15.2.2 Function Pointer Interception (QMPI)

Motivation

While the name-shifted interface with the prefix PMPI_ has been successful in allowing a tool to access application calls into MPI, it has the notable limitation that only a single tool can intercept MPI calls. This restricts the ability to have complimentary tools or even allow tools to attach duplicate versions of themselves to profile different behaviors.

In this section, we introduce a new interface, using C function pointers, which is more flexible than the PMPI_ interface and will address these concerns. This interface continues to impose little overhead on application performance. This will still include a name-shifted interface using the prefix QMPI_ which can be used to avoid tools using the interception interface, but the primary interaction method will be through function pointers as described below.

Requirements

The requirements for the function pointer interface are similar to the PMPI_ name-shifted interface. An implementation *must*

1. provide an interface to register callback functions to be used when requested MPI procedures are called.
2. provide interfaces for applications to specify a set of tools and an ordering for those tools to be called when a callback function has been registered and the matching MPI procedure is called.
3. provide an interface for a tool to register tool-specific memory addresses that can be provided back to the tool when its callback functions are called.
4. provide a mechanism through which all of the MPI defined functions, except those allowed as macros (See Section 2.6.4), may be accessed with a name shift. This requires, in C and Fortran, an alternate entry point name, with the prefix QMPI_ for each MPI function in each provided language binding and language support method. For routines implemented as macros, it is still required that the QMPI_ version be supplied and work as expected, but the tools callback functions may never be called.

Tool Life Cycle

This section describes the life cycle of a tool using the callback function interfaces (as opposed to the name-shifted PMPI_ functions). The details for each new API function will be defined in Section 15.2.2.

Each tool will need to go through three stages:

Registration For a tool, the registration phase begins before the application's main function is called and finishes when MPI is initialized (whether that happens via an explicit call in the World Model or implicitly in the Sessions Model). During this phase, the tool is responsible for registering itself with the MPI library by calling the function MPI_REGISTER_TOOL_NAME. This function will allow the tool to provide its name to the MPI library and provide a callback function to use if the user requests that the tool be loaded. All registration must occur before MPI initialization is started and no tool can be registered with MPI after that point.

The specific mechanism for calling the registration function before MPI is initialized is not specified here, but a number of options are available (e.g., compiler-specific constructor attributes on library functions).

Initialization The initialization phase of a tool occurs when the callback function registered via MPI_REGISTER_TOOL_NAME is called. During this phase, the tool should register a pointer to storage for tool data (if needed) and pointers to functions for each of the MPI procedures that the tool would like to intercept. The storage pointer will be provided back to the tool when the interception functions are called later.

Interception The interception phase begins when the tool's initialization callback function returns. During this phase, callback functions will be called for any MPI procedures for which the tool registered a callback function. These functions will be called in an order specified by the user in an implementation-dependent way. Inside of the interception function, the tool may call any other MPI function, but it must do so using the function pointer of the tool that would be called next according to the user-specified ordering. This happens by getting the function pointer for the MPI procedure via MPI_GET_NEXT_TOOL_FUNCTION. The tool should avoid calling the MPI procedure directly to avoid recursion back into through other tools that may already have been called in addition to calling itself again.

When the tool's interception function is called, it will include all of the MPI procedure's arguments as provided by the user (or the previous tool if there is more than one in use). In addition, two more arguments: a context object and the ID of the tool being called. The ID is assigned by the MPI library itself and may be non-monotonic or non-increasing from tool to tool. Each tool is responsible for retrieving the function pointer and ID for the next tool when it intercepts an MPI procedure. The function pointer for the last tool will point to the MPI library's implementation of the MPI procedure.

Application Life Cycle with a Tool

This section describes the usage of a tool from an application's perspective when using the callback function interfaces (as opposed to the name-shifted `PMPI_` functions). As opposed to the tool life cycle, this life cycle does not have stages because, from an application's perspective, the tool is essentially transparent. The application needs to do two things to ensure a tool is intercepting its MPI procedure calls.

First, the tool needs to be linked with the application's binary. This is the same as with the name-shifted interfaces, but instead of overriding symbols, the tool will be responsible for its own bootstrapping.

