This chapter discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). The procedures for entering and leaving the MPI execution environment are also described here.

### 7.1 Implementation information

#### 7.1.1 ENVIRONMENTAL INQUIRIES

A set of attributes that describe the execution environment are attached to the communicator MPI_COMM_WORLD when MPI is initialized. The value of these attributes can be inquired by using the function MPI_ATTR_GET described in Chapter 5. It is erroneous to delete these attributes or free their keys.

The list of predefined attribute keys include:

- **MPI.TAG.UB** Upper bound for tag value.
- **MPI.HOST** Host process rank, if such exists, MPI_PROC_NULL, otherwise.
- **MPI.IO** Rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.

Vendors may add implementation-specific parameters (such as node number, real memory size, virtual memory size, etc.).

The required parameter values are discussed in more detail below.

### Tag values

Tag values range from 0 to the value returned for MPI.TAG.UB inclusive. These values are guaranteed to be unchanging during the execution of an MPI program. In addition, the tag upper bound value must be at least 32767. An MPI implementation is free to make the value of MPI.TAG.UB larger than this; for example, the value $2^{30} - 1$ is also a legal value for MPI.TAG.UB.
Host rank

The value returned for MPI.HOST gets the rank of the HOST process in the group associated with communicator MPI.COMM_WORLD, if there is such. MPI.PROC.NULL is returned if there is no host. MPI does not specify what it means for a process to be a HOST, nor does it require that a HOST exists.

IO rank

The value returned for MPI.IO is the rank of a processor that can provide language-standard I/O facilities. For Fortran, this means that all of the Fortran I/O operations are supported (e.g., OPEN, REWIND, WRITE). For C, this means that all of the ANSI-C I/O operations are supported (e.g., fopen, fprintf, fseek).

If every process can provide language-standard I/O, then the value MPI.ANY_SOURCE must be returned. If no process can provide language-standard I/O, then the value MPI.PROC_NULL must be returned. If several processes can provide I/O, then any of them may be returned. The same value (rank) need not be returned by all processes.

MPI.GET_PROCESSOR_NAME( name, resultlen )

OUT name A unique specifier for the actual (as opposed to virtual) node.

OUT resultlen Length (in printable characters) of the result returned in name

int MPI.Get_processor_name(char *name, int *resultlen)

MPI.GET_PROCESSOR_NAME( NAME, RESULTLEN, IERROR)

CHARACTER*(*) NAME

INTEGER RESULTLEN, IERROR

This routine returns the name of the processor on which it was called at the moment of the call. The name is a character string for maximum flexibility. From this value it must be possible to identify a specific piece of hardware; possible values include “processor 9 in rack 4 of mpp.cs.org” and “231” (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least MPI.MAX_PROCESSOR_NAME characters long. MPI.GET_PROCESSOR_NAME may write up to this many characters into name.

The number of characters actually written is returned in the output argument, resultlen.

Rationale. This function allows MPI implementations that do process migration to return the current processor. Note that nothing in MPI requires or defines process migration; this definition of MPI.GET_PROCESSOR_NAME simply allows such an implementation. (End of rationale.)
7.2 Error Handling

An MPI implementation cannot or may choose not to handle some errors that occur during MPI calls. These can include errors that generate exceptions or traps, such as floating point errors or access violations. The set of errors that are handled by MPI is implementation-dependent. Each such error generates an MPI exception.

A user can associate an error handler with a communicator. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for a communication with this communicator. MPI calls that are not related to any communicator are considered to be attached to the communicator MPI.COMM.WORLD. The attachment of error handlers to communicators is purely local: different processes may attach different error handlers to the same communicator.

A newly created communicator inherits the error handler that is associated with the “parent” communicator. In particular, the user can specify a “global” error handler for all communicators by associating this handler with the communicator MPI.COMM.WORLD immediately after initialization.

Several predefined error handlers are available in MPI:

MPI_ERRORS_ARE_FATAL The handler, when called, causes the program to abort on all executing processes. This has the same effect as if MPI.ABORT was called by the process that invoked the handler.

