Computational Science and Engineering

Technische Universität München

Master's Thesis

Design, Implementation and Testing of a new Profiling Interface for MPI

Bengisu Elis

Computational Science and Engineering

Technische Universität München

Master's Thesis

Design, Implementation and Testing of a new Profiling Interface for MPI

Design, Implementierung und Testen einer neuen Profiling-Schnittstelle für MPI

Author:	Bengisu Elis
Supervisor:	Prof. Dr. rer. nat. Martin Schulz
Advisor:	M.Sc. Dai Yang
Submission Date:	05.10.2018

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 05.10.2018 Bengisu Elis

Abstract

MPI is a library that is widely-used to develop applications to run in parallel. It provides an efficient profiling interface- PMPI - which is used by many profiling tools based on MPI. However, PMPI supports a single profiling tool at a time and the application must be re-linked with the tool each time a new tool is to be used. This one tool at a time nature of PMPI leads to monolithic tool designs, that are tailored for a certain MPI application and profiling goal. Hence, it discourages the reusability and modularity of the code leading to a very complex tool nature.

On the other hand, if tools could work in coordination and could be utilized concurrently, desired tool functionalities could be achieved by combination of simpler tools. This approach would enhance testabilitiy, reusability and modularity of tools.

The major question related to such an approach is, how to make tools work concurrently and in coordination with each other. In this thesis a prototype of an interface - called QMPI- , that enables concurrent and collaborative execution of tools is designed, implement and tested. The final design of the interface is made by enhancing and extending other previous designs, which are also a part of this thesis work. The final design aims to achieve concurrent execution of tools by successive calls of tool wrappers in a nested fashion. The order of the tools in the toolchain is determined by a vector data structure, in which each tool has their corresponding instance. The final design is fully implemented along with some simple tools, that are used for testing the final design implementation.

Finally, some performance tests are conducted to prove the efficiency of our QMPI prototype. It is shown that, linking QMPI with the MPI application - without using tools- causes negligible time overhead. Since QMPI provides such a low time overhead it can be linked with applications by default, in order to enable simple activation or deactivation of tools. Furthermore, it is also shown that our prototype implementation has linear timeoverhead scaling with the increasing size of the toolchain. It provides

low time overhead for a realistic size of toolchain.

Contents

List of Figures

List of Tables

1 Introduction

QMPI enables profiling tools for MPI applications to work in collaboration in order to achieve a certain profiling task or provide some certain features to the user. It builds a tool-chain structure, which defines the execution order of the nested calls of tool functions. In this work its design, implementation and features are explained in details.

1.1 Problem and Motivation

PMPI is an interface provided by Message Passing Interface (MPI) for profiling the MPI applications. To profile an MPI application using PMPI, a developer must implement a tool, that performs all the functionality required for the desired profiling. Later, the application to be profiled, must be linked with the tool library. For each different tool, application should be re-linked with the tool to be used. Since tool libraries are made of wrapper functions that are linked to the MPI calls in an MPI application, using and linking more than one tool library at a time is not possible. Therefore, all of the functionality necessary to achieve a profiling goal, must be implemented in a single tool. This limitation not only increases code complexity, but also increases the reusability and modularity of the tool code. However, considering the variety of profiling needs for MPI applications, it is both cumbersome and inefficient to implement a new tool for each profiling need.

A good solution for this problem may be enabling tools, that are simple and that have few functionalities, to work in collaboration. In other words, designing small tool modules rather than monolithic tools. Necessary functionality, to perform the profiling, can be acquired by using the combination and collaboration of different tools, that are each designed for analyzing a narrow set of features of an application.

Tools can successively call each others wrapper functions in a certain order to provide the desired functionalities. This approach not only leads to simple tool codes but also leads to better testability and maintainability. The collaborative behavior and the resulting chain like structure is named as toolchain or toolstack and illustrated in Figure 1.1[1]. Throughout this work toolchain and toolstack are used interchangeably.

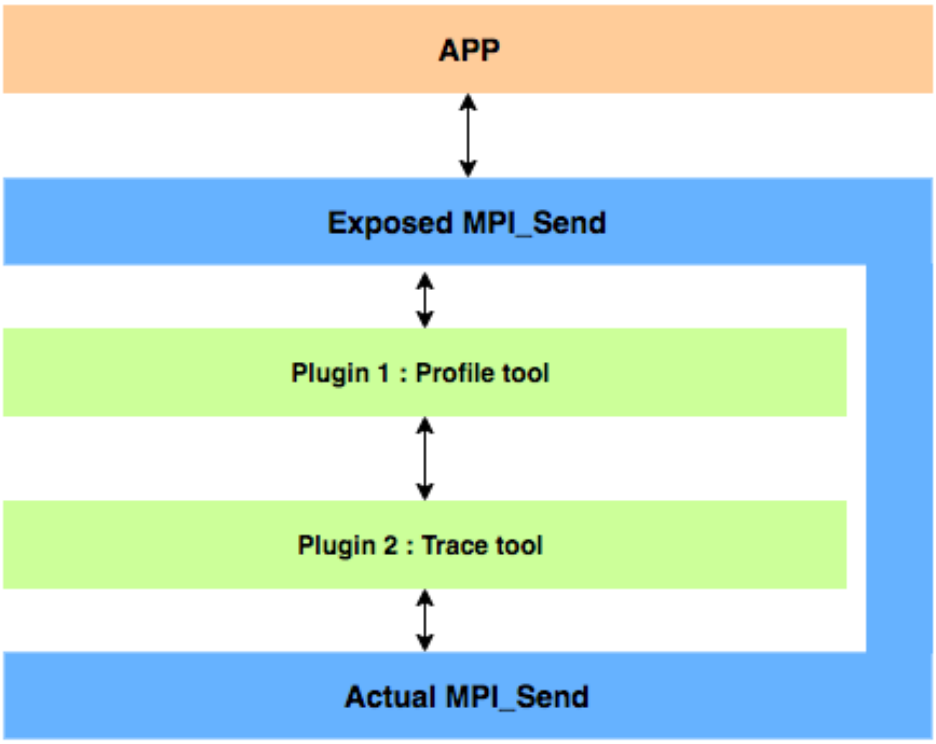


Figure 1.1: Toolchain structure supported by QMPI

However, a toolchain structure requires (i) interfacing of tools with each other, (ii) interfacing with PMPI, (iii) interfacing with the original MPI application. Construction of these interfaces in a programmatic way underlies the major focus of this work. The QMPI interface designed, implemented and tested as a part of this work, handles all of the required interfaces for a toolchain structure to function as desired.

[1] taken from Lawrence Livermore National Laboratory MPI forum meeting presentation "PMPI successor- Restart 2017"

Although there exists other designs and implementations of interfaces, to accomplish the mentioned behavior of the tools (section 5.3, 5.4), QMPI interface design and implementation in this thesis proposes and introduces a different approach to the problem. After initialization phase, it constructs a table that contains pointers to follow-on routines of each tool wrapper. Thus, tools can directly access these functions through the pointers in the table. This reduces the time overhead between each wrapper call.

1.2 Goals

MPI already offers an efficient profiling interface- called PMPI. The main goal of this work is to extend PMPI, in order to present backwards compatibility when enabling concurrent and collaborative execution of tools in toolchain formation.It is aimed to design and implement a prototype of an interface that provides these features without introducing any changes to the existing implementations of PMPI and MPI. Furthermore, It is decided and aimed to design QMPI as a PMPI tool, in order to have a simpler QMPI design.

Another goal of this work is to achieve low time overhead, when MPI applications are executed with a toolchain, that contains some tools, and very low time overhead, when there are no tools in the toolchain. Also, when there are no tools in the toolchain the functionality of the applications should not be affected. In other words, linking QMPI with an application and executing without any tools in the toolstack, must not reduce the performance in a statistically significant manner and should not alter the existing functionality of the application.

1.3 Structure of the Thesis

In this thesis an interface for MPI applications, called QMPI, is introduced. The rest of this thesis is structured as follows. In Chapter 2 three different designs for the QMPI interface are presented. After that, the implementation of the final design of the QMPI is explained in details in Chapter 3. In Chapter 4, implementation and results of evaluation tests of QMPI are given. Chapter 5 gives background information related

to Message Passing Interface (MPI), its profiling interface PMPI and other interfaces that aim similar goals of QMPI. Finally, Chapter 6 concludes this thesis by giving an overview of the contributions and explaining possible ways to extend this work in the future.

2 Design

In this section three different designs that are tailored for the QMPI interface prototype are introduced. Firstly, some general assumptions that have been made for the design process are specified. Second, the requirements of the designs are given. After that, the proposed MPI forum design, followed by the intermediate design are explained. Finally, the final design which leads to the actual implementation of the interface and how this design fulfills the requirements are clarified.

2.1 General Assumptions

Following assumptions have been made for the simplicity of the design of the QMPI interface.

First of these assumptions is that, **the tools are responsible for matching runtimes when they are implemented in C++ programming language.** Therefore, simple tools aimed in testing QMPI are all either written using C programming language or its interfacing from C++ to C programming language is assumed to be already done.

Secondly, the designed QMPI will only support simple chain like tool stacks as in Figure 1.1. Directed acyclic tool structures, inter-tool services, tool dependencies or coordinations are not supported. Another assumption is that, the tools inserted in the toolstack should be independent from each other. In other words, tools should be capable of performing their functionalities only with an MPI application and the MPI library, without requiring any support from the other tools.

The last assumption is that, the tools and MPI applications are to be executed

using a single thread. Therefore, thread safety is not considered within the scope of this work.

2.2 Requirements and Description of the Interface

The aim of the QMPI is to allow plugging toolstacks in between MPI applications and the actual MPI library. The desired structure is illustrated in Figure 1.1. The orders of the tools in the stack should be defined by the user. Interfacing and flow of the execution between tools is established by QMPI interface. A design for the QMPI should meet the following requirements:

1. The interface should be implemented in C according to C99 standards.

2. Designed interface should enable load tools at runtime.

3. The interface should support multiple tools that run concurrently in a single process.

4. The existing MPI functionalities should be maintained. Therefore, the interface must cover all MPI routines.

5. The existing functionalities of the tools must be preserved.

6. The tools should have different context, i.e., they should have independent storage space for their variables.

7. In case of no tools added to the toolstack the interface should provide zero to negligible time-overhead.

8. In case that there are tool/s in the stack, the time overhead must be low enough so that the application performance will not be badly affected.

9. The design should adopt wrapper functions approach to support pre and post call activation of the tools

10. QMPI interface must be built upon PMPI and PMPI should provide the profiling interface functionality to the MPI library.

11. QMPI interface must enable pre and post call activation of the tools in the stack. The nested structure that enables pre and post call activation behavior is illustrated in Figure 2.1[1]

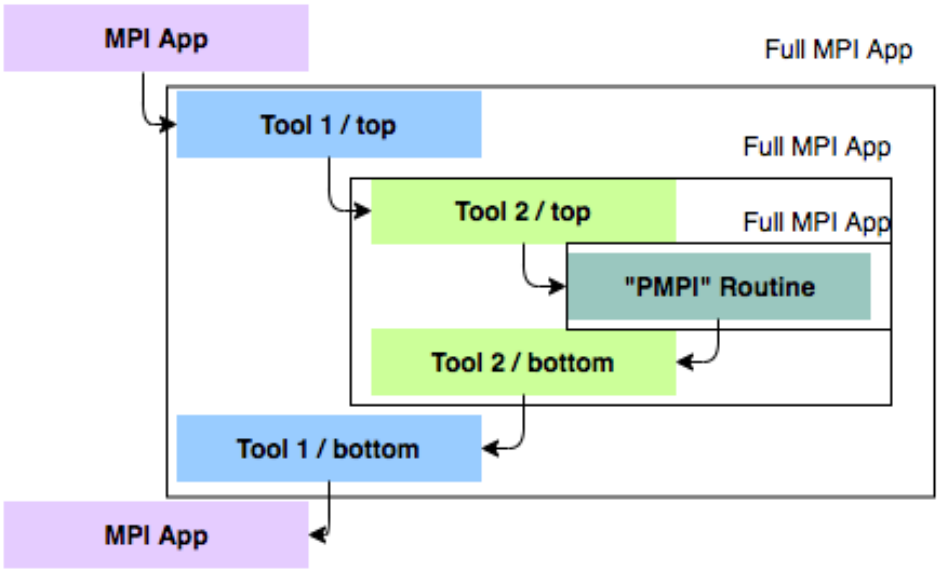


Figure 2.1: Pre and Post call activation enabled by nested wrapper calls

2.3 MPI Forum Design

The MPI forum design is the initial design, that is made by MPI standards forum and serves as a starting point for the final design of our QMPI prototype. This design is expanded, changed and adjusted to form the final design.

[1] .

2.3.1 Additional Assumptions for the Design

One of the assumptions made for this design is that, once the configuration of the toolstack structure is done, adding new tools to the toolstack and removing tools from the stack are not allowed. Another assumption for this design is that threading features are not necessary. Therefore, the features of MPI for threaded executions are ignored. In this design QMPI is considered as another tool that is located at the top of the toolstack and orchestrates the tools and their interfaces.

2.3.2 Basic Wrapping of the MPI functions

In order to enable pre-and post call activation of the tools the function wrapping approach must be adopted. Therefore, each tool implements a set of wrapper routines for each MPI function, that it is required to intercept.

Since in this design QMPI is considered as another tool that is used for configuration and orchestration of the toolstack, it is required to have wrapper routines as well. QMPI must have one wrapper per MPI function so that the final structure can cover all of the existing features of the MPI library.The key wrapper template for this design is as in Listing 2.1:

Listing 2.1: QMPI wrapper template for Initial design

```
Int QMPI_X (<MPI arguments>, void* context, MPI_blob){
    retrieve function/context table from own context -> next_table;
    qmpi_x_t pqmpi_x;
    MPI_Table_query( QMPI_X , &pqmpi_x , &next_context , next_table);
    //... Do work ...
    err=pqmpi_x( <MPI arguments> , next_context , blob);
    // ... Do work
    return err;
}
```

"<MPI_arguments>" , among the parameters of "QMPI_X" wrapper, are standard arguments an MPI function has.The parameter "void* context" pointer is used for

context separation of variables in the tool. "MPI_blob", among the parameters of the "QMPI_X" wrapper, is for extra parameters that might be necessary to be passed to the wrapper routine.

2.3.3 Function Tables

Function tables are the data structures that help the tools keep track of the next function, i.e., next tool in the toolstack, to be called. Each tool in the toolchain stores its own function table.

These tables are 1D tables and may be considered as a single column or a row. The tables contain pointers to the corresponding follow on routines, that the tools have to call in their wrapper routines. In order to get the pointer to a follow-on routine, tools send queries to their function tables. Determination of which element in the table to choose as a follow-on routine, depends on either the name of the MPI function to be wrapped or the index of a table element that contains a pointer to the follow-on routine. Therefore, a query to get an element of this table requires either an MPI function name or an index. The output of such a query is a function pointer and a context pointer.

Context pointers are another very important data structures that are unique and private for each tool. In addition to function pointers, each element of a function table includes context pointers that belong to the tool, whose wrapper routine is pointed by the function pointer. Context pointers are void pointers that can point to any kind of data structure. They point to the separate storage space for variables of the tools or any kind of information that should not be exposed to other contexts.In other words, they provide the tools the location of their separate storage space. Thus, context pointers provide context separation for two different executions of the same tool file that is caused by including the same tool file twice in the toolchain. This gives the user the advantage of using same tool file twice without allocating extra memory on their system for a replica of the tool that is to be used.

Function tables may be sparse initially and are filled after initialization. During or before MPI_Init function call, QMPI instantiates an instance for each of the tools in the stack. Also, at this stage context pointers and function tables are configured and assigned to the corresponding tool instances. Once function tables are configured

they cannot be altered during the execution of the MPI Application.

The cells of the function tables are filled in a way that even if the tool does not intercept some certain MPI function there is still a function pointer located, in the reserved cell of the table, pointing to some follow-on routine. In this case the pointers point either to the corresponding routine of the next tool, which intercepts the same specific MPI function with the caller wrapper function, or a function pointer to corresponding PMPI routine if none of the tools in the stack intercepts that specific MPI function. It should be noted that, these tables cannot have NULL pointers, i.e., holes.

Such placement of the function pointers in the tables, enables wrappers to call MPI functions, that are different than the MPI Functions the wrapper routine aims to wrap. The desired behavior may be explained as follows. When a tool, makes an MPI call in one of its wrappers, to a different MPI function than the wrapper aims to wrap, the program flow continues execution in the same level of the toolstack, i.e, in the same tool. Then, the next tools in the levels below are called in accordance with the pointers in function tables. At the very last level of the toolstack, PMPI is called and the flow of the execution climbs up in the levels of the stack until the level of the tool, that had made the MPI call is reached. This behavior is illustrated by the red arrows in Figure 2.2. It can be seen how a function call to the MPI_Comm_rank interrupts execution of MPI_Send execution chain.

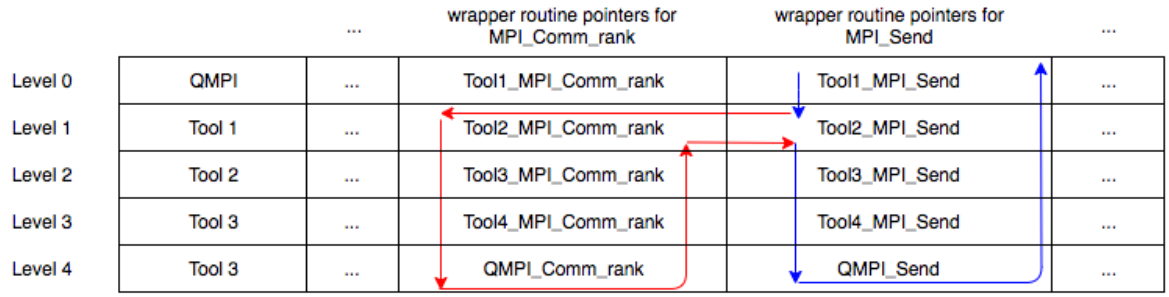


Figure 2.2: Illustration of execution flow when tool wrappers call different MPI functions

In summary, each tool has following structures exclusively available: a function table and a context pointer. They provide the necessary information to the tool to call its follow-on routine: the pointer to the routine to be call and the context to use.