Second, the user needs to specify which tools should be loaded and in which order. The exact mechanism for this is implementation-specific, but one option would be a comma-separated list provided by an environment variable.

Note that while any number of tools can be loaded, an implementation may have practical limits on the number of tools that are supported due to resource consumption concerns. An MPI implementation should provide documentation indicating the maximum number of tools it supports by default.

Callback Function Interfaces

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

MPI_REGISTER_TOOL_NAME(tool_name, init_fn_ptr)

IN	tool_name	name of tool (string)
IN	init_fn_ptr	pointer to callback function (function)

C binding

```
int MPI_Register_tool_name(char *tool_name,
             MPI_Tool_init_function *init_fn_ptr)
```

This function is used by the tool to register with the MPI implementation. This function must be called before MPI is initialized by any other function. A single tool can call this procedure multiple times, but it must provide a unique tool_name each time the procedure is called. When the MPI library is initialized by any call (either explicitly in the World Model or implicitly in the Sessions Model), calling this function will result in undefined behavior.

When the MPI library is initialized, it will determine the number and order of the tools that the user has requested and call the function pointer specified by init_fn_ptr once for each instance of the tool requested by the user. Each time a function pointer is called, a new tool ID will be provided to allow the tools to differentiate themselves and to be used later when the tool needs to get information from MPI about itself or other tools.

The function pointer init_fn_ptr should be of the form:

```
typedef void MPI_Tool_init_function(int tool_id);
```

When inside the tool initialization function, the tool can call two MPI procedures: register internal storage and register callback functions for MPI procedures. To accomplish the first, the tool should use this function.

MPI_REGISTER_TOOL_STORAGE(tool_id, tool_storage)

IN	tool_id	ID of calling tool (integer)
IN	tool_storage	pointer to tool-specific storage

C binding

```
int MPI_Register_tool_storage(int tool_id, void *tool_storage)
```

This function allows MPI to internally associate a tool-specific storage address pointed to by tool_storage with an ID indicated by tool_id. tool_id should be the same value that was provided to the tool during initialization. This storage address will be provided back to the tool each time one of its callback functions is called so the tool has the ability to store state about itself in a portable way.

> *Rationale.* While a tool could store information in its own address space, when multiple copies of the same tool are used, it may be necessary to be able to differentiate between the storage of multiple copies of the same tool. Therefore, it is more portable to use the tool_storage argument than to use another mechanism. (*End of rationale.*)

After the tool has set up its internal storage, it will also need to register callback functions for any MPI procedure it intends to intercept. This happens using the following function.

MPI_REGISTER_TOOL_FUNCTION(tool_id, function_enum, function_ptr)

IN	tool_id	ID of calling tool (integer)
IN	function_enum	identifier of function to be intercepted (integer)
IN	function_ptr	pointer to callback function (function)

C binding

```
int MPI_Register_tool_function(int tool_id, enum
             MPI_Functions_enum function_enum, void (*function_ptr) (void))
```

This function uses the same value for tool_id that was provided to the tool during initialization. It also uses an enumeration value, function_enum, that has a value for each MPI procedure that can be intercepted (along with any implementation-specific functions, if any). The values for function_enum should match with their respective MPI procedure names in all capital letters with a `_T` at the end. For example, the value for MPI_SEND would be `MPI_SEND_T`.

The final argument is a function pointer that will be called when the requested MPI procedure is called. This function pointer should match the original MPI procedure, but should have two additional arguments at the beginning of the argument list. The first argument is a context object of type MPI_Context, and the second is the tool_id. The function_ptr argument will accept any function pointer. Again, using MPI_SEND as an example, the function pointer would look like this:

```
int MPI_Send(QMPI_Context, int tool_id, const void *buf, int count,
             MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

Once a tool has intercepted a function, it will need to retrieve the pointer for the function it needs to call next when it is ready to continue the execution of the MPI procedure. To do this, the tool will use the following function.

MPI_GET_NEXT_TOOL_FUNCTION(tool_id, function_enum, function_ptr, next_tool_id)

IN	tool_id	ID of calling tool (integer)
IN	function_enum	identifier of function being intercepted (integer)
OUT	function_ptr	pointer to function of the next tool (function)
OUT	next_tool_id	ID of calling tool (integer)

C binding