MPI_ERRORS_RETURN The handler has no effect.

Implementations may provide additional predefined error handlers and programmers can code their own error handlers.

The error handler MPI_ERRORS_ARE_FATAL is associated by default with MPI.COMM.WORLD after initialization. Thus, if the user chooses not to control error handling, every error that MPI handles is treated as fatal. Since (almost) all MPI calls return an error code, a user may choose to handle errors in its main code, by testing the return code of MPI calls and executing a suitable recovery code when the call was not successful. In this case, the error handler MPI_ERRORS_RETURN will be used. Usually it is more convenient and more efficient not to test for errors after each MPI call, and have such error handled by a nontrivial MPI error handler.

After an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or MPI_ERRORS_RETURN, does not necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits.
An MPI implementation is free to allow MPI to continue after an error but is not required to do so.

Advice to implementors. A good-quality implementation will, to the greatest possible extent, circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked. The implementation documentation will provide information on the possible effect of each class of errors. (End of advice to implementors.)

An MPI error handler is an opaque object, which is accessed by a handle. MPI calls are provided to create new error handlers, to associate error handlers with communicators, and to test which error handler is associated with a communicator.

MPI_ERRHANDLER_CREATE( function, errhandler )

<table>
<thead>
<tr>
<th>IN</th>
<th>OUT</th>
<th>user-defined error handling procedure</th>
<th>MPI error handler (handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>errhandler</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

int MPI_Errhandler.create(MPI_Handler_function *function,
                           MPI_Errhandler *errhandler)

MPI_ERRHANDLER_CREATE(FUNCTION, HANDLER, IERROR)

EXTERNAL FUNCTION

INTEGER ERRHANDLER, IERROR

Register the user routine function for use as an MPI exception handler. Returns in errhandler a handle to the registered exception handler.

Advice to implementors. The handle returned may contain the address of the error handling routine. This call is superfluous in C, which has a referencing operator, but is necessary in Fortran. (End of advice to implementors.)

The user routine should be a C function of type MPI_Handler_function, which is defined as:

typedef void (MPI_Handler_function)(MPI_Comm *, int *, ...);

The first argument is the communicator in use. The second is the error code to be returned by the MPI routine. The remaining arguments are "stdargs" arguments whose number and meaning is implementation-dependent. An implementation should clearly document these arguments. Addresses are used so that the handler may be written in Fortran.

Rationale. The variable argument list is provided because it provides an ANSI-standard hook for providing additional information to the error handler; without this hook, ANSI-C prohibits additional arguments. (End of rationale.)
MPI_ERRHANDLER_SET (comm, errhandler)

IN     comm    communicator to set the error handler for
         (handle)
IN     errhandler    new MPI error handler for communicator
                     (handle)

int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)

MPI_ERRHANDLER_SET (COMM, ERRHANDLER, IERROR)

INTEGER COMM, ERRHANDLER, IERROR

Associates the new error handler errhandler with communicator comm
at the calling process. Note that an error handler is always associated
with the communicator.

MPI_ERRHANDLER_GET (comm, errhandler)

IN     comm    communicator to get the error handler from
         (handle)
OUT    errhandler    MPI error handler currently associated with
                     communicator (handle)

int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)

MPI_ERRHANDLER_GET (COMM, ERRHANDLER, IERROR)

INTEGER COMM, ERRHANDLER, IERROR

Returns in errhandler (a handle to) the error handler that is currently
associated with communicator comm.

Example: A library function may register at its entry point the current
error handler for a communicator, set its own private error handler for this
communicator, and restore before exiting the previous error handler.

MPI_ERRHANDLER_FREE (errhandler)

IN     errhandler    MPI error handler (handle)

int MPI_Errhandler_free(MPI_Errhandler *errhandler)

MPI_ERRHANDLER_FREE (ERRHANDLER, IERROR)

INTEGER ERRHANDLER, IERROR

Marks the error handler associated with errhandler for deallocation and sets
errhandler to MPI_ERRHANDLER_NULL. The error handler will be deallocated after
all communicators associated with it have been deallocated.