2.3.4 Tool Responsibilities

(a) **Independent Tools:** Tools must be independent of each other. They must be capable of running on their own and they should be built upon a standard compliant implementation of MPI routines (PMPI interface).

(b) **Tool registration:** Tools must register themselves with the QMPI by calling an MPI routine.

(c) **Who am I Data (WAID):** As tools register themselves they should pass a Who am I data to QMPI. WAID constitutes of the following information:

- Name of the tool
- Name of the developer
- Instance initialization routine
- Extension options for other information, e.g., dependencies etc.

(d) **Storage Structure:** Another responsibility of the tools is that they must create their own storage structure and allocate memory for it. Later on, they should hook the context pointer provided by the QMPI to this storage structure. At this point, a good advice to tool developers can be to avoid using global variables but use this storage structure with the context pointer hooked, instead of global variables.

(e) **Function Tables:** Each tool in the toolchain is responsible for storing their own function tables, provided by QMPI, in their storage structure.

(f) **Intercepted Routines:** Tools should expose the MPI routines, that are to be intercepted by them, to QMPI. This is essential to fill function tables.

(g) **Dynamic Tools:** Design of QMPI prototype is built upon the assumption that tools are loaded at the runtime. Therefore, tools must be compiled as shared (dynamic) libraries.

(h) **Wrappers:** The tools implement wrappers that are derived from the key wrapper - given in section 2.3.2. Tool routines must call follow on routine to avoid breaking the toolchain.

(i) **Access through function tables:** Wrappers cannot call the native MPI routines directly, but they can call the routines located in their function table. Therefore, access to MPI routines is only possible through function tables.

(j) **Well defined function names:** Tools contain a set of wrapper routines with well-defined names and there must be wrapper, hence one name defined for each MPI routine to be intercepted.

(k) **Extra parameters:** QMPI interface may require extra parameters to be passed between tools. Therefore,the wrappers in the tools should be able to take and pass the extra parameters required by QMPI. For instance signature of a QMPI wrapper function may be as follows: int QMPI_Comm_rank(MPI_Comm comm, int *rank, <extra data>). , <extra data> parameters are extra parameters used by QMPI. Tools should be able to receive and carry over these extra parameters to their follow-on routine.

2.3.5 Aimed Workflow by the Design

The flow diagram of the original - design until the end of MPI initialization ("MPI_Init") - is given in the 2.3.

The execution starts with an "MPI_Init" call from the MPI application. If there are no tools to be placed to the toolchain the flow jumps directly to assigning of context pointers and construction of the function tables. Since there are no tools in the stack, a single function table is instantiated and assigned to QMPI interface. All of the elements in this function table points to a PMPI routine. Therefore, as the flow jumps to "QMPI_Init" wrapper which directly calls "PMPI_Init" function. As "PMPI_Init" returns, execution flow directly returns to QMPI interface. Later, as "QMPI_Init" returns, flow of execution returns back to the MPI application.

As "MPI_Init" is called from the MPI application, if there are tools to be placed to the toolchain, tools register themselves with the QMPI interface and QMPI instantiates

instances for each tool and loads the tool libraries. Then, QMPI instantiates and assigns context pointers and constructs function tables. The function tables are passed to the tools to be stored by the tools. The context pointers are passed to the tools in order to be hooked to the memory location, allocated by the tools. Later, "QMPI_Init" is executed. "QMPI_Init" queries a function pointer for the follow-on routine and calls the follow-on routine for execution. This query and execute cycle goes on from tool wrapper to tool wrapper until "PMPI_Init" routine is reached. "PMPI_Init" executes actual functionality of "MPI_Init" and returns to the tool wrapper which called it. The nested function calls return one by until the execution flow reaches back to "QMPI_Init". Finally, "QMPI_Init" returns to MPI application.

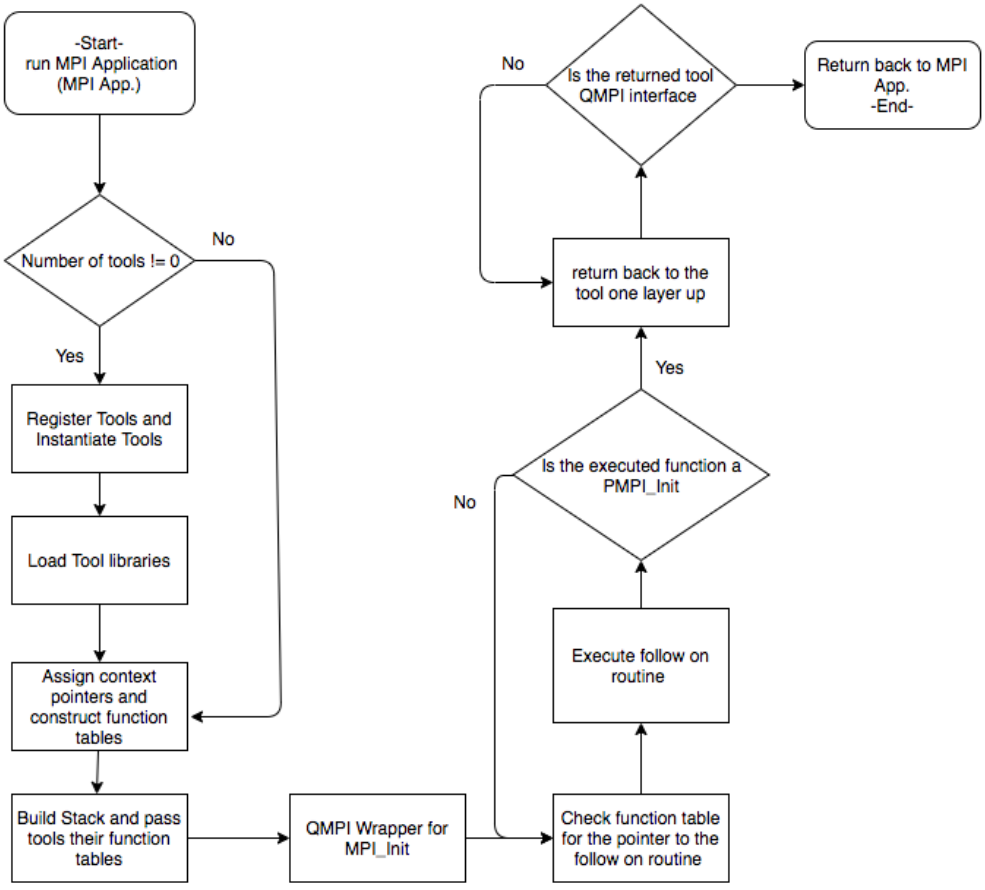


Figure 2.3: Aimed workflow of "MPI_Init" by Original design

Also, the flow of an MPI application, after initialization, executed with QMPI interface is also given in Figure 2.4. The flowchart starts with a MPI call from the MPI application which directs the execution flow to the corresponding QMPI wrapper

function to the MPI call. QMPI wrapper checks its function table and gets the function pointer to its follow-on routine. Query and execute cycle continues until the function pointer to corresponding PMPI function is executed by a tool in the toolchain. Then, PMPI function performs the actual functionality of the MPI function, that is called by the MPI application and execution returns back to the tool wrapper, that called the PMPI function. The nested call of the tool wrappers returns one by one until QMPI wrapper is reached again. QMPI wrapper returns execution flow back to MPI application. This flow occurs for each MPI function call by the MPI application.

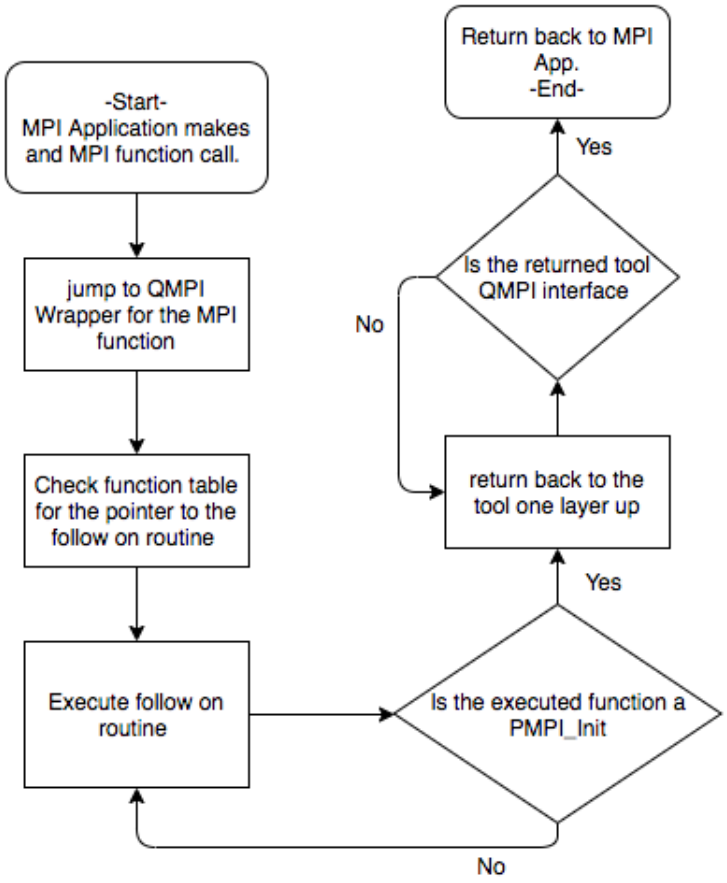


Figure 2.4: Aimed Workflow by the original design

2.4 Intermediate Design

This design is made by ignoring some of the assumptions and requirements for sake of simplicity. In the final design this design is enhanced, changed and expanded, according to requirements and performance needs of the interface.

Since the design of the function tables is the same as in the original MPI design, there is not a section in this chapter devoted to them.

Also, this design is the first design used for implementations and is fully implemented except for the context pointers and all 360 of the QMPI wrapper routines. Therefore, its implementation serves as an initial proof that some desired features and structures, such as constructing toolchains or constructing function tables, can be achieved.

2.4.1 Additional Assumptions for the Design

This design adopts all the assumptions stated before in general assumptions section-3.1. However, the following assumptions are unique to this design.

Firstly, it is assumed that the design can be made in C++. This can enable use of built-in data structures, such as vectors, maps etc., and the methods C++.

Secondly, in this design performance issues are not taken into considerations and main purpose of the design is to demonstrate that the desired interface is in fact possible to be built.

2.4.2 Tool Responsibilities

In this section, only the tool responsibilities, that are different than the original design, are explained.

(a) **Tool registration:** Tools no longer need to register themselves with the QMPI. In this design QMPI discovers tools through a config.txt file.

(b) **String array and Intercepted Routines:** To expose the MPI intercepting functions to QMPI, tools must have a string array with as many elements as the number of MPI functions in the standard MPI library. This array is used to store well-defined names of the wrapper routine names in the tool. QMPI uses indices of the elements in the array, in order to determine which element in the string array intercepts which MPI routine. Therefore, the order of the names of the routines must be in the same order with their wrapped MPI routine, when MPI routines are sorted in alphabetical order. In case the tool does not intercept a certain function then the NULL string is placed in the element where the function name for this intercepting routine is put. A sample string array and listing of the indices of the MPI functions can be found in the Appendix section.

(c) **Who am I Data (WAID):** WAID data is no longer necessary. QMPI gathers most of the information included in WAID by itself and stores them in the tool instance created.

(d) **Function Tables:** In this design function tables are kept by QMPI in a data structure and put into a vector that represents the toolchain. Therefore, storing the data structure is no longer a responsibility of the tool.

(e) **Well defined function names:** Tools constitute of set of wrapper routines with well-defined names and there should be one name defined for each MPI routine. Also, the names of the routines must be different than the names of the MPI routines. For example, as the name of a wrapper routine that wraps MPI_Init MPI_Init is not allowed but XMPI_Init can be used instead. This name shift is necessary since all MPI routines are declared as weak symbols and any other library linked to an application with routines with the same names as in MPI library - can replace the implementation of the MPI functions. QMPI interface uses wrapper names with MPI prefix therefore, the application links it MPI calls with the functions of the QMPI. Thus, if tools use the same MPI function names, this can cause to symbol collisions when tools are linked with the QMPI.

(f) **get_interceptions function:** There is also a routine declared and defined to return the names of the routines, from the array, one by one. This routine must be declared as follows:

Listing 2.2: get_interceptions signature

```
extern string get_interceptions ( int )
```

2.4.3 Configuration of Tools

In this design user must prepare a configure.txt file, which includes the paths of the tools to be added into the tool stack. Each tool path should be written in a single line with one line per each tool path. The order of the paths in the configuration file determines the order of the tools in the stack.

2.4.4 Tool Instances & Vector of tools

By using the configuration file provided by the user, QMPI instantiates instances of a structure called dynamic_lib. The dynamic_lib structure has the following fields:

1. a string to store path of the tool,
2. a void pointer to hook to the handle of the tool,
3. a context pointer to be hooked to the context structure provided by the tool,
4. a string array to store names of the MPI interception functions of the tool,
5. a table to be used as function table.

The implementation of the dynamic_lib structure is given in Listing 2.3:

Listing 2.3: implementation of dynamic lib structure

```
struct dynamic_lib
{
    std::string   path;
    void*    handle;
    void*     context;
```

```
    string   mpi_interceptions [ NUM_FUNCS ];
    std::map<std::string, void*> mapOfFunctions;

    //constructor
    dynamic_lib(std::string p) : path(p) ,handle(nullptr) {}

    //destructor
    ~dynamic_lib()
    {
        if (handle != nullptr)
        dlclose(handle);
    }
};
```

After initialization of these instances once per tool, they are pushed into the toolstack in the same order they are put in the configuration file. This stack is implemented as a vector by using the standard library of C++ .

A dynamic_lib instance named as QMPI is instantiated and it is the first element of the toolchain vector by default. Since MPI applications are to be linked by the QMPI interface at the compile-time, each MPI call in the MPI application jumps to the wrapper of the QMPI interface which redirects the execution flow to the first real tool in the vector.

2.4.5 Wrapping of MPI functions

For this design, the wrapper principle is still used, however the key wrapper template is changed as in Listing 2.4:

Listing 2.4: Wrapper template for Intermediate design

```
Int QMPI_X(. . .)
{
    ... Do work ...
    void* function_ptr;
    ++level;
    QMPI_Table_query(tool_vector[level, \
```

```
    mpi_interceptions[<function index>], &function_ptr, mapOfFunctions);
    err=exec_func(function_ptr,level);
    ... Do work ...
    return err;
}
```

The function_ptr instance is used to point to and to call the follow-on routine in line. QMPI_Table_query is a QMPI function that searches the next routine in chain and hooks the function_ptr to it. The parameter mapOfFunctions is the function table of the tool that calls the QMPI_Table_query function. tool_vector is the vector in which the tool instances are put, and level parameter is simply a counter which gets incremented as the calls move down in the layers of the tool_vector. Therefore, it keeps the track of the current layer in the toolstack used in execution.

exec_func routine is another QMPI function which is used to execute the follow-on routine that is pointed by the function_ptr.

2.4.6 Aimed Workflow by the Design

The aimed workflow of initialization phase by the intermediate design is given in Figure 2.5. The start of the flow starts with an MPI_Init call from the MPI application. Then, execution jumps to QMPI_Init wrapper which parses tool paths from the config.txt file. In config.txt file path of each tool is written on a single line. Later, with each tool path QMPI instantiates a dynamic_lib structure and pushes these instances into the vector which represents the toolchain. The order of the tools in the vector is the same as the order of their paths in the config.txt file. In the next, step QMPI loads the tool libraries and passes the handle obtained to a pointer stored in dynamic_lib instances. Get_tool_interceptions method is invoked to get string arrays in each tool which includes list of names of the wrapper routines intercepting MPI functions. According to the read strings from the string arrays, function tables are filled. When a function table of a tool instance is being filled, string array of the next tool in the toolchain is used. This is due to the fact that, function table of each tool must point to functions of the next tool in the toolchain. In case the string in the nth element of the array is NULL, then nth elements of the string arrays of other tools are checked one by one until either the end of the toolchain is reached or a string that is different

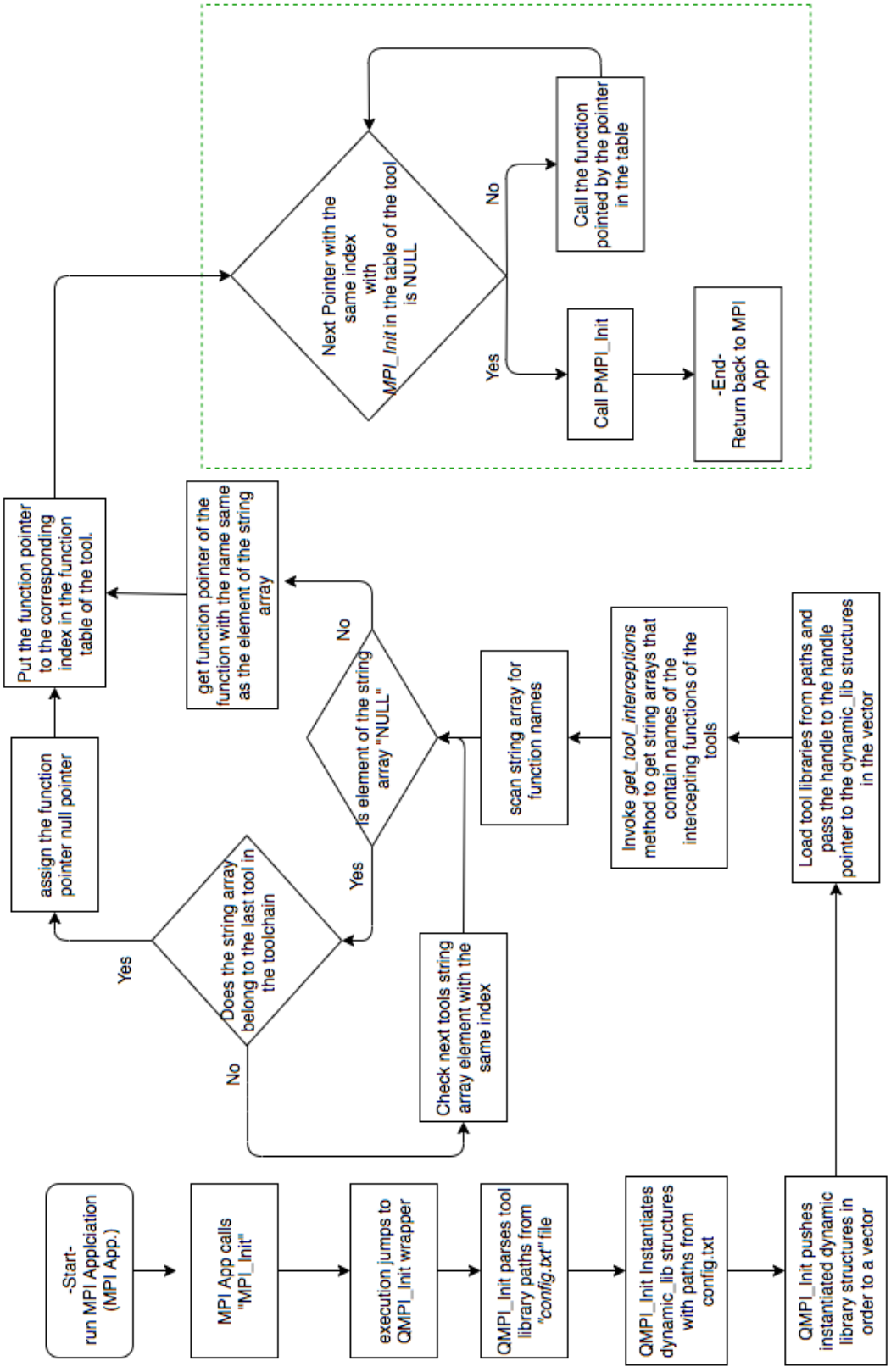


Figure 2.5: Aimed workflow of "MPI_Init" by Intermediate Design

than NULL is found. In case the string in the nthe element of the array is different than NULL, then a pointer to the wrapper routine, with the name same as the string element, is pulled. This pointer is placed into the nth element of the function table.

