen	length of the name of the set returned on this iteration (string)
OUT	type	type of the set returned this iteration (integer)

```
int MPIT_Taxon_query_set_sets(MPIT_Taxonquery_iterator *iterator, char
            *setname, char *name, int *namelen, int *type);
```

Iterate over all sets contained in the set identified by setname. A unique identifying name for the contained set is returned in name and namelen is set to the number of characters written. The value of namelen cannot be larger than MPIT_MAX_SET_NAME-1. The set of all root sets (sets that no other set contains) available in the implementation can be queried by using a setname of MPIT_ROOT_SETS. The type parameter is set to MPIT_TAXON_CTRLVAR if the variable is a control variable and to MPIT_TAXON_PERFVAR if it is a performance variable.

On the first call to MPIT_TAXON_QUERY_SET_SETS, the caller must initialize a variable to MPIT_TAXON_QUERY_START and pass this variable as the iter parameter. Subsequent calls require the user to pass the returned value iter to query further taxonomic information. Once all taxonomic information is returned, the call to MPIT_TAXON_QUERY_SET_SETS returns MPIT_END and sets iter to MPIT_TAXON_QUERY_END.

MPIT_TAXON_QUERY_VARIABLE_SETS(iterator, varname, name, namelen, type)

INOUT	iterator	iterator variable passed in by user (iterator)
IN	varname	name of the variable to be queried (string)
OUT	name	name of the set returned this iteration (string)
OUT	namelen	length of the name of the set returned this iteration (integer)
OUT	type	type of the set returned this iteration (integer)

```
int MPIT_Taxon_query_variable_sets(MPIT_Taxonquery_iterator *iterator, char
            *varname, char *name, int *namelen, int *type);
```

Iterate over all sets that contain the performance/control variable identified by varname. A unique identifying name for the set is returned in name and namelen is set to the number of characters written. The value of namelen cannot be larger than MPIT_MAX_SET_NAME-1. The type parameter is set to MPIT_TYPE_CTRLVAR if the variable is a control variable and to MPIT_TYPE_PERFVAR if it is a performance variable.

On the first call to MPIT_TAXON_QUERY_VARIABLE_SETS, the caller must initialize a variable to MPIT_TAXON_QUERY_START and pass this variable as the iter parameter. Subsequent calls require the user to pass the returned value iter to query further taxonomic information. Once all taxonomic information is returned, the call to MPIT_TAXON_QUERY_VARIABLE_SETS returns MPIT_END and sets iter to MPIT_TAXON_QUERY_END.

MPIT_TAXON_QUERY_SET_VARIABLES(iterator, setname, name, namelen, type)

INOUT	iterator	iterator variable passed in by user (iterator)
IN	setname	name of the set to be queried (string)
OUT	name	name of the variable returned this iteration (string)
OUT	namelen	length of the name of the variable returned this iteration (integer)
OUT	type	type of the variable returned this iteration (integer)

```
int MPIT_Taxon_query_set_variables(MPIT_Taxonquery_iterator *iterator, char
              *setname, char *name, int *namelen, int *type);
```

Iterate over all variables directly contained in the set identified by setname. That is, variables contained indirectly by a contained set will not be returned by this call. A unique identifying name for the variable is returned in name and namelen is set to the number of characters written. The value of namelen cannot be larger than MPIT_MAX_SET_NAME-1. The type parameter is set to MPIT_TYPE_CTRLVAR if the variable is a control variable and to MPIT_TYPE_PERFVAR if it is a performance variable.

On the first call to MPIT_TAXON_QUERY_SET_VARIABLES, the caller must initialize a variable to MPIT_TAXON_QUERY_START and pass this variable as the iter parameter. Subsequent calls require the user to pass the returned value iter to query further taxonomic information. Once all taxonomic information is returned, the call to MPIT_TAXON_QUERY_SET_VARIABLES returns MPIT_END and sets iter to MPIT_TAXONQUERY_END.

MPIT_TAXON_CHANGED(flag)

OUT	flag	true if the taxonomic information has changed since the last call to a query function (boolean)

```
int MPIT_Taxon_changed(int *flag);
```

This routine returns true in the flag argument if the list of available performance/control variables or sets has changed since the last time the user has called any of the MPIT_TAXON_QUERY_ routines with the argument MPIT_TAXON_QUERY_START as the first argument. If the user has not yet called any such routines, the argument flag will contain the value true.

Return Code	Description
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

Table 1.5: Return codes used by any MPIT function.

MPIT_TAXON_DESCRIBE_SET(name, desc, desclen)

IN	name	name of the set to describe (string)
OUT	desc	description of the set (string)
OUT	desclen	length of the string returned in desc (int)

```
int MPIT_Taxon_describe_set(char *name, char *desc, int desclen);
```

Retrieve the description for the set identified by name and store it in desc. The desclen parameter is set to the number of characters written. The value of desclen cannot be larger than MPIT_MAX_SET_DESC-1.

Set and Variable Names MPI does not specify the character encoding of strings in the MPIT interface. The only requirement is that strings are terminated with a null character.

MPI reserves all set and variable names with the prefixes "MPI_" and "MPIT_" for its own use.

1.2.7 Return and Error Codes

All MPIT functions return a return or error code. The following constants are available for this for the specific calls. 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 library is undefined.

Return Codes for all MPIT Functions

The return codes in Table 1.5 apply to all MPIT functions.

Return Codes for Type Functions

The return codes in Table 1.6 apply to MPIT_TYPE_ENUMQUERY, MPIT_TYPE_GETCLASS and MPIT_TYPE_ENUMITEM.

Return Codes for Control Variable Access Functions

The return codes in Table 1.7 apply to MPIT_CONFIG_READ and MPIT_CONFIG_WRITE.

Return Code	Description
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 number queried is out of range (for MPIT_TYPE_ENUMITEM only)

Table 1.6: Return codes used by MPIT type functions.

Return Code	Description
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

Table 1.7: Return codes used by MPIT control variable access functions.

Return Code	Description
MPIT_ERR_INVALIDVAR	Performance variable does not exist
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

Table 1.8: Return codes used by MPIT performance variable access, start, stop, or activation functions.

Return Code	Description
MPIT_ERR_NOSET	The set does not exist
MPIT_ERR_NODATA	No description for this set available

Table 1.9: Return codes used MPIT taxonomy functions.

Return Codes for Performance Variable Access and Control

The return codes in Table 1.8 apply to MPIT_PERFVAR_START , MPIT_ERR_PERFVAR_STOP , MPIT_PERFVAR_READ , MPIT_ERR_PERFVAR_WRITE , MPIT_PERFVAR_RESET , and MPIT_PERFVAR_READRESET .

Return Codes for Taxonomy Functions

The return codes in Table 1.9 apply to MPIT_TAXON_ routines.

1.2.8 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.

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 Function Index

The underlined page numbers refer to the function definitions.