MPI_ERROR_STRING (errorcode, string, resultlen)

IN     errorcode    Error code returned by an MPI routine
OUT    string    Text that corresponds to the errorcode
int MPI_Error_string(int errorcode, char *string, int *resultlen)

MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
INTEGER ERRORCODE, RESULTLEN, IERROR
CHARACTER*(*) STRING

Returns the error string associated with an error code. The argument string must represent storage that is at least MPI_MAX_ERROR_STRING characters long. The number of characters actually written is returned in the output argument, resultlen.

Rationale. The form of this function was chosen to make the Fortran and C bindings similar. A version that returns a pointer to a string has two difficulties. First, the return string must be statically allocated and different for each error message (allowing the pointers returned by successive calls to MPI_ERROR_STRING to point to the correct message). Second, in Fortran, a function declared as returning CHARACTER*(*) cannot be referenced in, for example, a PRINT statement. (End of rationale.)

7.3 Error Codes and Classes

The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING).

To make it possible for an application to interpret an error code, the routine MPI_ERROR_CLASS converts an error code into one of a small set of specified values, called error classes. Valid error classes include

<table>
<thead>
<tr>
<th>MPI_SUCCESS</th>
<th>No error</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_ERR_BUFFER</td>
<td>Invalid buffer pointer</td>
</tr>
<tr>
<td>MPI_ERR_COUNT</td>
<td>Invalid count argument</td>
</tr>
<tr>
<td>MPI_ERR_TYPE</td>
<td>Invalid datatype argument</td>
</tr>
<tr>
<td>MPI_ERR_TAG</td>
<td>Invalid tag argument</td>
</tr>
<tr>
<td>MPI_ERR_COMM</td>
<td>Invalid communicator</td>
</tr>
<tr>
<td>MPI_ERR_RANK</td>
<td>Invalid rank</td>
</tr>
<tr>
<td>MPI_ERR_REQUEST</td>
<td>Invalid request (handle)</td>
</tr>
<tr>
<td>MPI_ERR_ROOT</td>
<td>Invalid root</td>
</tr>
<tr>
<td>MPI_ERR_GROUP</td>
<td>Invalid group</td>
</tr>
<tr>
<td>MPI_ERR_OP</td>
<td>Invalid operation</td>
</tr>
<tr>
<td>MPI_ERR_TOPOLOGY</td>
<td>Invalid topology</td>
</tr>
<tr>
<td>MPI_ERR_DIMS</td>
<td>Invalid dimension argument</td>
</tr>
</tbody>
</table>
An implementation is free to define more error classes; however, the standard error classes must be used where appropriate. The error classes satisfy,

\[ 0 = \text{MPI.SUCCESS} < \text{MPI.ERR.\ldots} < \text{MPI.ERR.LASTCODE}. \]

*Rationale.* The difference between MPI.ERR.UNKNOW and MPI.ERR.OTHER is that MPI.ERROR.STRING can return useful information about MPI.ERR.OTHER.

Note that MPI.SUCCESS = 0 is necessary to be consistent with C practice; the separation of error classes and error codes allows us to define the error classes this way. Having a known LASTCODE is often a nice sanity check as well. *(End of rationale.)*

**MPI.ERROR.CLASS(errorcode, errorclass)**

<table>
<thead>
<tr>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorcode</td>
<td>errorclass</td>
</tr>
</tbody>
</table>

Error code returned by an MPI routine

Error class associated with errorcode

```c
int MPI_Error_class(int errorcode, int *errorclass)
```

**MPI.ERROR.CLASS(ERROCODE, ERRORCLASS, IERROR)**

```c
INTEGER ERROCODE, ERRORCLASS, IERROR
```

### 7.4 Timers

MPI defines a timer. A timer is specified even though it is not “message-passing,” because timing parallel programs is important in “performance debugging” and because existing timers (both in POSIX 1003.1-1988 and 1003.4D 14.1 and in Fortran 90) are either inconvenient or do not provide adequate access to high-resolution timers.