After function tables are filled toolchain can be followed to complete execution of MPI_Init. Index number of the MPI_Init is 200. Therefore, 200th element of the function table of first instance in the toolchain, QMPI interface by default, is checked. If it is NULL pointer, PMPI_Init is called directly. If it is not NULL pointer then function pointed by this pointer is called. Flow including this decision is marked with the green box in the flow diagram. Even if the flow diagram looks like it represents a loop behavior, in fact it is recursive call of function pointer. At this stage tools recursively call the function pointer to follow-on routines in the 200th element of their function pointers until a null pointer is reached by a tool. If a null pointer is reached PMPI_Init is called.

The general flow of an MPI function call from an MPI application is given in Figure 3.6. When a MPI function other than MPI_Init is called by the MPI application, first execution flow jumps to the corresponding QMPI wrapper. This wrapper directly calls the next pointer in its tables, unless a null pointer is located in the corresponding function table element. If a null point is located in the corresponding element of the function table, then the PMPI function corresponding to the MPI function is called. Tools call wrapper routines, through their function tables, belonging to the next tools in line and these recursive calls continue until a null pointer is reached. When a null pointer is reached PMPI function, corresponding to the MPI function call of the application, is called. As PMPI function call returns the execution flow returns from nested wrapper calls until it reaches back to the application.

2.5 Final Design

Final design is the last design and it is used in the implementation phase.

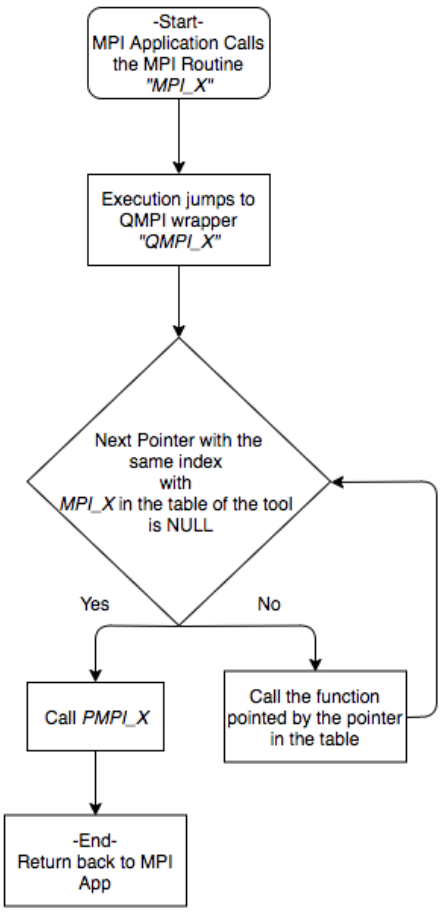


Figure 2.6: Aimed Workflow by the intermediate design

2.5.1 Additional Assumptions for the Design

This design takes over all of the assumptions explained before in Section 3.1. Moreover, in this design following assumptions, made for Intermediate design, have been changed.

Firstly, in intermediate design it was assumed that the design can be made in C++. In this design, the interface is re-designed in C. The vector structure that is in standard C++ library is replaced by a custom-designed data structure. Also, the map structure is replaced by a static array whose elements can be reached with an index value and O(1) complexity.

Secondly, call of function pointers, that are pulled from the function tables, in a tool are re-designed to improve the usability of the tool for the developers. In this design, the usability is assumed to be more important over the performance of the tool.

2.5.2 Tool Responsibilities

In this section, only the tool responsibilities, that are different than the intermediate design, are explained. The other tool responsibilities that are explained in sections 2.3.3 and 2.4.2 but not here, are still valid for this design as well.

(a) **Tool registration:** Tool registration is done by QMPI via an environment variable named TOOLS. The TOOLS environment variable should contain the paths to the dynamic tool libraries separated by colon(:) and ordered in the same order as the user desires the toolchain to form in. For instance, usr/lib/mpiP/libmpiP.so:../../myTool.so value assigned tool TOOLS environment variable will put QMPI tool at the top of the toolchain by default, mpiP tool as the second and myTool as the third tool in the toolstack. Further, these paths are used in order to load libraries before actual "MPI_Init" is called by QMPI.

(b) **Storage Structure:** The tool should instantiate and initialize a structure which includes all global variables whose storage space are wished to be separated. Allocation of the storage space is handled by QMPI whereas, the tool is responsible for defining the data structure to be stored and calling the corresponding QMPI functions to trigger memory space allocation and to get the pointer to the space allocated by QMPI.

(c) **Function Tables:** Tools are no longer required to store their own function tables but call the QMPI routines that returns and executes the function pointers. In this design, since function tables are reached through a QMPI function call, time overhead is a concern. The time overhead of the get function that returns the function pointer to the follow-on routine must be as low as possible. Therefore, a data structure with search complexity O(1) must be used to reduce the overhead of the get function. For this purpose, a void pointer array is used in the implementation.

(d) **Wrappers:** The tools should implement wrappers that are derived from the key wrapper template. The key wrapper template is as in Listing 2.5:

Listing 2.5: wrapper template of final design

```
Int Function_Name_X(<parameters for MPI_x function>,int i,vector * v) {
        ... Do work ...
        void* f = NULL;
        QMPI_Table_query(_MPI_X, &f, (*(VECTOR_GET(v,i))).table )
    int err = exec_func(f, i, _MPI_X, v, <parameters for MPI_X> );
        ... Do work ...
        return err;
}
```

(i) *void* f* : a void pointer to hook to the follow on routine should be instantiated by the tool developer.

(ii) *exec_func* : a QMPI function which calls the routine pointed by the function pointer (f) that is passed to it. Parameters that are passed to exec_func and other than the function pointer, are parameters that are passed to the function to be executed by exec_func

(iii) <parameters for MPI_X function>: Standard parameters required by MPI function wrapped requires should be passed to the exec_func.

(iv) *(*(VECTOR_GET(v,i))).table* : Returns the function table of the toolstack(vector) element with level i.

(v) Function_Name in the signature of the wrapper function should not be QMPI, MPI or PMPI.

This version of the wrapper template leaves the tool developer less work to handle in toolchain side and provides better usability for QMPI.

Furthermore, tools wrapper routines must call their follow-on routines, unless they deliberately intend to break the flow of the toolchain. Exception to this are wrappers for MPI_Pcontrol function. Since wrappers for MPI_Pcontrol is used for

configuration of the profiling behavior of a tool structure and doesnt have a call to MPI_Pcontrol in it , QMPI handles toolchain execution of this function itself. QMPI scans the function tables in the toolchain-vector and calls pointers to these functions itself. Therefore, in the wrapper routines intercepting MPI_Pcontrol there is no need to get function pointers and execute them. Also, QMPI does not support variadic nature of the MPI_Pcontrol. The variadic parameters are ignored and only a single integer input parameter is taken into consideration when implementing QMPI wrapper for Pcontrol.

(e) **qmpi.h:** All tools must include *qmpi.h* header in the file in which their wrappers are defined.

(f) **Extra parameters:** Each wrapper function in the tools should have int i and vector * v as their input parameters. These two parameters should be passed to "*exec_func"* routine and "*VECTOR_GET*" routine of QMPI.

2.5.3 Wrapping of the MPI Functions

QMPI is designed like a tool library with wrappers which can both start and end the toolchain. Therefore, it has two wrapper routines per function in the standard MPI library. Examples for both wrapper functions for *"MPI_Abort"* are shown in Listing 2.6:

Listing 2.6: QMPI wrappers for "MPI Abort"

```
_EXTERN_C_ int QMPI_Abort(MPI_Comm comm, int errorcode, int level \
,vector* v){
       return PMPI_Abort(comm, errorcode);
}
_EXTERN_C_ int MPI_Abort(MPI_Comm comm, int errorcode) {
    void* f = NULL;
    QMPI_Table_query ( _MPI_Abort, &f, (*VECTOR_GET(&v, 0)).table );
    int ret= exec_func (f,0,_MPI_Abort,&v,comm,errorcode);
    return ret;
}
```

The wrapper functions with their names starting with QMPI are called at the very

end of the toolchain. They are wrappers for PMPI routines and provides consistency in the input parameters of all of the follow-on routines in the toolchain. Function pointers to these functions are placed in the function tables such that they are the very last functions to be called in the toolchain. As these functions return, the flow of execution begins to move upper layers of the chain until it reaches the MPI application.

QMPI has wrapper functions starting with MPI prefix. Since all MPI functions are declared as weak symbols, by statically linking QMPI library to MPI application, all MPI calls in the application are linked to these functions with MPI prefix of the QMPI library. The wrapper functions with names starting with MPI are the functions that are designed to make a call to a tool routine. These functions trigger the flow of execution in the toolchain. Their structure is just like any other QMPI compatible tool. The i parameter in the tool wrapper example-section 2.5.2 item f - are replaced with constant value of zero, since these functions are the very top level of the toolchain by design.

2.5.4 Function Tables

Function tables build-up of structures that can hold both function pointers to follow on routines and an integer value indicating the next level in the toolstack to be jumped after execution of the function pointed by the pointer. The level information in the function table elements enables calling other MPI functions than the intercepted MPI routines by the wrapper routine. Hence, for instance when MPI_Comm_rank function is called in a wrapper routine for intercepting MPI_Send function, a pointer to a follow-on routine for MPI_Comm_rank function from the function table of the same tool is called. Execution flows according to a subset of the toolchain. The subset starts from the level of the tool which made MPI_Comm_rank function call and ends at the end level of the toolchain. An example flow of execution is shown in Figure 2.8.

2.5.5 Context Pointers

Context pointers are stored in tool instances- "dynamic_lib" structures - in the vector and a single context pointer per tool instance is used.

This design gives tool developer the flexibility to choose whether or not to separate the context- variable and data storage space- of different executions of the same executable file. In case the tool developer decides to separate the storage space of two different executions of an executable, then the developer is responsible of declaring and defining a single data structure that will include all of the variables and data to be used privately by one execution of the tool. For example, mpiP is a tool that counts number of some MPI calls in an MPI application. It contains wrappers for most of the MPI functions and increments respective global counter variables of different MPI functions, each time their wrapper at mpiP is called. When there is a single mpiP executable file, whose path is the first and the last elements of the TOOLS environment variable, mpiP will generate a single report with results showing double the actual number of MPI functions in the MPI application. To prevent this behavior and get separate reports with right MPI function counts, it is required to separate both, the memory space, in which global variables counting number of MPI functions, are stored and the memory in which reports are written. Memory space for other global variables, that are read only or that do not affect the output, need not to be separated. Hence, double copies of unnecessary data are avoided.

The tool should instantiate and initialize a data structure which includes all global variables that are wished to be separated. By calling a QMPI function and passing the structure to the QMPI a separate storage space with the structure, same as the instantiated structure, is allocated. Therefore, allocation of storage space is handled by QMPI, whereas the tool is responsible of declaring and defining the data structure for the storage, then calling the corresponding QMPI functions to trigger memory allocation and to get the pointer to the space allocated by QMPI.

By this design, the developer may also adopt a hybrid approach at which some global variables are separated in different contexts, used alongside some other global variables that are unseparated in different contexts. This feature avoids copies of the same data that is not changed or that has no effect on the result of the computations, therefore consumption of unnecessary memory space.

2.5.6 Aimed Workflow by the Design

In Figure 2.7 the flow of the program, until the end of execution of MPI_Init is illustrated. The flow diagram of the final design is very similar to intermediate design. In the flow until the first decision operation placement of function pointer in the function tables- only difference between two designs is that in the final design QMPI reads the tool paths from the TOOLS environment variable. Therefore, the flow of the interface until the first decision first decision box, is not explained.

In the first decision block after the string from the string array is obtained, QMPI checks whether the obtained string is NULL or not. If the string is different than NULL then function pointer to this function is obtained and placed in the function table. If it is NULL and the tool, from which the string is obtained, is not the last tool in the toolchain then, the string - from the string array of the next tool - with the index corresponding MPI_Init is compared with NULL. In case the tool is the last tool of the toolchain then a pointer to PMPI_Init is placed in the function table. After function tables are set, QMPI gets the first function pointer in the function table of the first instance in the toolchain. Later, a function pointer type same with the obtained function pointer is declared. The obtained function pointer is casted to this type and executed. By this means, the toolchain is triggered. The first tool wrapper function executed checks its function table and gets the next function pointer to be executed. Then calls the function through the function pointer and this recursive calls between tool wrappers continue until a function pointer to PMPI_Init is obtained. After PMPI_Init is executed recursive wrapper calls return until the MPI application is reached. The part of the flow diagram in the green box is regular toolchain execution and is the same for all of the MPI functions after an MPI application calls an MPI function. This flow of the toolchain for all MPI functions is also illustrated in Figure 2.8. The paragraph above explaining the flow of the program in the green box in Figure 2.6, is specialized form of the flow in Figure 2.8 therefore, it is used to explain flow in the Figure 2.8.