```
int MPI_Get_next_tool_function(int tool_id, enum
             MPI_Functions_enum function_enum, void (*function_ptr) (void),
             int *next_tool_id)
```

This function requires the tool_id for the calling tool so MPI can determine the tool that should be called next, represented by next_tool_id. The function_enum value indicates which MPI procedure is being requested. Details of the enum type can be found in the definition of MPI_REGISTER_TOOL_FUNCTION. With these two pieces of information, MPI provides function_ptr, which points to the next function that should be called in order to fulfil the semantics of the MPI procedure, and next_tool_id, which should be passed as an argument to function_ptr to indicate the ID of the tool being called.

Advice to users. Requesting the next function pointer can be an expensive operation and the result does not change after the initialization stage is begun. Therefore, it is recommended that for performance sensitive tools, to avoid extra memory lookups during every MPI procedure, the tool caches all function pointers that it will use during the initialization phase. This can be accomplished by intercepting all of the MPI initialization procedures (implicit and explicit). (*End of advice to users.*)

While in the interception function, the tool may need access to its storage address that was previously registered with MPI. To accomplish this, it should use the following function.

MPI_GET_TOOL_STORAGE(context, tool_id, storage)

IN	context	Context object (handle)
IN	tool_id	ID of calling tool (integer)
OUT	storage	pointer to beginning of registered storage (choice)

C binding

```
int MPI_Get_tool_storage(MPI_Context context, int tool_id, void *storage)
```

This function returns the address of the storage location that was previously registered with a call to MPI_REGISTER_TOOL_STORAGE. The context argument should be the same as the context handle that was provided to the interception procedure. The handle is not directly usable by the tool and should only be used as input back to this function. The tool_id is that of the calling function to determine which storage pointer to return. MPI returns a pointer to the storage location with the storage argument.

MPI_GET_CALLING_ADDRESS(context, address)

IN	context	Context object (handle)
OUT	address	memory address of the calling location (choice)

C binding

```
int MPI_Get_calling_address(MPI_Context context, void *address)
```

For some tools, determining the address of the application function which called an MPI procedure can be useful. To fulfil this functionality, MPI_GET_CALLING_ADDRESS returns the memory location where the application called an MPI procedure that led to the function interception. This calling address will only represent the original MPI procedure call and not any of the interception functions that may have been called between the application and the current tool. The tool must provide the context handle and the address is returned with the address argument.

15.3 The MPI Tool Information Interface

MPI implementations often use internal variables to control their operation and performance and rely on internal events for their implementation. Understanding and manipulating these variables and tracking these events can provide a more efficient execution environment or improve performance for many applications. This section describes the MPI tool information interface, which provides a mechanism for MPI implementors to expose variables, each of which represents a particular property, setting, or performance measurement from within the MPI implementation, as well as expose events that can be tracked by tools. The interface is split into three parts: the first part provides information about, and supports the setting of, control variables through which the MPI implementation tunes its configuration. The second part provides access to performance variables that can provide insight into internal performance information of the MPI implementation. The third part enables tools to query available events within an MPI implementation and register callbacks for them.

To avoid restrictions on the MPI implementation, the MPI tool information interface allows the implementation to specify which control variables, performance variables, and events exist. Additionally, the user of the MPI tool information interface can obtain metadata about each available variable or event, such as its datatype, and a textual description. The MPI tool information interface provides the necessary routines to find all variables and events that exist in a particular MPI implementation; to query their properties; to retrieve descriptions about their meaning; to access and, if appropriate, to alter their values; and (in case of events) set callbacks triggered by them.

Variables, events, and categories across connected MPI processes with equivalent names are required to have the same meaning (see the definition of “equivalent” as related to strings in Section 15.3.3). Furthermore, enumerations with equivalent names across connected MPI processes are required to have the same meaning, but are allowed to comprise different enumeration items. Enumeration items that have equivalent names across connected MPI processes in enumerations with the same meaning must also have the same meaning. In order for variables and categories to have the same meaning, routines in the tools information interface that return details for those variables and categories have requirements on what parameters must be identical. These requirements are specified in their respective sections.