**MPI.WTIME()**

```c
double MPI_Wtime(void)
```

**DOUBLE PRECISION MPI.WTIME()**

`MPI.WTIME` returns a floating point number of seconds, representing elapsed wall-clock time since some time in the past.

The “time in the past” is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.
This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```c
{
    double starttime, endtime;
    starttime = double MPI_Wtime();
    .... stuff to be timed ....
    endtime = double MPI_Wtime();
    printf("That took %f seconds\n", endtime-starttime);
}
```

The times returned are local to the node that called them. There is no requirement that different nodes return "the same time."

**MPI.WTICK()**

double MPI_Wtick(void)

**DOUBLE PRECISION MPI.WTICK()**

**MPI.WTICK** returns the resolution of **MPI.WTIME** in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by **MPI.WTICK** should be \(10^{-3}\).

### 7.5 Startup

One goal of **MPI** is to achieve _source code portability_. By this we mean that a program written using **MPI** and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does _not_ say anything about how an **MPI** program is started or launched from the command line, nor what the user must do to set up the environment in which an **MPI** program will run. However, an implementation may require some setup to be performed before other **MPI** routines may be called. To provide for this, **MPI** includes an initialization routine **MPI.INIT**.

**MPI.INIT()**

```c
int MPI_Init(int *argc, char **argv)
```

**MPI.INIT(IERROR)**

```c
INTEGER IERROR
```

This routine must be called before any other **MPI** routine. It must be called at most once; subsequent calls are erroneous (see **MPI.INITIALIZED**).

All **MPI** programs must contain a call to **MPI.init**; this routine must be called
before any other MPI routine (apart from MPI_INITIALIZED) is called. The version
for ANSI-C accepts the argc and argv that are provided by the arguments to main:

MPI_init( argc, argv );

The Fortran version takes only IERROR.

MPI_FINALIZE()

int MPI_Finalize(void)

MPI_FINALIZE(IERROR)
   INTEGER IERROR

   This routine cleans up all MPI states. Once this routine is called, no MPI rou-
tine (even MPI_INIT) may be called. The user must ensure that all pending com-
munications involving a process complete before the process calls MPI_FINALIZE.

MPI_INITIALIZED( flag )

   OUT flag Flag is true if MPI_INIT has been called and
   false otherwise.

int MPI_Initialized(int *flag)

MPI_INITIALIZED(FLAG, IERROR)
   LOGICAL FLAG
   INTEGER IERROR

   This routine may be used to determine whether MPI_INIT has been called. It is the only
   routine that may be called before MPI_INIT is called.

MPI_ABORT( comm, errorcode )

   IN comm communicator of tasks to abort
   IN errorcode error code to return to invoking environment

int MPI_Abort(MPI_Comm comm, int errorcode)

MPI_ABORT(COMM, ERRRCODE, IERROR)
   INTEGER COMM, ERRRCODE, IERROR

   This routine makes a "best attempt" to abort all tasks in the group of comm. This
   function does not require that the invoking environment take any action
   with the error code. However, a Unix or POSIX environment should handle
   this as a return_errorcode from the main program or an abort(errorcode).
   MPI implementations are required to define the behavior of MPI_ABORT at
   least for a comm of MPI_COMM_WORLD. MPI implementations may ignore the comm
   argument and act as if the comm was MPI_COMM_WORLD.
8.1 Requirements

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

1. provide a mechanism through which all of the MPI defined functions may be accessed with a name shift. Thus all of the MPI functions (which normally start with the prefix “MPI.”) should also be accessible with the prefix “PMPI.”.
2. ensure that those MPI functions which are not replaced may still be linked into an executable image without causing name clashes.
3. document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether she must implement the profile interface for each binding, or can economize by implementing it only for the lowest level routines.
4. where the implementation of different language bindings is done through a layered approach (e.g., the Fortran binding is a set of “wrapper” functions which call the C implementation), ensure that these wrapper functions are separable from the rest of the library.

This is necessary to allow a separate profiling library to be correctly implemented, since (at least with Unix linker semantics) the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the person who builds the profiling library to extract these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.
5. provide a no-op routine MPI_PCONTROL in the MPI library.

8.2 Discussion

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines.
Since MPI is a machine-independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code which implements MPI on any particular machine. It is therefore necessary to provide a mechanism by which the implementors of such tools can collect whatever performance information they wish without access to the underlying implementation.

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

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

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

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

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

### 8.3 Logic of the Design

Provided that an MPI implementation meets the requirements above, it is possible for the implementor of the profiling system to intercept all of the MPI calls which are made by the user program. She can then collect whatever information she requires before calling the underlying MPI implementation (through its name shifted entry points) to achieve the desired effects.

#### 8.3.1 MISCELLANEOUS CONTROL OF PROFILING

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

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

These requirements are met by use of the MPI_PCONTROL.
MPI_PCONTROL(level, ...)

IN int level Profiling level

int MPI_Pcontrol(const int level, ...)

MPI_PCONTROL(level)

INTEGER LEVEL, ...

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

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

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

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

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

The provision of MPI_PCONTROL as a no-op in the standard MPI library allows them to modify their source code to obtain more detailed profiling information, but still be able to link exactly the same code against the standard MPI library.

8.4 Examples

8.4.1 Profiler Implementation

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

static int totalBytes;
static double totalTime;

int MPI_SEND(void * buffer, const int count, MPI_Datatype datatype,
8.4.2 MPI LIBRARY IMPLEMENTATION

On a Unix system, in which the MPI library is implemented in C, there are various possible options, of which two of the most obvious are presented here. Which is better depends on whether the linker and compiler support weak symbols.

Systems with weak symbols

If the compiler and linker support weak external symbols (e.g., Solaris 2.x, other system V.4 machines), then only a single library is required through the use of #pragma weak thus:

```
#pragma weak MPI_Example = PMPIL_Example
```

```
int PMPIL_Example(/* appropriate args */)
{
    /* Useful content */
}
```

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

Systems without weak symbols

In the absence of weak symbols then one possible solution would be to use the C macro pre-processor thus:

```
#ifdef PROFILELIB
#    ifdef __STDC__
#        define FUNCTION(name) P##name
#    else
```

```
define FUNCTION(name) P/**/name
#define FUNCTION(name) name
#endif
#endif

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

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

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

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

```bash
% cc ... -lnyprof -lmpi -lm
```

Here libyprof.a contains the profiler functions which intercept some of the MPI functions. libmpi.a contains the “name shifted” MPI functions, and libm.a contains the normal definitions of the MPI functions.

### 8.4.3 COMPLICATIONS

**Multiple counting**

Since parts of the MPI library may themselves be implemented using more basic MPI functions (e.g., a portable implementation of the collective operations implemented using point-to-point communications), there is potential for profiling functions to be called from within an MPI function which was called from a profiling function. This could lead to “double counting” of the time spent in the inner routine. Since this effect could actually be useful under some circumstances (e.g., it might allow one to answer the question “How much time is spent in the point-to-point routines when they’re called from collective functions?”), we have decided not to enforce any restrictions on the author of the MPI library which would overcome this. Therefore the author of the profiling library should be aware of this problem, and guard against it herself. In a single-threaded world this is easily achieved through use of a static variable in the profiling code which remembers if you are already inside a profiling routine. It becomes more complex in a multi-threaded environment (as does the meaning of the times recorded!).
Linker oddities

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

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

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

8.5 Multiple Levels of Interception

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

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

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

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

Unfortunately such an implementation may require more cooperation between the different profiling libraries than is required for the single-level implementation detailed above.