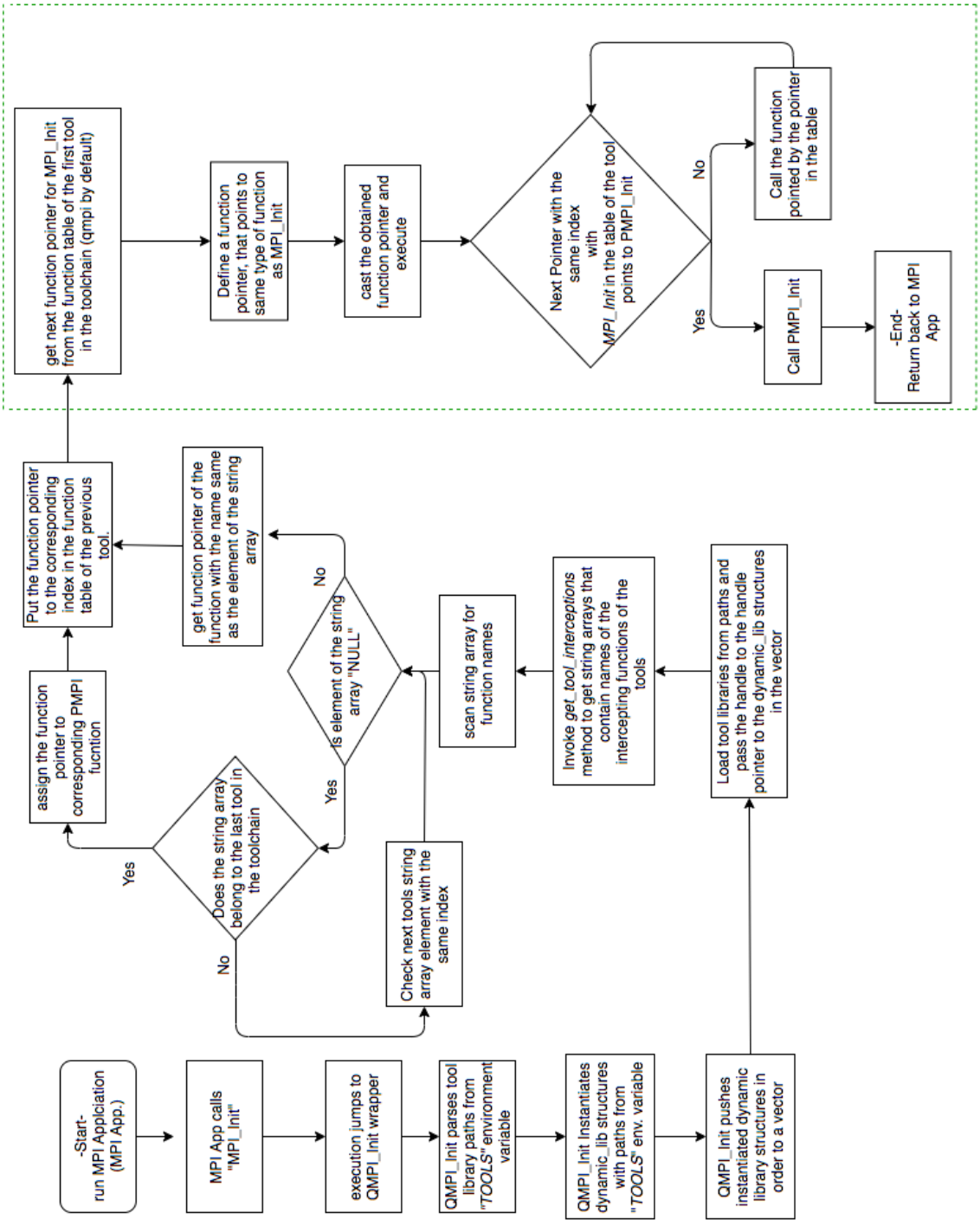


Figure 2.7: Aimed initialization workflow by the Final design

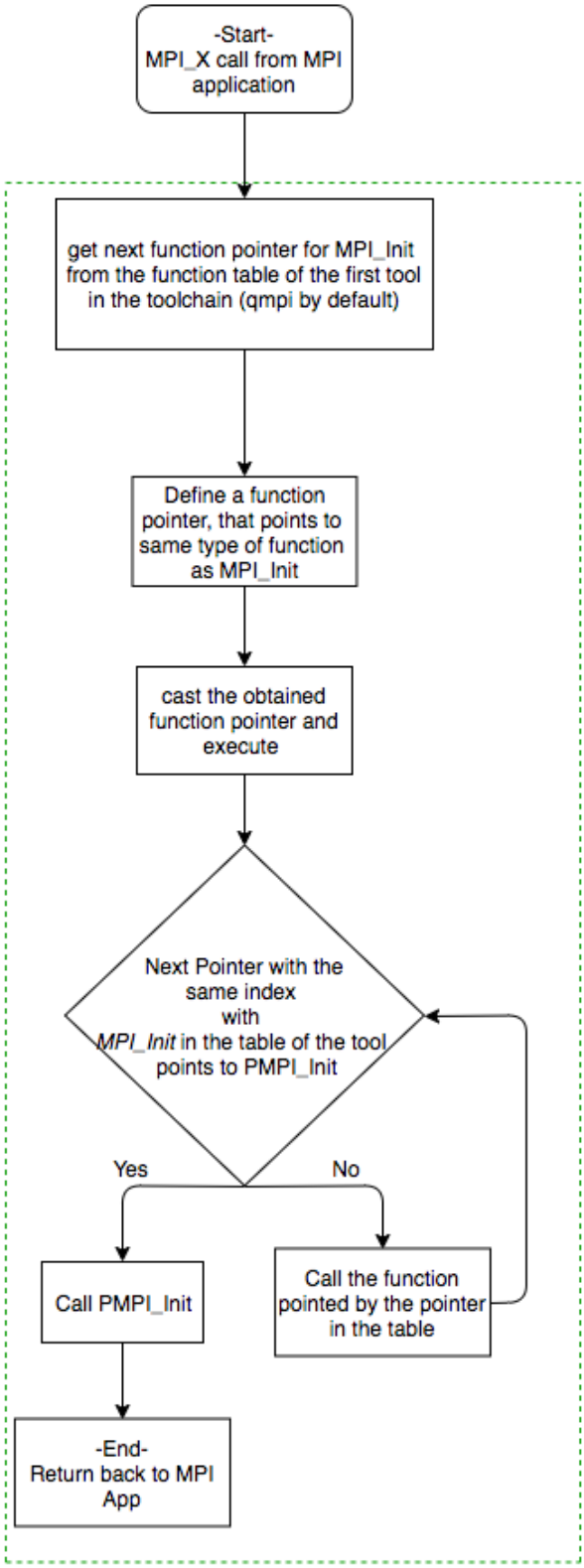


Figure 2.8: Aimed workflow by the Final design

3 Implementation

In this section details of the implementation of the final design are explained.

3.1 Dynamic Linking

Due to both requirements and assumptions of the final design, the tools are compiled as dynamic libraries. Therefore, QMPI needs to load dynamic tool libraries and get function pointers to routines in the tools. Loading of the dynamic libraries is done in "MPI_Init" wrapper of QMPI. This is done via a routine called load_lib. The code for this function is as in Listing 3.1:

Listing 3.1: Dynamic linking - loading tool libray- implementation

```
dynamic_lib_handle load_lib (const char* path) {
    void* tmp = dlopen( path , RTLD_NOW);
    // Error checks
    const char *dlsym_error = dlerror();
    if (dlsym_error) {
        printf("Cannot load library: ", dlsym_error);
    }
    return tmp;
}
```

The path variables passed to the "load_lib" function, are extracted from TOOLS environment variable. In MPI_Init before tool libraries are loaded, TOOLS environment variable is read and split from the : characters. The resulting paths obtained from the TOOLS variable are used to load the dynamic tool libraries. After the tool libraries are loaded the function pointer to the routine in the tool, called get_interceptions, is

called to get the names of the wrapper routines that intercept standard MPI functions. The index numbers of the function names are used to differentiate which wrapper routine intercepts which MPI function. The string has as many elements as the number of standard MPI functions and each element corresponds to an MPI function. In case the tool has a wrapper routine that intercepts a MPI function, the name of the wrapper is written into the corresponding element of the string array. In case tool does not have a wrapper that intercepts a certain MPI function then the string NULL is placed in the corresponding element of the string array.

By using the elements of the string array, QMPI gets the pointers to the wrapper routines in the tools by the help of get_func_ptr function. Implementation of the get_func_ptr function is given in Listing 3.2:

Listing 3.2: Dynamic linking implementation

```
void* get_func_ptr(const dynamic_lib_handle handle, char* func_name){

    if (handle == NULL) {
        printf("handle null, returns for \n");
        return NULL ;
    }
    void* status = dlsym(handle , func_name);
    // Error checks
    const char *dlsym_error = dlerror();
    if (dlsym_error) {
        printf("Cannot load pointer: %s \n ", dlsym_error); }
        if (status == NULL) {
            printf("function pointer in get_func_ptr = NULL \n");
        }
    }
    return status;
}
```

3.2 QMPI

3.2.1 Data Structures

Environment variable TOOLS

TOOLS environment variable must include the paths of the tools that are desired to be in the toolstack. The order of the paths of the tools is the same order of them in the toolstack. Moreover, tool paths must be separated by : character. In the wrapper routine of QMPI intercepting "MPI_Init", QMPI reads TOOLS environment variable and extracts tool paths by dividing the rad string from : characters. Later uses these extracted paths to dynamically load tool libraries.

Dynamic_lib structure

This structure is used to store variables private to a tool. Its declaration is given in Listing 3.3.

Listing 3.3: Tool structure - "dynamic_lib" - implementation

```
// NUM_MPI_FUNCS : Number of standard MPI functions

struct dynamic_lib {
    // Path of the tool read from TOOLS environment variable
    char path[100];

    // Assigns handle of the tool when it's loaded by load_lib function
    dynamic_lib_handle handle;

    // Includes names of the wrappers of MPI_functions
    char*   mpi_interceptions[NUM_MPI_FUNCS];

    // 1-dimension function table of the tool
    cell table[NUM_MPI_FUNCS];

    // Pointer to the memory space to store global variables
    void* context;
};
```

Toolchain Vector

It is a dynamically allocated array like structure which is implemented especially for the QMPI interface. It stores dynamic_lib type of structures which is the type of instances that are instantiated per tool in the toolchain. The type declaration of the structure is given in Listing 3.4:

Listing 3.4: Toolchain vector implementation

```
typedef struct vector {
    // Points to the vector elements
    struct dynamic_lib **items;

    // Maximum number of elements
    int capacity;

    // Number of elements that are already stored
    int total;
} vector;
```

There are some functions used to manage this vector structure and they are given in Listing 3.5.

Listing 3.5: Helper functions for toolchain vector

```
struct dynamic_lib {
    // Dynamically allocates memory using the initial size of vector
    void vector_init(vector *v)

    // Returns number of elements found in vector
    int vector_total(vector *v)

    // Increases size of memory space allocated for the vector
    static void vector_resize(vector *v, int capacity)

    // Adds new items to the vector
    void vector_add(vector *v, struct dynamic_lib *item)
```

```
    // Returns a pointer to an element of the vector
    // found at the given index.
    struct dynamic_lib *vector_get(vector *v, int index)

    // Frees allocated memory space for the vector
    void vector_free(vector *v)
};
```

Function tables

Each dynamc_lib instance in the toolstack vector has a one-dimensional function table. Function tables are arrays of cell structures and they have as many elements as number of standard MPI functions. Implementation of this cell structure is given in Listing 3.6.

Listing 3.6: Helper functions for toolchain vector

```
typedef struct cell {
        void* func_ptr;
        int level;
} cell;
```

Void pointer variable is the function pointer to the next follow-on routine and integer variable level holds the level of the tool in the toolchain- from which the function that is pointed is. This level information helps switching between levels, tables, of the toolstack rather than a sequential flow between the tables.

In MPI_Init wrapper of QMPI, after the vector for toolchain is declared and initialized with the instances of dynamic_lib structures- created with each tool path obtained from TOOLS variable. Then, wrapper routine names in the tools are obtained from the tools and QMPI gets function pointers to those wrapper routines. These pointers are placed in the function tables in an order so that toolchain will be achieved as they are called successively and there should not be any holes in the function tables. In the following examples how, function tables are filled by QMPI are illustrated for different cases of tool wrappers. Assume that the user will use three different tools (called T1, T2, T3). These tools will be put in function tables for different use-cases given in Figure 3.1.
In case (a) T3 has wrapper for MPI_Send and other tools do not have.

In case (b) T2 has wrapper for MPI_Send and other tools do not have.
In case (c) T1 has wrapper for MPI_Send and other tools do not have.
In case (d) T1 and T3 have wrappers for MPI_Send and T2 do not have.
In case (e) T2 and T3 have wrappers for MPI_Send and T1 do not have.
In case (f) T1 and T2 have wrappers for MPI_Send and T3 do not have.
In case (g) all tools have wrappers for MPI_Send.
In case (h) none of the tools has wrappers for MPI_Send.

This way of filling the function tables prevents empty cells in these tables. These empty cells prevent wrapper functions to call other MPI routines than the ones they are designed to intercept.

Furthermore, when there are no tools in the toolchain, the table of QMPI instance is not filled to reduce the time overhead.

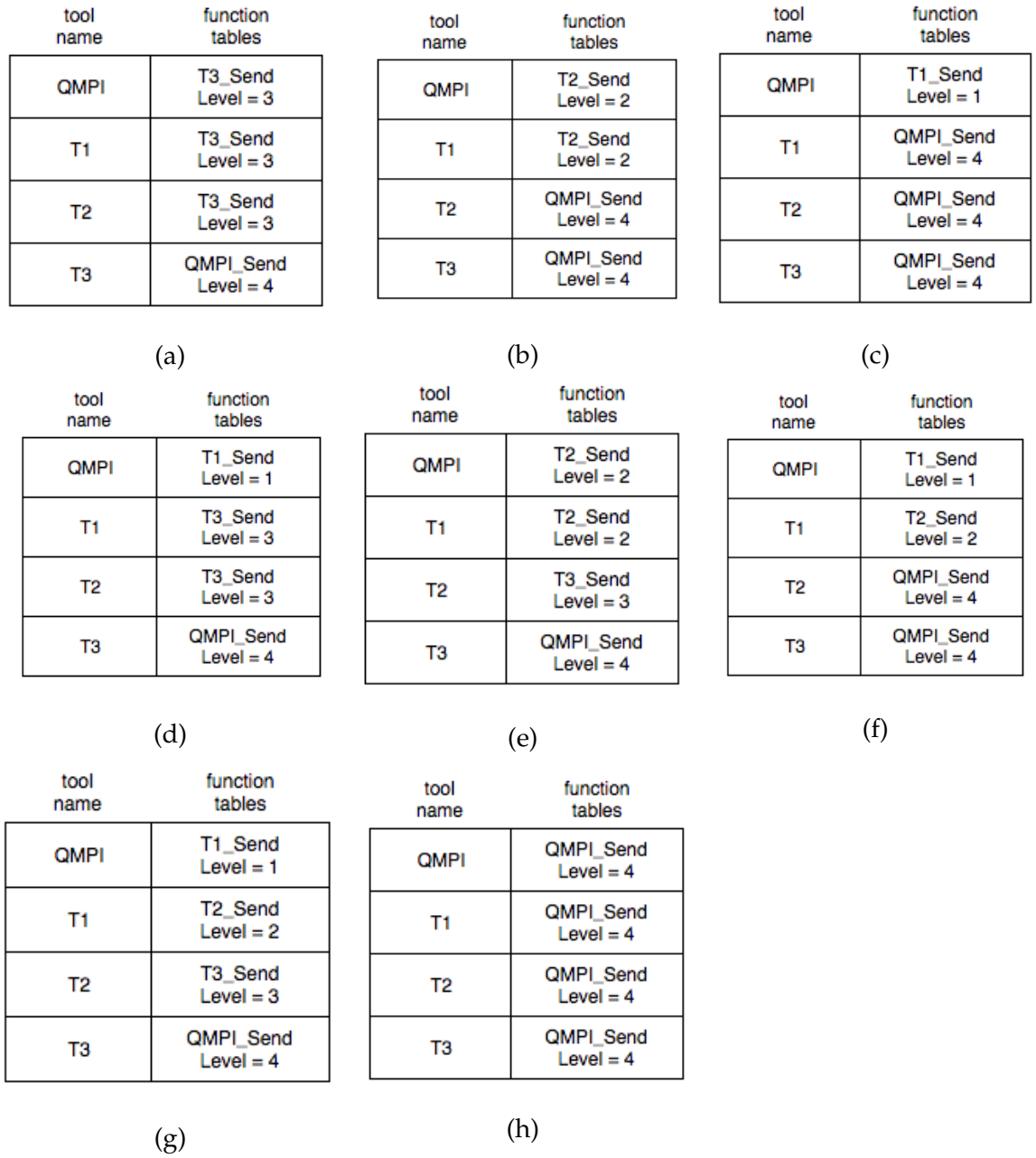
Enumeration of the wrapper routines indices

In order to differentiate which wrapper in a tool is written to intercept which standard MPI routine, enumeration of standard MPI routine names is used. This enumeration of MPI routines is done by sorting MPI routines in alphabetical order and assigning them their sequence number in the order. The names assigned to integer values are the names of MPI functions with an underscore at the beginning, e.g. _MPI_Abort. Full enumeration chart can be found in the appendix.

The order of the function pointers in the function tables is according to this enumeration. Therefore when QMPI_Table_query function attempts finds a function pointer in function tables it goes directly to the integer index which is passed to QMPI_Table_query function as enum. For instance, when QMPI_Table_query is called by the line *"QMPI_Table_query (_MPI_Abort, &f, (*VECTOR_GET(&v, 0)).table)*, the element found in the first index of the function table is returned, since _MPI_Abort enum name corresponds to zero.

3.2.2 QMPI Special Functions

QMPI contains special functions that can be used by tools to perform the functionality of the QMPI interface.



(a)

tool name	function tables
QMPI	T3_Send Level = 3
T1	T3_Send Level = 3
T2	T3_Send Level = 3
T3	QMPI_Send Level = 4

(b)

tool name	function tables
QMPI	T2_Send Level = 2
T1	T2_Send Level = 2
T2	QMPI_Send Level = 4
T3	QMPI_Send Level = 4

(c)

tool name	function tables
QMPI	T1_Send Level = 1
T1	QMPI_Send Level = 4
T2	QMPI_Send Level = 4
T3	QMPI_Send Level = 4

(d)

tool name	function tables
QMPI	T1_Send Level = 1
T1	T3_Send Level = 3
T2	T3_Send Level = 3
T3	QMPI_Send Level = 4

(e)

tool name	function tables
QMPI	T2_Send Level = 2
T1	T2_Send Level = 2
T2	T3_Send Level = 3
T3	QMPI_Send Level = 4

(f)

tool name	function tables
QMPI	T1_Send Level = 1
T1	T2_Send Level = 2
T2	QMPI_Send Level = 4
T3	QMPI_Send Level = 4

(g)

tool name	function tables
QMPI	T1_Send Level = 1
T1	T2_Send Level = 2
T2	T3_Send Level = 3
T3	QMPI_Send Level = 4

(h)

tool name	function tables
QMPI	QMPI_Send Level = 4
T1	QMPI_Send Level = 4
T2	QMPI_Send Level = 4
T3	QMPI_Send Level = 4

Figure 3.1: Use-cases for function tables with 3 tools

QMPI_Set_context This function is used to set the storage space context per element of the vector that constitutes the toolchain. It may or may not be used depending on the tool developers intentions. This function copies the structure passed to it, into a dynamically allocated memory space and hooks the context pointer of the tool, that called the QMPI_Set_context . The implementation of the function is given in Listing 3.7.

Listing 3.7: Implementation of QMPI_Set_context

```
void QMPI_Set_context(vector *v, int index, void* cntxt, size_t size)
{
    if (index >= 0 && index < v->total) {
        (*VECTOR_GET(v,index)).context = malloc(size);
        memcpy((*VECTOR_GET(v,index)).context, cntxt, size);
    } else{
        printf("Invalid level of tool, context not found \n");
    }
    return;
}
```

A tools wrapper signature must be *int Function_Name_X(<parameters for MPI_x function>, int i, vector *v)*.The level information of a tool is passed to its wrappers by *int i* parameter. The *vector * v* parameter can be directly passed to the QMPI_Set_context function.

void cntxt* is a pointer to a structure, that is declared, defined and instantiated by the tool developer. This structure must include all of the global variables that are desired to be separated per each tool instance in the vector that constitutes toolchain.

size_t size is the size -in bytes- of the structure that contains global variables to be separated. There is also a macro implementation of this function which has exactly same functionality and parameters. It can be called by name QMPI_SET_CONTEXT.

QMPI_Get_context

This function, whose implementation is given in Listing 3.8, can be used to get the context pointer.

Listing 3.8: Implementation of QMPI_Set_context

```
void* QMPI_Get_context(vector* v, int i ) {
    return ((*VECTOR_GET(v,i)).context);
}
```

QMPI_Get_context function returns a void pointer. The returned void pointer must correspond to the type casted by the tool. Its type is the same with the structure declared before, in order to collect all global variables in, and passed to QMPI_Set_context function. There is also a macro implementation of this function which has exactly same functionality and parameters. It can be called by name *QMPI_GET_CONTEXT*.

QMPI_Table_query *QMPI_Table_query*, whose implementation is given in Listing 3.9, returns function pointers from the function tables. Its use is essential to keep the flow of the program in toolchain structure.

Listing 3.9: Implementation of QMPI_Set_context

```
int QMPI_Table_query(_MPI_funcs idx, void** func_ptr, cell table[])
{
    // if there exists a follow-on routine of another tool returns 0
    // and sets the function ptr

    *func_ptr = (table[idx]).func_ptr;
    if (*func_ptr== NULL)
    {
        printf("No tools in the toolchain, redirecting to PMPI");
        *func_ptr = QMPI_Array[idx];
        return 1;
    }
    return 0;
}
```

MPI_funcs idx is either integer or enum name that corresponds to the MPI function to be called. *void** *func_ptr* is the function pointer to be hooked with the function pointer from the function tables. *cell table[]* is a function table obtained from a *dynamic_lib* instance of the vector. In case the toolstack is empty, i.e. there are no tools to be used and execution is a simple MPI application, the only "*dynamic_lib*" instance

in the toolchain vector is QMPI, which is always the first element of the vector. When an MPI function call is made from the application the wrapper functions with MPI prefix in the QMPI library is called. As a tool wrapper would do, wrapper functions of QMPI , with MPI prefix, call "QMPI_Table_query" function. In case there are no tools in the toolchain, "QMPI_Table_query" calls in the MPI functions return function pointers to the PMPI function wrappers of QMPI. By this means, an overhead of filling the function table of QMPI instance in the vector is avoided.

exec_func

exec_func is a function which executes the function pointed by the pointer passed to it. Moreover, it gets - from function tables - and sets the next level to be jumped , then passes it to the function it executes. Its declaration is as follows: int exec_func(void* func_ptr,int level ,_MPI_funcs func_index, vector* v , ...)

void* func_ptr is the function pointer to the follow-on routine that is to be executed. int level parameter is added to the parameters of the wrapper functions of the tools. It holds the level information of the tool by which exec_func is called._MPI_funcs func_index is the enum to the MPI function whose wrapper in a tool is pointed by the function pointer passed by void* func_ptr.vector* v is a parameter that is added extra to the parameters of the wrapper functions of the tools. It can be directly passed to exec_func.

As it can be understood from its declaration, exec_func is a variadic function. The variadic parameters are the input parameters of any MPI function, that is wrapped by the pointed function by void* func_ptr. exec_func is used to execute function pointers that are obtained by QMPI_Table_query. Even if the functionality provided by exec_func causes some considerable amount of time overhead and it can be implemented by the tool developer in the wrappers with lower time overhead, the use of this function provides better usability of the interface.

In cases, at which performance is a crucial priority, tool developer has to implement the functionality of exec_func function for each wrapper. This functionality includes getting the level information from the function table cell, in which function pointer to be executed is located. This can be achieved by QMPI_Get_level function whose declaration is as follows: int QMPI_Get_level(int level,_MPI_funcs func_index, vector* v) _MPI_funcs func_index is enum name or integer value that corresponds to the MPI function whose wrapper functions are intended to be called.

QMPI_Get_level is also implemented as a macro function. It can be used with the same parameters and has the same return value. It can be called by using the name QMPI_GET_LEVEL. The new level value obtained from QMPI_Get_level should be passed to the follow-on routine. Since all function pointers are kept as void pointers in the function tables, they must be casted to their function types before calling the functions they point to.

This kind of implementation of tool wrappers, without using exec_func, requires some knowledge on how QMPI interface functions and is more cumbersome for the tool developer. However, in case performance is the main and the most crucial goal of the toolchain, this approach can be useful. An example of a tool wrapper implemented for MPI_Comm_rank by this approach is given in Listing 3.10.

Listing 3.10: Implementation of QMPI_Set_context

```
_EXTERN_C_ int T1_Comm_rank(MPI_Comm comm, int *rank,int i,\
vector *v){
    printf("T1_Comm_rank \n");
    void* f = NULL;
    QMPI_TABLE_QUERY(_MPI_Comm_rank,&f,(*VECTOR_GET(v,i)).table);
    int new_level=QMPI_Get_level(i,_MPI_Comm_rank, v);
    typedef int (*comm_rank_func)(MPI_Comm comm,int *rank,int i, \
    vector* v);
    int ret = ( (comm_rank_func) f ) ( comm, rank, new_level, v);
    return ret;
}
```

MPI_Pcontrol wrapper

Among all of functions of the standard MPI library MPI_Pcontrol is routine that needs special handling of its wrappers. MPI_Pcontrol is used to adjust the profiling features at the runtime and its wrappers does not need a call to MPI_Pcontrol. Therefore, MPI_Pcontrol wrapper of QMPI scans through the function tables to find which tools intercept the MPI_Pcontrol function and calls those functions one by one itself, rather than enabling a flow between tools without QMPI intervention. Therefore, unlike other wrapper routines, in wrappers of MPI_Pcontrol there should not be any QMPI_Table_query or exec_func calls to their follow-on routines. Listing 3.11 shows QMPI wrappers for "MPI_Pcontrol".

Listing 3.11: Implementation of QMPI_Set_context

```
extern int QMPI_Pcontrol(const int level, int level, vector* v) {
  return PMPI_Pcontrol(level);
}

_EXTERN_C_ int MPI_Pcontrol(const int level, ...)
{
  int i =0,ret=0;
  typedef int (*pcontrol_func) (const int first_level, int level,\
  vector* v);

  for ( ; i < vector_total(&v) - 1;)
  {
     ( (pcontrol_func) ((*VECTOR_GET(&v,i)).table[_MPI_Pcontrol].\
     func_ptr) )(level, i, &v);
     i= ( (*VECTOR_GET(&v,i)).table[_MPI_Pcontrol].level );
  }
  return ret;
}
```

Also, variadic parameters of MPI_Pcontrol is ignored since it is almost impossible to implement wrappers for variadic functions. Therefore, only the integer input parameter of MPI_Pcontrol is taken into consideration when QMPI wrappers are implemented.

3.3 Tool Implemetation

In order to write a QMPI compatible tool first thing, that must be done is to include qmpi.h header in the files at which wrapper functions for MPI routines will be declared and defined.

Second, each tool must declare and define a string array that has 360 elements-number of standard MPI_functions. Then names of the wrapper routines in the tool should be placed in this string array as strings. Which wrapper name should be placed in which element of the array can be found by the enumeration table in the

appendix. The string NULL must be placed in to the elements in the string array, that correspond to MPI functions, which are not intercepted by the tool.

If context separation of a tool is not a concern, then variables can be declared and used same way as in a PMPI compatible tool. If context separation is desired then a data structure, which will include all of the global variables, that affect the results of the tool, must be declared. Later, this data structure must be defined and passed to QMPI_Set_context, before the variables in the data structures are used. Each time a variable - that is among the global variables included by the structure and stored in the separate context - is to be used, QMPI_Get_context function must be used. QMPI_Get_context returns a void pointer which points to the separated context. Before using this pointer, it must be casted to the data structure, which is declared by the developer before.

Wrapper routines in a tool are not very different than the wrappers of a PMPI compatible tool. Therefore, it is very easy to refactor a PMPI compatible tool into a QMPI compatible one. Developer must only replace MPI calls in the wrappers of a PMPI compatible tool, with 3 lines of code.

In First line a void pointer must be declared. In the second line function pointer that points to the follow-on routine must be queried. This can be done through QMPI_Table_query function of the QMPI. The void pointer that is declared in the first line must be passed to QMPI_Table_query function and QMPI_Table_query function hooks this void pointer with the follow-on routine. In the third line exec_func must be called to execute the function pointed by the void pointer - declared in the first line.

The examples of implemented tools for QMPI can be found in the following sections.

3.3.1 Simple Tools

In order to test the QMPI interface prototype some simple tools, that only print some information, are implemented. These simple tools may be very basic examples of how to write tools for QMPI interface prototype. An example wrapper that intercepts MPI_Init, from one of these tools is as in Listing 3.12:

Listing 3.12: Implementation of Simple tool wrapper

```
_EXTERN_C_ int T1_Init(int *argc,char ***argv,int i,vector * v){
        printf("T1_Init ! Lvl: %d\n", i);
        int ret =1;
        void* f=NULL;
        QMPI_TABLE_QUERY(200,&f,(*VECTOR_GET(v,i)).table);
        ret= EXEC_FUNC(f,i,200,v,argc,argv);
        printf("Return 1 !\n");
        return ret;
}
```

Full implementation of the most comprehensive of these tools can be found in section 7.2 of the appendix.

3.3.2 Tool with Custom Broadcast implementation

In order to test the calls of different MPI functions, then the one the wrapper routine is designed to wrap, a simple tool that has its own implementation of "MPI_Bcast" is implemented.

By calling wrapper routines pointed by the function pointers in the function tables for MPI_Comm_rank, "MPI_Comm_size, MPI_Send and MPI_Recv, of the next tool, the broadcast behavior that is wished to be investigated is achieved.

Implementation of the broadcast wrapper routine of this tool is given in Listing 3.13.

Listing 3.13: Implementation of Simple tool wrapper

```
_EXTERN_C_ int TB_Bcast( void *buffer,int count,MPI_Datatype datatype, \
int root,MPI_Comm comm,int i, vector* v) {
    int ret = 1;
    void* f = NULL;
    int rank = 0;

    // MPI_Comm_rank wrapper call
    QMPI_TABLE_QUERY(_MPI_Comm_rank, &f, ( *(VECTOR_GET(v,i) ) ).table);
```

```
    EXEC_FUNC(f, i, _MPI_Comm_rank, v, MPI_COMM_WORLD,&rank);

    int comm_size = 0;
    // MPI_Comm_size call
    QMPI_TABLE_QUERY(_MPI_Comm_size,&f,(*(VECTOR_GET(v,i))).table);
    EXEC_FUNC(f, i, _MPI_Comm_size, v, comm, &comm_size);

    if (rank == root) {
       for ( int index =0;index < comm_size; index++) {
           if(index != root) {
               MPI_TABLE_QUERY(_MPI_Send,&f,(*(VECTOR_GET(v,i))).table);
               ret = EXEC_FUNC(f,i,_MPI_Send,v,buffer, count, datatype\
               ,index, 0, comm);
           }
       }
    } else {
        MPI_TABLE_QUERY(_MPI_Recv,&f,(*(VECTOR_GET(v,i))).table);
        ret = EXEC_FUNC(f,i,_MPI_Recv,v,buffer, count, datatype, root,0\
        ,comm, MPI_STATUS_IGNORE);
    }
    return ret;
}
```

The full implementation of the tool can be found in Section 7.3 of the appendix.

3.3.3 MPIP refactoring for QMPI compatibility

MpiP is a tool that counts number of calls that have been made to some major MPI functions, during the execution of the program[8]. It also measures of the execution of these MPI functions and the total time application time. It also measures the total application time and calculates the ratio of time spent on MPI routines to application time.

Within this work, MpiP source code is re-factored to make it is compatible with QMPI. This refactoring is both useful in tests and in proving that QMPI can be used

with a more complex tool than the simple tools implemented.

Wrapper names starting with MPI are changed so that they start with TMPI. Also, instead of PMPI function calls QMPI_Table_query and exec_fuc are placed so that pointers to follow-on routines are pulled from the tables and executed accordingly.

Additional data structure, which can be found in Section 7.4 of the appendix, is defined and implemented in order to provide the context separation.

The variables gathered in the data structure is global variables that have effect on the result of the tool. Other variables, such as variables defining report format-i.e. read only variables, are not included in the data structure since they do not have any effect on the computations of the tool.

Later, this structure is instantiated and by QMPI_Set_context and its memory space is allocated accordingly. Whenever a variable- included in the structure- is to be used in a routine, context pointer is obtained by calling QMPI_Get_context.

Further, qmpi.h file is included in the files that includes wrappers- wrappers.c, wrappers_special.c,- and that uses variables or calls functions declared in qmpi.h- mpiPi.h, mpiP-hash.h. One of the refactored mpiP wrappers is given in the Appendix as an exampleto better explain how the refactoring is made.

4 Evaluation

4.1 Test Environment

All the experiments are conducted on SuperMUC, the high-end supercomputer at the Leibniz-Rechenzentrum (Leibniz Supercomputing Centre). For the experimentation purposes, only Phase 2 of SuperMUC is used. Phase 2 constitutes of 6 islands, each contain 512 nodes and all compute nodes within an individual Island are connected via a fully non-blocking Infiniband network (FDR14 for the Haswell nodes of Phase 2) [12]. Nodes of Phase 2 have 2.6 GHz Haswell Xeon E5-2697 v3 processors with 2 processors in each node and each processor has 14 cores, i.e. 28 cores per node [12]. The system uses SUSE Linux Enterprise Server 11-SP4 as the operating system [2]. In addition, the codes were compiled with Intel/17.0 compiler and linked against OpenMPI 2.0 Intel version. All tests were run on a single node of phase 2 with 28 processes.

For the application for overhead and tool layer increment tests Asc Sequoia AMG2006 and HPCG benchmarks, which will be explained in this section, were used. A simple application that calls only MPI_Init, MPI_Finalize and MPI_Bcast is used for Broadcast re-implementation test.

4.2 ASC SEQUOIA AMG 2006 Benchmark

ASC SEQUOIA AMG 2006 is a parallel algebraic multigrid solver for linear systems, which stem from problems on unstructured grids. It is a SPMD (single program,

[2] More information related to the test environment can be found at https://www.lrz.de/services/compute/supermuc/systemdescription/

multiple data) application which achieves parallelism by decomposing the data. Data decomposition is done by subdividing the grid into P x Q x R (in 3D) chunks of equal size

As the problem size increases, majority of the execution itme is spent on MPI operations, i.e. about 90% of the total time for large problems. Collective calls consume more than 90% of the time spent on MPI executions [1].

Names and numbers of the MPI calls for an execution with 28 processes are listed in Table 4.1.

Function	**Count**
Allgather	280
Allgatherv	140
Allreduce	2464
Irecv	23330
Isend	23330
Waitall	9800

Table 4.1: Names and numbers of MPI calls in AMG 2006 Benchmark

Even if AMG2006 benchmark has OpenMP support, it is not used to conduct experiments. For experimentation one of the test problems that solves 3D Laplace equation [3]

$$-cxu_{xx} - cyu_{yy} - czu_{zz} = (1/h)^2 \tag{4.1}$$

with Dirichlet boundary conditions of $u = 0$ is used. h in the formula represents mesh spacing in each direction of the unit cube.

[3] README can be found at
https://asc.llnl.gov/sequoia/benchmarks/AMG2006_v1.0.tar.gz

4.3 High-Performance Conjugate Gradients (HPCG) Benchmark

The High-Performance Conjugate Gradient (HPCG) benchmark is a program that generates a synthetic sparse linear system that is similar to finite difference discretization of a 3D heat diffusion problem with zero drichlet boundary conditions. [2]

The HPC Conjugate Gradient (HPCG) benchmark measures the performance of HPC platforms in terms of memory access and global communication, which are frequently observed patterns of executions. It achieves these performance measurements by using a Preconditioned Conjugate Gradient (PCG) algorithm. The PCG implementation uses a regular 27-point stencil discretization in 3 dimensions of an elliptic Partial Differential Equation (PDE). [4]

The global domain dimension of the model problem are

$$(n_x \ X \ np_x)(n_y \ X \ np_y)(n_z \ X \ np_z)$$

where

$$(n_x * n_y * n_z)$$

are the local sub-grid dimensions assigned to each MPI process and the dimensions

$$(np_x * np_y * np_z)$$

are a factoring of the MPI process space that is computed automatically in the HPCG setup phase. The 3D domain is scaled to fill a 3D virtual process grid of all available MPI process ranks.[2]

HPCG application calls a small portion of MPI routines with limited message sizes. Even if the variety of the MPI calls is limited, number of MPI calls is quite high. Names and numbers of the MPI calls for an execution with 28 processes are listed in Table 4.2 [5].

HPCG implementation is written in C++ with MPI and OpenMP support, however OpenMP support is not used for the conducted experiments.

Function	Count
Allreduce	15540
Bcast	28
Irecv	485208
Isend	485208
Waitall	485208

Table 4.2: Names and numbers of MPI calls in HPCG Benchmark

4.4 MpiP tool

MpiP is a profiling library for MPI applications, that is used for Broadcast Re-implementation tool test. It can report statistical information, such as number of calls to a certain MPI routine or time spent on those routines, related to MPI calls in an application.

The set of routines analyzed by MpiP covers 114 MPI functions therefore it does not cover all of the routines in a standard MPI library. However, since MPI_Bcast, MPI_Send and MPI_Recv are included in this set of analyzed MPI routines, the tool is sufficient for showing how the statistical information related to these three routines is affected before and after the execution of a special broadcast tool.

As described in Section 3.3.3, MpiP is re-factored in order to make it compatible with QMPI prototype and provides context separation of variables- which is a feature that QMPI prototype provides. In addition, the tool is re-compiled as a dynamic library and linked with QMPI library statically.

4.5 Tests

4.5.1 Overhead Tests

QMPI interface supposed to cause very low to negligible overhead, when executed without any tools in the toolchain, compared to execution of applications without

QMPI. Overhead tests are used in order to prove that the prototype of QMPI satisfies this requirement. The purpose of the overhead tests is to measure the time spent by executions of applications with and without QMPI interface. In order to observe only the effect of QMPI interface on time overhead, we have not placed any tools in the toolstack during tests.

For these tests AMG 2006 and HPCG benchmarks are used as MPI applications. Both benchmarks are executed with and without QMPI interface 50 times and averages of execution times of both cases are calculated, respectively. The MPI time in the plots indicates the time measurements between the end of "MPI_Init" and "MPI_Finalize". By using MPI time plots, overhead that is caused by the functions other than "MPI_Init" and "MPI_Finalize" can be observed. App time in the plots are time measurements taken form the beginning until the end of the main functions in benchmarks. Therefore, by comparing MPI time and App time plots, the time overhead caused by initialization and finalization of MPI can be observed. The average time overhead caused by initialization and finalization of MPI can be found by subtracting averages of App time and MPI time. For AMG executions it is 5.351 seconds without QMPI linked to the appliciation and 5.358 seconds with QMPI linked. For HPCG executions 5.83 seconds with and without QMPI linked to the appliciation. Hence, it is evident that linking QMPI to the application without any tools in the toolchain, does not cause a major difference in terms of time spent on two functions -"MPI_Init" and "MPI_Finalize". The plots obtained from executions using only PMPI are named as "App/MPI time PMPI". The plots obtained from executions using QMPI are named as "App/MPI time QMPI".

Boxplots of test results obtained using AMG2006 and HPCG benchmarks are given in Figures 4.1 & 4.2 and Figures 4.3 & 4.4, respectively.

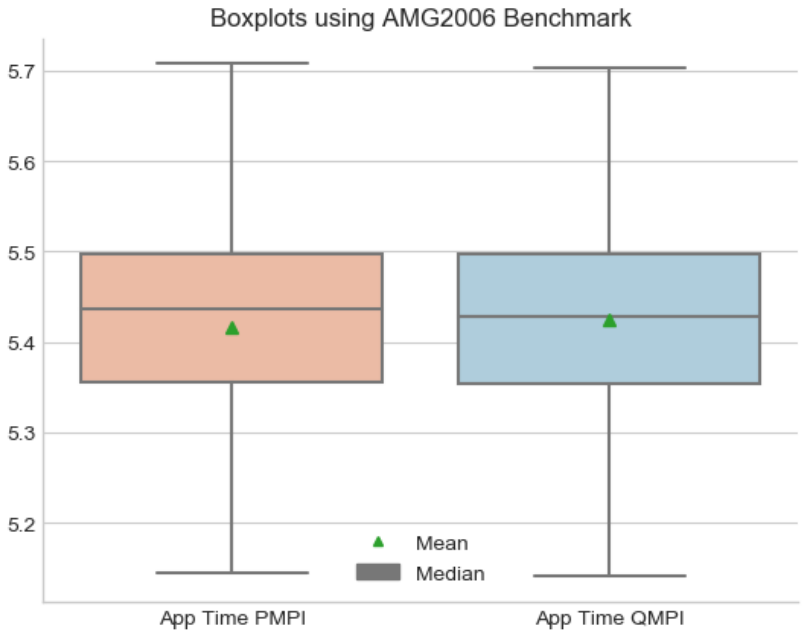


Figure 4.1: Comparison between application time of PMPI and QMPI using AMG Benchmark - PMPI average: 5.41s, QMPI average: 5.42s

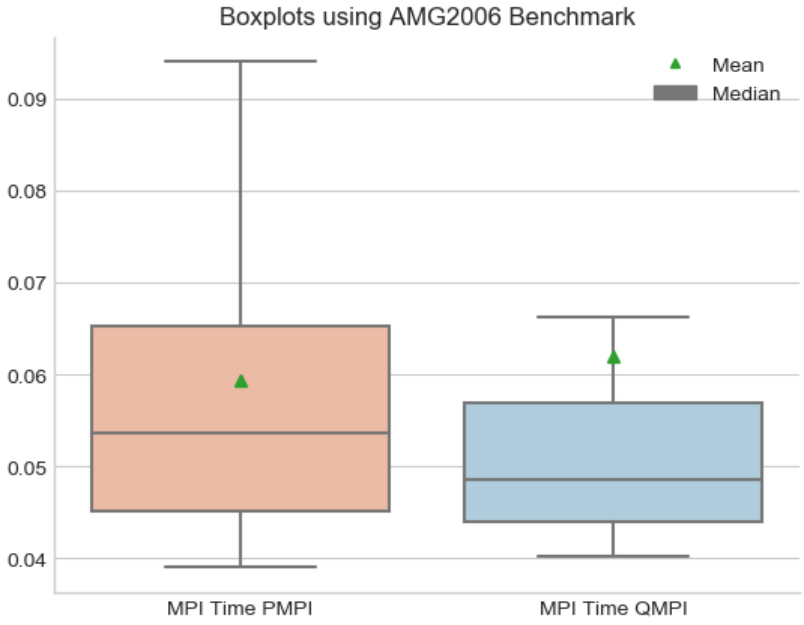


Figure 4.2: Comparison between MPI time of PMPI and QMPI using AMG Benchmark - PMPI average: 0.059s, QMPI average: 0.062s

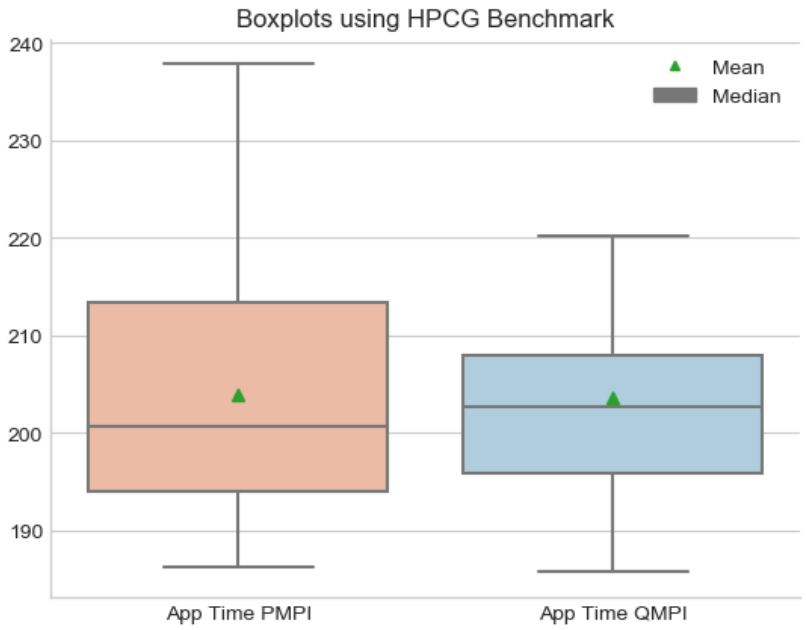


Figure 4.3: Comparison between application time of PMPI and QMPI using HPCG Benchmark - PMPI average: 203.98s, QMPI average: 203.6s

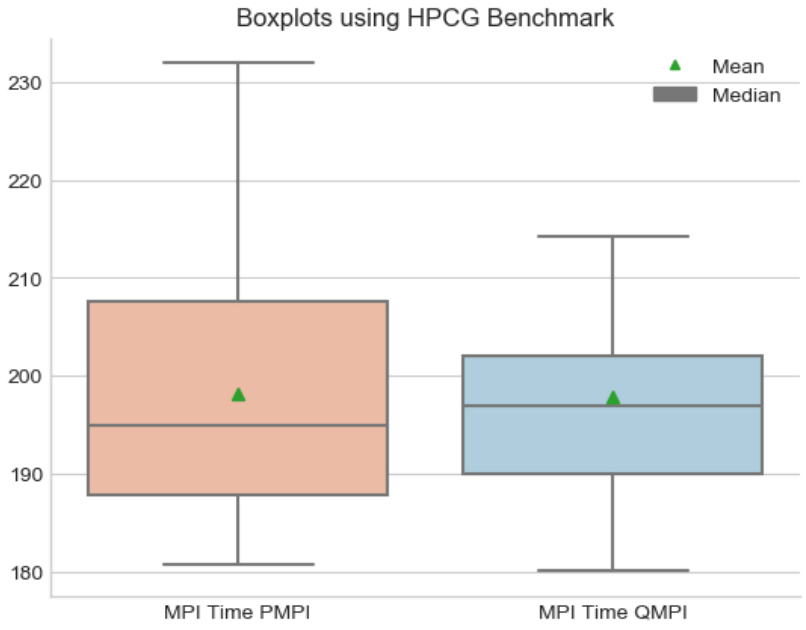


Figure 4.4: Comparison between MPI time of PMPI and QMPI using HPCG Benchmark - PMPI average: 198.15s, QMPI average: 197.77s

Benchmark	Type	P-value	Effect Size
AMG 2006	App Time	0.1249	0.058
AMG 2006	MPI Time	0.0048	0.106
HPCG	App Time	0.3955	0.0321
HPCG	MPI Time	0.4028	0.0316

Table 4.3: Results of Independent T-tests

It can be seen from the boxplots, that QMPI causes very low time overhead to the App time measurements and low overhead to the MPI time measurements.

The results of the conducted T-tests between PMPI and QMPI are also given in Table 4.3. We consider statistical significance by using two factors:

(i) if p-value is less than pre-defined threshold: 0.05,

(ii) if effect size is greater than *large* effect size threshold: 0.8 [9].

Therefore, none of these results indicate a statistically significant difference between using QMPI compared to PMPI in terms of respective time overheads. It should be noted that, the p-value obtained using AMG 2006 benchmark in terms of MPI time overhead is less than 0.05, but it's effect is limited and it falls between *very small* and *small* effect size thresholds [9].

4.5.2 Tool Layer Increment Test

Another requirement of the QMPI interface is to provide very low overhead per each tool added to the toolchain. To prove that the implemented prototype of QMPI causes very low time overhead per tool, tool layer increment tests are used. The purpose of the tool layer increment tests is to increase the number of tools in the stack one by one and to observe the effects of expanding the tool layer, on the time overhead.

Since the purposes of this test is to observe the effect of increasing the size of the toolstack by one tool - rather than the overhead caused by the operations done by the tools- a special tool is implemented for this purpose. The wrappers of this tool have no functionality other than calling the follow-on routines in the function tables.

For this test the size of the toolstack is increased from 1 to 1000 tools, with step size of one. With each different size of the toolstack, from 1 to 1000, the benchmarks are executed 20 times. The average of the time measurements, collected from these 20 runs, is calculated. Further, AMG 2006 benchmark is used as the MPI application to conduct the test. The results are given in Figures 4.5 and 4.6. The terms MPI time and App time in the plots are used to imply the time measurements between "MPI_Init" and "MPI_Finalize" and the beginning and the end of the main function of the benchmark, respectively.

As it can be seen from respective Figures 4.5 and 4.6, the time overhead increases linearly as the toolstack size increases. There are two peak points in the plot -in Figure 4.5- showing irregular high time measurements for toolstack sizes around 250 tools and 695 tools. Apart from these two peaks, the rest of the measurements show linear scaling characteristics and considering the range of toolstack sizes tested(from 1 to 1000 tools) these two peak points are considered as outlier measurements. Hence, they are ignored when assesing the time overhead characteristics of QMPI. In a real-life use-case of the system it is unrealistic to expect more than 100 tools in a toolstack. Therefore, our prototype implementation for QMPI provides good enough time overhead scaling with increasing size of the toolstack.

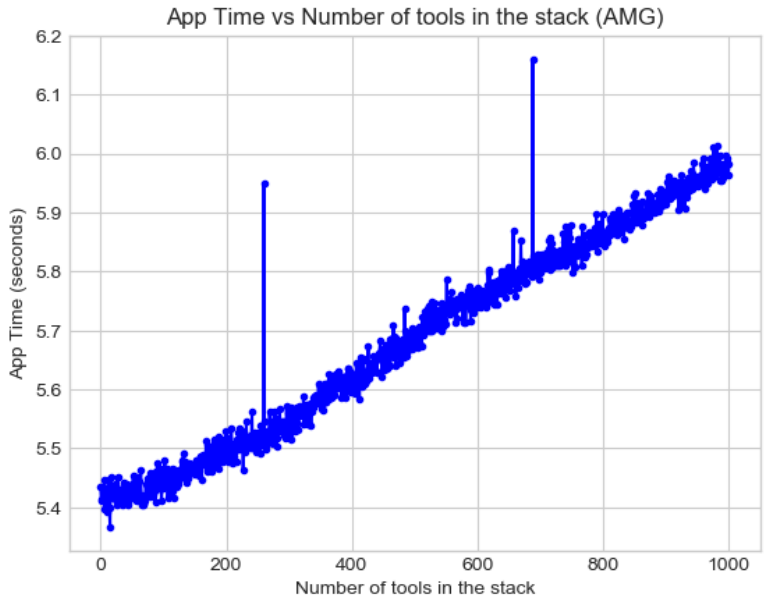


Figure 4.5: App time measurements of AMG benchmark with 1 to 1000 tools

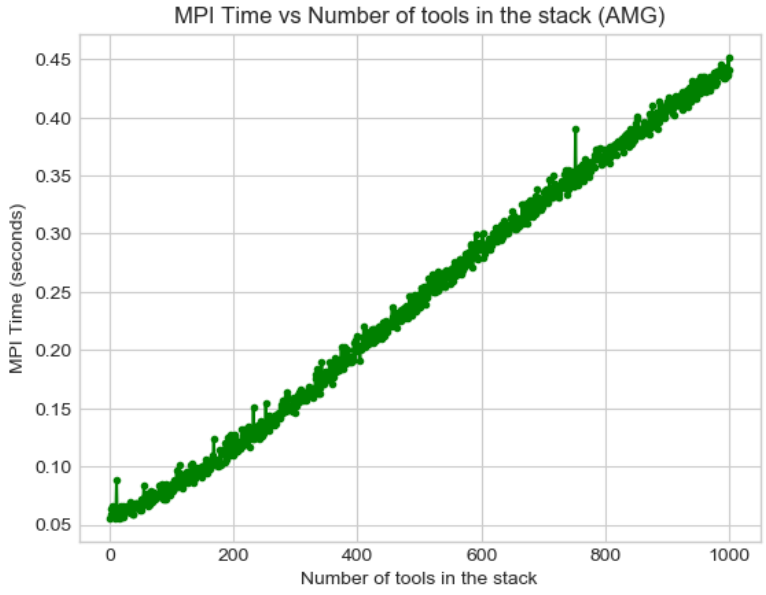


Figure 4.6: MPI time measurements of AMG benchmark with 1 to 1000 tools

4.5.3 Broadcast Re-implementation Tool Test

The purpose of this test is to prove that the context separation between tools is achieved properly. In addition, this test is used to prove the capability of loading tools multiple times.

For this test an MPI application, that broadcasts an integer array with size of 1MB and with randomly assigned integer elements.

For this test, mpiP tool is re-factored and used. In the re-factoring stage apart from making mpiP compatible with the QMPI, storage spaces of some variables that have influence on the computations and results, are separated to achieve context separation behavior. MpiP can report numbers of certain MPI calls and it can measure time spent on those calls in a text file. After re-factoring, with the help of context separation, mpiP is made capable of generating two different reports when it is included and run twice in the toolchain.

A special tool (TBcast tool), which has its own broadcast implementation in the wrapper, that intercepts "MPI_Bcast" is implemented and used. The "MPI_Bcast" wrapper of the tool performs broadcast operation by calling functions via the pointers in the function tables. It calls "MPI_Comm_rank", "MPI_Comm_size", "MPI_Send" and "MPI_Recv" wrappers of the next tool. Therefore, after execution of the layer -at which TBcast tool is placed-, only "MPI_Comm_rank", "MPI_Comm_size", "MPI_Send" and "MPI_Recv" - not "MPI_Bcast" - wrappers of the tools at the layers below are called.

Hence, if TOOLS environment variable includes path of the same mpiP library file twice, in addition to the path of aforementioned TBcast tool in between as illustrated in Figure 4.7, the result should be two different reports coming from different executions of the same mpiP library file- mpiP(1) and mpiP (2) in Figure 4.7. The MPI function counts in these reports should also be different since wrappers intercepting different MPI routines will be called during mpiP execution performed after TBcast tool execution. In fact, the result of the described test setting gives two different mpiP outputs indicating different function calls. The fraction of the resulting reports can be found in Figures 4.8 and 4.9 .

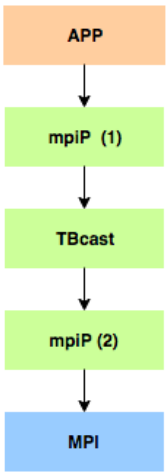


Figure 4.7: Toolchain structure for TOOLS=libmpiP.so:TBcast.so:libmpiP.so

```
Bcast                 1    *     28      9.21      5.81      2.42     24.02     51.72
---------------------------------------------------------------------------------------
@--- Callsite Message Sent statistics (all, sent bytes) ----------------------------
---------------------------------------------------------------------------------------
Name                 Site Rank   Count       Max      Mean       Min       Sum
Bcast                   1    0       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1    1       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1    2       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1    3       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1    4       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1    5       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1    6       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1    7       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1    8       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1    9       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   10       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   11       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   12       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   13       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   14       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   15       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   16       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   17       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   18       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   19       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   20       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   21       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   22       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   23       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   24       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   25       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   26       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1   27       1 1.049e+06 1.049e+06 1.049e+06 1.049e+06
Bcast                   1    *      28 1.049e+06 1.049e+06 1.049e+06 2.936e+07
---------------------------------------------------------------------------------------
@--- End of Report -----------------------------------------------------------------
---------------------------------------------------------------------------------------
```

Figure 4.8: Fraction of the mpiP Report, that is constructed by first call of mpiP tool library before the call of TBcast tool.

```
Recv              38   27      1      9.18      9.18      9.18    30.96    57.01
Recv              38    *      1      9.18      9.18      9.18     1.11     2.96

Recv              40    9      1      5.05      5.05      5.05    17.06    38.53
Recv              40    *      1      5.05      5.05      5.05     0.61     1.63

Recv              42   23      1      7.79      7.79      7.79    26.36    68.25
Recv              42    *      1      7.79      7.79      7.79     0.94     2.51

Recv              44   22      1      7.63      7.63      7.63    25.81    45.46
Recv              44    *      1      7.63      7.63      7.63     0.92     2.46

Recv              46    6      1      3.66      3.66      3.66    12.31    33.09
Recv              46    *      1      3.66      3.66      3.66     0.44     1.18

Recv              48   26      1      8.66      8.66      8.66    29.15    72.99
Recv              48    *      1      8.66      8.66      8.66     1.04     2.79

Recv              50   13      1      5.44      5.44      5.44    18.39    55.00
Recv              50    *      1      5.44      5.44      5.44     0.66     1.75

Recv              52   14      1      5.51      5.51      5.51    18.57    67.60
Recv              52    *      1      5.51      5.51      5.51     0.66     1.77

Recv              54   16      1      5.67      5.67      5.67    19.14    46.65
Recv              54    *      1      5.67      5.67      5.67     0.68     1.83

Recv              56    8      1      3.84      3.84      3.84    12.93    68.60
Recv              56    *      1      3.84      3.84      3.84     0.46     1.24

Send               2    0     27      1.88     0.313     0.183    28.11    52.37
Send               2    *     27      1.88     0.313     0.183     1.02     2.72
---------------------------------------------------------------------------
@--- Callsite Message Sent statistics (all, sent bytes) --------------------
---------------------------------------------------------------------------
Name              Site Rank   Count       Max      Mean       Min       Sum
Send                 2    0      27 1.049e+06 1.049e+06 1.049e+06 2.831e+07
Send                 2    *      27 1.049e+06 1.049e+06 1.049e+06 2.831e+07
---------------------------------------------------------------------------
@--- End of Report ---------------------------------------------------------
---------------------------------------------------------------------------
```

Figure 4.9: Fraction of the mpiP Report, that is constructed by second call of mpiP tool library after the call of TBcast tool (Third column with asterisks indicate the rank number of the caller processes).

5 Related Work

5.1 The Message Passing Interface (MPI)

Message passing is a paradigm that is widely used, especially in distributed memory systems. Until the release of the first MPI standardization document in 1994 each vendor had implemented their own variant of a message passing interface. MPI is an industry standard that defines both the syntax and the semantics of the core library routines of a message passing interface. It is efficiently implementable on a wide range of computers. In fact it is influenced by many different existing message passing systems implemented for wide range of computer architectures and made use of their most attractive features. [7]

The major goals of MPI are as follows [7]:

- Design an application programming interface.
- Allow efficient communication.
- Enable implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran bindings for the interface.
- Provide the user failure free with communication. Failures in communication are dealt with by the underlying communication subsystem.
- Define an interface that can be implemented on different platforms, with no significant changes in the underlying communication and system software.

- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread safety.

Since its first release in 1994, there have been many versions of the MPI standard. However, the latest version of the MPI standard released in June, 2015 includes standardization of the following features [3]:

- Point-to-point communication
- Datatypes
- Collective Communication
- Groups,Contexts, Communicators and Caching
- Process Topologies
- MPI Environment Management
- Process Creation and Management
- One-Sided Communications
- Input / Output
- External interfaces
- Language Bindings
- Tool Support

5.2 Profiling Interface for MPI (PMPI)

PMPI is the profiling interface of MPI and it provides the necessary information about the operation of MPI processes to the tools, e.g. debuggers, performance analyzers etc.

An implementation of PMPI must provide access to all MPI functions, except those allowed as macros, with a name shift. This requires an alternate entry point name, with the prefix PMPI_ for each MPI function[3]. It provides all the functionality provided by MPI through its functions with shifted names.

All MPI functions are declared as weak symbols. Therefore, they can be easily overwritten by any function, that is linked to the application and uses the same name with an MPI function. On the other hand, PMPI symbols are implemented as regular symbols, so that the tool libraries can declare and define wrapper routines with the same names as MPI functions and within these wrappers they can call name shifted version of the MPI functions in PMPI [10]. This enables tool developers to build a layer between the application and PMPI, e.g. wrappers, and extract necessary information related to the execution of the application.

Even if PMPI is considered as the essence of tool implementations for MPI applications, it has some major shortcomings. First of these shortcomings is that, there can be only one tool linked between the application and the MPI library at a time. Therefore, using multiple tools concurrently is not possible using only PMPI. Another shortcoming is, each time a different tool to be used, it must be re-linked with the MPI application. This means the application must be re-compiled each time a different tool is desired to be used. For small applications this may not be a major drawback but in cases when compilation takes too much time re-compiling several times can be quite cumbersome. Moreover, since PMPI does not support concurrent execution of tools, it is not possible to combine different tools as modules of a bigger tool, to work in collaboration. This discourages code reuse and provides harder testability of the tools[10].

5.3 PNMPI

PNMPI is an interface that enables running MPI tools concurrently in a single execution of the MPI application. It creates toolstacks and executes the tools in this stack concurrently as the MPI application in executed. PNMPI is designed to be linked with the MPI applications by default. It causes very low time overhead and does not effect the functionality of the MPI application when it is used without any tools.

PNMPI has three main components: a stub library, the PNMPI core and a configuration loader module. Stub library integrates the whole toolchain into the MPI application. PNMPI core manages and controls the toolchain. Configuration loader module defines the modules to be used in the structure. A configuration file lists the modules to be used in the structure and these modules are loaded during initialization phase of PNMPI. "MPI_Init" wrapper, which belongs to the PNMPI core, builds the chain structure with the modules listed in the configuration file. By using a publisher/subscriber interface, tools register their services and can search for services offered by other tools. This provides tools to be layered on top of functionalities provided by other tools [10].

PNMPI is based on similar concepts and it provides features that are also offered by QMPI. In fact, the ideas and concepts of PNMPI are used as inspiration for QMPI. The most significant difference in approaches of two interfaces is that, QMPI does not interfere with function calls once the toolchain is triggered. Tools request the function pointers and execute them themselves in a nested manner. On the contrary, PNMPI takes over the program control each time a function wrapper returns from execution. Thus, it causes extra jumps back and forth between the tools and the PNMPI. Therefore, theoretically QMPI would provide better time overhead performance in comparison to PNMPI. The execution mechanism of QMPI is shown in Figure 2.1 and simplified execution mechanism for PNMPI is shown in Figure 5.1.

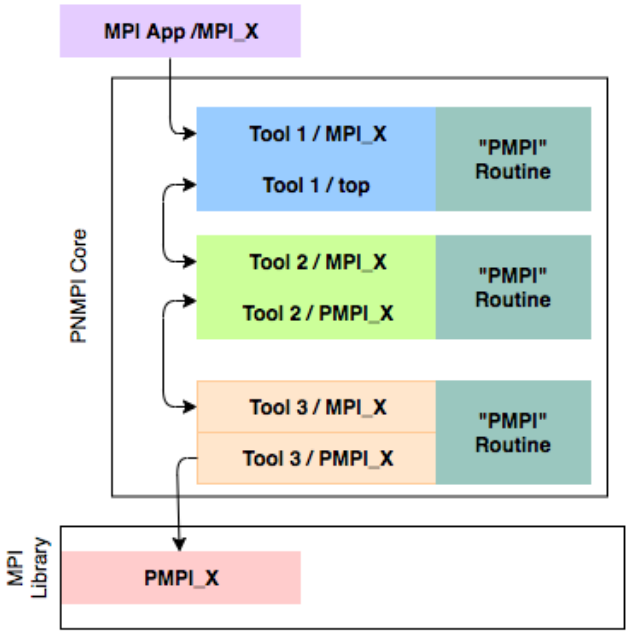


Figure 5.1: Execution mechanism of PNMPI [11]

5.4 IBM Spectrum MPI - Dynamic MPI profiling Interface with Layering

IBM Spectrum MPI 10.1 is a commercial MPI implementation developed by IBM and it complies with all the requirements of the MPI standard v3.1 released in June, 2015 [6]. It has a profiling interface called *dynamic MPI profiling interface with layering*. This interface allows multiple dynamic profiling tool libraries to be executed concurrently in layers placed between MPI applications and the MPI library.

If a dynamic library, that is put in a layer, has a wrapper for a certain MPI function; then this wrapper is executed before invoking the actual MPI function from MPI library (libmpi.so). If the dynamic tool library does not have a wrapper for the certain MPI function, then the execution jumps directly to libmpi.so. When several layers filled with dynamic libraries are used, the execution flows between the layers of these dynamic libraries using a depth counter. As the MPI application is called, depth counter is initialized to zero and incremented when execution of each layer is completed until function call from libmpi.so is made. The order of the dynamic tool libraries used in the toolstack, is provided with the execution command by `-entry`

option after `mpirun` command. [6].

Since IBM's dynamic MPI profiling interface is a part of their IBM MPI Spectrum library, this profiling interface cannot be used for other MPI implementations. Therefore, the usability of this interface is only restricted to IBM's MPI implementation.

6 Conclusion

Throughout this work we described the design, architecture, and innovative features of QMPI. In this chapter we provide an overview of contributions and suggest interesting topics as a future work.

6.1 Overview of contributions

QMPI allows simple profiling tools to achieve a more complex profiling tasks by working . It provides easier tool implementations and their testing & maintenance for the developers.

In previous chapters we presented QMPI, which enables different tools to work concurrently and in a collaborative manner within a toolchain structure. Three different design approaches- namely Original MPI Forum Design, Intermediate Design and Final Design- for QMPI are presented in details. Each design is made by improving the previous one in order to meet the requirements of QMPI. Then, the implementation of the Final Design, its features and special functions are explained. Details of the implemented tools to test previously-explained features and functionalities of QMPI are also provided.

In section 3.3 how to implement a QMPI compatible tool is explained. The developer must replace the PMPI function calls in its wrapper functions with three lines of code. Thus, it is really easy to re-factor a PMPI compatible tool into a QMPI compatible one. Furthermore, the tool developer can use context separation to use the same tool multiple times in a toolchain if desired.

Performance of QMPI is assessed using the tests explained in chapter 4 (Eval-

uation). It is shown that QMPI causes negligible time overhead for the following scenarios: (i) QMPI is linked to the MPI application, (ii) No tools are used. Moreover, it is shown that as the number of tools in the toolchain increases, the time overhead caused by adding each tool is low and increases linearly instead of exponentially. Furthermore, the context separation is also achieved properly with the help of the test explained in Section 4.5.3.

In conclusion, the QMPI design and implementation presented in this work, fulfills all of its requirements. It causes negligible time overhead and doesn't change the functionality of the application when the toolchain is empty. Therefore, it can be linked with the MPI applications by default. By the help of this default linking, our QMPI implementation provides a very easy activation and deactivation of toolchain - through TOOLS environment variable. It solves "one tool at a time" restriction of PMPI and allows running multiple tools concurrently. It is a very useful extension to PMPI to avoid re-linking application whenever a new tool is to be used. At last but not at least, it allows simpler tool implementations with easier testability and promotes code reuse.

6.2 Future work and Extensions

In the current version of QMPI, function tables are stored and all of these tables are reachable through "QMPI_Table_query" function regardless of the tool they are queried from. The fact that QMPI stores the function tables as a whole structure, requires a function call- "QMPI_Table_query" - each time a follow on routine is to be called by a tool. Therefore, in the future QMPI should be able to distribute function tables to the corresponding tools and tools should be able to store their part of the function tables. This can help avoiding the additional jump between "QMPI_Table_query" and wrappers hence, it provides more efficient interface.

In addition, even if it is a design choice made for better usability, "exec_func" causes a considerable amount of time overhead. Therefore, we suggest implementing "exec_func" as a macro. However, a better approach to this problem might be providing another type of implementation for "QMPI_Table_query" that hooks the function pointer to the follow-on routine and returns the level information in the table cell, which contains the pointer to the follow-on routine of a wrapper. By this way, the

tool developer can cast the obtained function pointer to the corresponding type and call the function pointed by the pointer with the obtained level parameter. However, this approach requires some amount of knowledge related to the internal functioning of QMPI and provides less usability then the approach in the current version of QMPI implementation. Nevertheless, in certain cases, at which time performance of the tool is more important than usability, this suggested approach can be used. Therefore, it is better to provide such an option and the necessary function implementations and leave it to the tool developer to choose between these approaches according to hisher needs. Also, implementing "QMPI_Table_query" and "QMPI_Set_context" as a macro can improve the performance of the QMPI interface.

Moreover, in the future a ping-pong benchmark can be used to observe the latency of the system with QMPI interface linked to the benchmark to better assess the performance of QMPI. Even if QMPI takes over some major ideas and concepts from PNMPI, they have different implementations and approaches to the problem. Therefore, conducting tests, that have already been conducted using QMPI, by using PNMPI - in order to prove that QMPI performs better in terms of time over head - is another task that can be accomplished in the future.

7 Appendix

7.1 MPI indices

_MPI_Abort = 0,	_MPI_Accumulate = 1,	_MPI_Add_error_class = 2,
_MPI_Add_error_code = 3,	_MPI_Add_error_string = 4,	_MPI_Address = 5,
_MPI_Allgather = 6,	_MPI_Allgatherv = 7,	_MPI_Alloc_mem = 8,
_MPI_Allreduce = 9,	_MPI_Alltoall = 10,	_MPI_Alltoallv = 11,
_MPI_Alltoallw = 12,	_MPI_Attr_delete = 13,	_MPI_Attr_get = 14,
_MPI_Attr_put = 15,	_MPI_Barrier = 16,	_MPI_Bcast = 17,
_MPI_Bsend = 18,	_MPI_Bsend_init = 19,	_MPI_Buffer_attach = 20,
_MPI_Buffer_detach = 21,	_MPI_Cancel = 22,	_MPI_Cart_coords = 23,
_MPI_Cart_create = 24,	_MPI_Cart_get = 25,	_MPI_Cart_map = 26,
_MPI_Cart_rank = 27,	_MPI_Cart_shift = 28,	_MPI_Cart_sub = 29,
_MPI_Cartdim_get = 30,	_MPI_Close_port = 31,	_MPI_Comm_accept = 32,
_MPI_Comm_call_errhandler = 33,	_MPI_Comm_compare = 34,	_MPI_Comm_connect = 35,
_MPI_Comm_create = 36,	_MPI_Comm_create_errhandler = 37,	_MPI_Comm_create_group = 38,
_MPI_Comm_create_keyval = 39,	_MPI_Comm_delete_attr = 40,	_MPI_Comm_disconnect = 41,
_MPI_Comm_dup = 42,	_MPI_Comm_dup_with_info = 43,	_MPI_Comm_free = 44,
_MPI_Comm_free_keyval = 45,	_MPI_Comm_get_attr = 46,	_MPI_Comm_get_errhandler = 47,
_MPI_Comm_get_info = 48,	_MPI_Comm_get_name = 49,	_MPI_Comm_get_parent = 50,
_MPI_Comm_group = 51,	_MPI_Comm_idup = 52,	_MPI_Comm_join = 53,
_MPI_Comm_rank = 54,	_MPI_Comm_remote_group = 55,	_MPI_Comm_remote_size = 56,
_MPI_Comm_set_attr = 57,	_MPI_Comm_set_errhandler = 58,	_MPI_Comm_set_info = 59,
_MPI_Comm_set_name = 60,	_MPI_Comm_size = 61,	_MPI_Comm_split = 62,
_MPI_Comm_split_type = 63,	_MPI_Comm_test_inter = 64,	_MPI_Compare_and_swap = 65,
_MPI_Dims_create = 66,	_MPI_Dist_graph_create = 67,	_MPI_Dist_graph _create_adjacent = 68,
_MPI_Dist_graph_neighbors = 69,	_MPI_Dist_graph _neighbors_count = 70,	_MPI_Errhandler_create = 71,
_MPI_Errhandler_free = 72,	_MPI_Errhandler_get = 73,	_MPI_Errhandler_set = 74,

Table 7.1: MPI Indices - Part 1

_MPI_Error_class = 75,	_MPI_Error_string = 76,	_MPI_Exscan = 77,
_MPI_Fetch_and_op = 78,	_MPI_File_call_errhandler = 79,	_MPI_File_close = 80,
_MPI_File_create_errhandler = 81,	_MPI_File_delete = 82,	_MPI_File_get_amode = 83,
_MPI_File_get_atomicity = 84,	_MPI_File_get_byte_offset = 85,	_MPI_File_get_errhandler = 86,
_MPI_File_get_group = 87,	_MPI_File_get_info = 88,	_MPI_File_get_position = 89,
_MPI_File_get_position_shared = 90,	_MPI_File_get_size = 91,	_MPI_File_get_type_extent = 92,
_MPI_File_get_view = 93,	_MPI_File_iread = 94,	_MPI_File_iread_all = 95,
_MPI_File_iread_at = 96,	_MPI_File_iread_at_all = 97,	_MPI_File_iread_shared = 98,
_MPI_File_iwrite = 99,	_MPI_File_iwrite_all = 100,	_MPI_File_iwrite_at = 101,
_MPI_File_iwrite_at_all = 102,	_MPI_File_iwrite_shared = 103,	_MPI_File_open = 104,
_MPI_File_preallocate = 105,	_MPI_File_read = 106,	_MPI_File_read_all = 107,
_MPI_File_read_all_begin = 108,	_MPI_File_read_all_end = 109,	_MPI_File_read_at = 110,
_MPI_File_read_at_all = 111,	_MPI_File_read_at_all_begin = 112,	_MPI_File_read_at_all_end = 113,
_MPI_File_read_ordered = 114,	_MPI_File_read _ordered_begin = 115,	_MPI_File_read_ordered_end = 116,
_MPI_File_read_shared = 117,	_MPI_File_seek = 118,	_MPI_File_seek_shared = 119,
_MPI_File_set_atomicity = 120,	_MPI_File_set_errhandler = 121,	_MPI_File_set_info = 122,
_MPI_File_set_size = 123,	_MPI_File_set_view = 124,	_MPI_File_sync = 125,
_MPI_File_write = 126,	_MPI_File_write_all = 127,	_MPI_File_write_all_begin = 128,
_MPI_File_write_all_end = 129,	_MPI_File_write_at = 130,	_MPI_File_write_at_all = 131,
_MPI_File_write_at_all_begin = 132,	_MPI_File_write_at_all_end = 133,	_MPI_File_write_ordered = 134,
_MPI_File_write _ordered_begin = 135,	_MPI_File_write _ordered_end = 136,	_MPI_File_write_shared = 137,
_MPI_Finalize = 138,	_MPI_Finalized = 139,	_MPI_Free_mem = 140,

Table 7.2: MPI Indices - Part 2

_MPI_Gather = 141,	_MPI_Gatherv = 142,	_MPI_Get = 143,
_MPI_Get_accumulate = 144,	_MPI_Get_address = 145,	_MPI_Get_count = 146,
_MPI_Get_elements = 147,	_MPI_Get_elements_x = 148,	_MPI_Get_library_version = 149,
_MPI_Get_processor_name = 150,	_MPI_Get_version = 151,	_MPI_Graph_create = 152,
_MPI_Graph_get = 153,	_MPI_Graph_map = 154,	_MPI_Graph_neighbors = 155,
_MPI_Graph_neighbors_count = 156,	_MPI_Graphdims_get = 157,	_MPI_Grequest_complete = 158,
_MPI_Grequest_start = 159,	_MPI_Group_compare = 160,	_MPI_Group_difference = 161,
_MPI_Group_excl = 162,	_MPI_Group_free = 163,	_MPI_Group_incl = 164,
_MPI_Group_intersection = 165,	_MPI_Group_range_excl = 166,	_MPI_Group_range_incl = 167,
_MPI_Group_rank = 168,	_MPI_Group_size = 169,	_MPI_Group_translate_ranks = 170,
_MPI_Group_union = 171,	_MPI_Iallgather = 172,	_MPI_Iallgatherv = 173,
_MPI_Iallreduce = 174,	_MPI_Ialltoall = 175,	_MPI_Ialltoallv = 176,
_MPI_Ialltoallw = 177,	_MPI_Ibarrier = 178,	_MPI_Ibcast = 179,
_MPI_Ibsend = 180,	_MPI_Iexscan = 181,	_MPI_Igather = 182,
_MPI_Igatherv = 183,	_MPI_Improbe = 184,	_MPI_Imrecv = 185,
_MPI_Ineighbor_allgather = 186,	_MPI_Ineighbor_allgatherv = 187,	_MPI_Ineighbor_alltoall = 188,
_MPI_Ineighbor_alltoallv = 189,	_MPI_Ineighbor_alltoallw = 190,	_MPI_Info_create = 191,
_MPI_Info_delete = 192,	_MPI_Info_dup = 193,	_MPI_Info_free = 194,
_MPI_Info_get = 195,	_MPI_Info_get_nkeys = 196,	_MPI_Info_get_nthkey = 197,
_MPI_Info_get_valuelen = 198,	_MPI_Info_set = 199,	_MPI_Init = 200,
_MPI_Init_thread = 201,	_MPI_Initialized = 202,	_MPI_Intercomm_create = 203,
_MPI_Intercomm_merge = 204,	_MPI_Iprobe = 205,	_MPI_Irecv = 206,
_MPI_Ireduce = 207,	_MPI_Ireduce_scatter = 208,	_MPI_Ireduce_scatter_block = 209,

Table 7.3: MPI Indices - Part 3

_MPI_Irsend = 210,	_MPI_Is_thread_main = 211,	_MPI_Iscan = 212,
_MPI_Iscatter = 213,	_MPI_Iscatterv = 214,	_MPI_Isend = 215,
_MPI_Issend = 216,	_MPI_Keyval_create = 217,	_MPI_Keyval_free = 218,
_MPI_Lookup_name = 219,	_MPI_Mprobe = 220,	_MPI_Mrecv = 221,
_MPI_Neighbor_allgather = 222,	_MPI_Neighbor_allgatherv = 223,	_MPI_Neighbor_alltoall = 224,
_MPI_Neighbor_alltoallv = 225,	_MPI_Neighbor_alltoallw = 226,	_MPI_Op_commutative = 227,
_MPI_Op_create = 228,	_MPI_Op_free = 229,	_MPI_Open_port = 230,
_MPI_Pack = 231,	_MPI_Pack_external = 232,	_MPI_Pack_external_size = 233,
_MPI_Pack_size = 234,	_MPI_Pcontrol = 235,	_MPI_Probe = 236,
_MPI_Publish_name = 237,	_MPI_Put = 238,	_MPI_Query_thread = 239,
_MPI_Raccumulate = 240,	_MPI_Recv = 241,	_MPI_Recv_init = 242,
_MPI_Reduce = 243,	_MPI_Reduce_local = 244,	_MPI_Reduce_scatter = 245,
_MPI_Reduce_scatter_block = 246,	_MPI_Register_datarep = 247,	_MPI_Request_free = 248,
_MPI_Request_get_status = 249,	_MPI_Rget = 250,	_MPI_Rget_accumulate = 251,
_MPI_Rput = 252,	_MPI_Rsend = 253,	_MPI_Rsend_init = 254,
_MPI_Scan = 255,	_MPI_Scatter = 256,	_MPI_Scatterv = 257,
_MPI_Send = 258,	_MPI_Send_init = 259,	_MPI_Sendrecv = 260,
_MPI_Sendrecv_replace = 261,	_MPI_Ssend = 262,	_MPI_Ssend_init = 263,
_MPI_Start = 264,	_MPI_Startall = 265,	_MPI_Status_set_cancelled = 266,
_MPI_Status_set_elements = 267,	_MPI_Status_set_elements_x = 268,	_MPI_Test = 269,
_MPI_Test_cancelled = 270,	_MPI_Testall = 271,	_MPI_Testany = 272,
_MPI_Testsome = 273,	_MPI_Topo_test = 274,	_MPI_Type_commit = 275,
_MPI_Type_contiguous = 276,	_MPI_Type_create_darray = 277,	_MPI_Type_ create_f90_complex = 278,
_MPI_Type_ create_f90_integer = 279,	_MPI_Type _create_f90_real = 280,	_MPI_Type_create_hindexed = 281,
_MPI_Type_create_ hindexed_block = 282,	_MPI_Type_ create_hvector = 283,	_MPI_Type_ create_indexed_block = 284,
_MPI_Type_create_keyval = 285,	_MPI_Type_create_resized = 286,	_MPI_Type_create_struct = 287,

centering

Table 7.4: MPI Indices - Part 4

_MPI_Type_create_subarray = 288,	_MPI_Type_delete_attr = 289,	_MPI_Type_dup = 290,
_MPI_Type_extent = 291,	_MPI_Type_free = 292,	_MPI_Type_free_keyval = 293,
_MPI_Type_get_attr = 294,	_MPI_Type_get_contents = 295,	_MPI_Type_get_envelope = 296,
_MPI_Type_get_extent = 297,	_MPI_Type_get_extent_x = 298,	_MPI_Type_get_name = 299,
_MPI_Type_get_true_extent = 300,	_MPI_Type_get_true_extent_x = 301,	_MPI_Type_hindexed = 302,
_MPI_Type_hvector = 303,	_MPI_Type_indexed = 304,	_MPI_Type_lb = 305,
_MPI_Type_match_size = 306,	_MPI_Type_set_attr = 307,	_MPI_Type_set_name = 308,
_MPI_Type_size = 309,	_MPI_Type_size_x = 310,	_MPI_Type_struct = 311,
_MPI_Type_ub = 312,	_MPI_Type_vector = 313,	_MPI_Unpack = 314,
_MPI_Unpack_external = 315,	_MPI_Unpublish_name = 316,	_MPI_Wait = 317,
_MPI_Waitall = 318,	_MPI_Waitany = 319,	_MPI_Waitsome = 320,
_MPI_Win_allocate = 321,	_MPI_Win_allocate_shared = 322,	_MPI_Win_attach = 323,
_MPI_Win_call_errhandler = 324,	_MPI_Win_complete = 325,	_MPI_Win_create = 326,
_MPI_Win_create_dynamic = 327,	_MPI_Win_create_errhandler = 328,	_MPI_Win_create_keyval = 329,
_MPI_Win_delete_attr = 330,	_MPI_Win_detach = 331,	_MPI_Win_fence = 332,
_MPI_Win_flush = 333,	_MPI_Win_flush_all = 334,	_MPI_Win_flush_local = 335,
_MPI_Win_flush_local_all = 336,	_MPI_Win_free = 337,	_MPI_Win_free_keyval = 338,
_MPI_Win_get_attr = 339,	_MPI_Win_get_errhandler = 340,	_MPI_Win_get_group = 341,
_MPI_Win_get_info = 342,	_MPI_Win_get_name = 343,	_MPI_Win_lock = 344,
_MPI_Win_lock_all = 345,	_MPI_Win_post = 346,	_MPI_Win_set_attr = 347,
_MPI_Win_set_errhandler = 348,	_MPI_Win_set_info = 349,	_MPI_Win_set_name = 350,
_MPI_Win_shared_query = 351,	_MPI_Win_start = 352,	_MPI_Win_sync = 353,
_MPI_Win_test = 354,	_MPI_Win_unlock = 355,	_MPI_Win_unlock_all = 356,
_MPI_Win_wait = 357,	_MPI_Wtick = 358,	_MPI_Wtime = 359

Table 7.5: MPI Indices - Part 5

7.2 Simple Tool Implementations

Listing 7.1: Tool 1 (T1)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../src/obj/qmpi.h"

#ifndef _EXTERN_C_
#ifdef __cplusplus
#define _EXTERN_C_ extern "C"
#else /* __cplusplus */
#define _EXTERN_C_
#endif /* __cplusplus */
#endif /* _EXTERN_C_ */

   char* interceptions[] = {
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "T1_Allgather", "T1_Allgatherv", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
   "T1_Comm_rank", "NULL", "NULL",
```

```
"NULL", "NULL", "NULL",
"NULL", "T1_Comm_size", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"T1_Finalize", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
```

```
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "T1_Init",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "T1_Pcontrol", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
```

```
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL"
    };
_EXTERN_C_ char* get_interceptions(int i) {
    return interceptions[i];
}
_EXTERN_C_ int T1_Init(int *argc,char ***argv,int i,vector * v){
    int ret = 1;
    void* f=NULL;
    QMPI_TABLE_QUERY(200,&f,(*VECTOR_GET(v,i)).table);
    ret = EXEC_FUNC(f,i,200,v,argc,argv);
    return ret;
}
```

```
_EXTERN_C_ int T1_Comm_rank(MPI_Comm comm, int *rank,int i,vector
    * v ){
    void* f = NULL;
    QMPI_TABLE_QUERY(_MPI_Comm_rank,&f,(*VECTOR_GET(v,i)).table);
    int new_level=QMPI_Get_level(i,_MPI_Comm_rank, v);
    typedef int (*comm_rank_func)(MPI_Comm comm,int *rank,int i ,
        vector* v );
    int ret = ( (comm_rank_func) f ) ( comm, rank, new_level, v);
    return ret;
}
_EXTERN_C_ void T1_Finalize(int i, vector* v ){
    void* f = NULL;
    QMPI_TABLE_QUERY( _MPI_Finalize, &f, (*VECTOR_GET(v,i)).table);
    EXEC_FUNC(f,i,_MPI_Finalize,v);
    return;
}

_EXTERN_C_ void T1_Comm_size(MPI_Comm comm, int *size,int i,vector
     * v ){
    void* f = NULL;
    QMPI_TABLE_QUERY(_MPI_Comm_size,&f,(*VECTOR_GET(v,i)).table);
    EXEC_FUNC(f,i,_MPI_Comm_size,v,comm,size);
    return;
}

_EXTERN_C_ void T1_Allgather(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm comm, int i ,vector* v){
    void* f = NULL;
    QMPI_TABLE_QUERY(_MPI_Allgather,&f,(*VECTOR_GET(v,i)).table);
    EXEC_FUNC(f,i,_MPI_Allgather,v,sendbuf, sendcount, sendtype,
        recvbuf, recvcount, recvtype, comm );
    return;
}

_EXTERN_C_ void T1_Allgatherv(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, const int recvcounts[],
    const int displs[], MPI_Datatype recvtype, MPI_Comm comm, int i
```

```
    ,vector* v){
    void* f = NULL;
    QMPI_TABLE_QUERY(_MPI_Allgatherv,&f,(*VECTOR_GET(v,i)).table);
    EXEC_FUNC(f,i,_MPI_Allgatherv,v,sendbuf, sendcount, sendtype,
       recvbuf, recvcounts, displs, recvtype, comm);
    return;
}

_EXTERN_C_ void T1_Pcontrol(const int level, int i ,vector* v){
    void* f = NULL;
    return;
}
```

7.3 Tool that reimplements MPI_Bcast (TBcast)

Listing 7.2: Tool 1 (T1)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../src/obj/qmpi.h"

#ifndef _EXTERN_C_
#ifdef __cplusplus
#define _EXTERN_C_ extern "C"
#else /* __cplusplus */
#define _EXTERN_C_
#endif /* __cplusplus */
#endif /* _EXTERN_C_ */

// char* interceptions[]={"NULL","T2_Comm_rank","T2_Finalize","
    T2_Comm_size"};
char* interceptions[]={
   "NULL", "NULL", "NULL",
   "NULL", "NULL", "NULL",
```

```
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "TB_Bcast",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
```

```
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
```

```
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
"NULL", "NULL", "NULL",
```

```
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL",
    "NULL", "NULL", "NULL"
};

_EXTERN_C_ char* get_interceptions(int i ) {
    return interceptions[i];
}

_EXTERN_C_ int TB_Bcast( void *buffer,int count,MPI_Datatype
    datatype,int root,MPI_Comm comm,int i, vector* v) {
    int ret = 1;
    void* f = NULL;
    int rank = 0;
    MPI_TABLE_QUERY(_MPI_Comm_rank,&f,(*(VECTOR_GET(v,i))).table);
    EXEC_FUNC(f, i, _MPI_Comm_rank,v,MPI_COMM_WORLD, &rank);

    int comm_size=0;
    MPI_TABLE_QUERY(_MPI_Comm_size,&f,(*(VECTOR_GET(v,i))).table);
    EXEC_FUNC(f, i, _MPI_Comm_size, v, comm, &comm_size);

    if( rank == root ) {
        for ( int index =0;index < comm_size; index++) {
            if(index != root) {
                MPI_TABLE_QUERY(_MPI_Send, &f,(*(VECTOR_GET(v,i))).
                    table);
                ret = EXEC_FUNC(f, i, _MPI_Send, v, buffer, count,
                    datatype, index, 0, comm);
            }
        }
    } else {
        MPI_TABLE_QUERY(_MPI_Recv, &f, (*(VECTOR_GET(v,i))).table);
        ret = EXEC_FUNC(f , i, _MPI_Recv, v, buffer, count,
            datatype, root, 0, comm, MPI_STATUS_IGNORE);
    }
    return ret;
}
```

7.4 Data structure for mpiP for context separation

Listing 7.3: Tool 1 (T1)

```
typedef struct _mpiPi_t {
  int ac;
  char *av[32];
  char *toolname;
  char *appName;
  char *appFullName;
  char oFilename[256];
  int tag;
  int procID;
  int rank;
  int size;
  int collectorRank;
#ifndef ENABLE_API_ONLY
  MPI_Comm comm;
  mpiPi_hostname_t hostname;
#endif
  int hostnamelen;
  char *outputDir;
  char *envStr;
  FILE *stdout_;
  FILE *stderr_;
  mpiPi_TIME startTime;
  mpiPi_TIME endTime;

  /* necessary for pcontrol */
  double cumulativeTime;
  time_t start_timeofday;
  time_t stop_timeofday;

  /* pcontrol */
```

```
int enabled;
int enabledCount;

mpiPi_TIMER timer;
mpiPi_hostname_t *global_task_hostnames;
double *global_task_app_time;
double *global_task_mpi_time;
double global_app_time;
double global_mpi_time;
double global_mpi_size;
double global_mpi_io;
double global_mpi_rma;
long long global_mpi_msize_threshold_count;
long long global_mpi_sent_count;
long long global_time_callsite_count;

int tableSize;
h_t *task_callsite_stats;
callsite_stats_t *rawCallsiteData;
h_t *global_callsite_stats;
h_t *global_callsite_stats_agg;
h_t *global_MPI_stats_agg;

mpiPi_lookup_t *lookup;

int stackDepth;
double reportPrintThreshold;
int baseNames;
MPIP_REPORT_FORMAT_TYPE reportFormat;
int calcCOV;
int do_lookup;
int inAPIrtb;
int messageCountThreshold;
long text_start;
int obj_mode;
int printRankInfo;
enum mpiPi_report_style
```

```
  { mpiPi_style_verbose, mpiPi_style_concise, mpiPi_style_both }
     report_style;
  int print_callsite_detail;

  int collective_report;
#ifdef SO_LOOKUP
  so_info_t **so_info;
  int so_count;
#endif
  int disable_finalize_report;

  int do_collective_stats_report;
  mpiPi_histogram_t coll_comm_histogram;
  mpiPi_histogram_t coll_size_histogram;
  double coll_time_stats[(mpiPi_DEF_END - mpiPi_BASE)][32][32];

  int do_pt2pt_stats_report;
  mpiPi_histogram_t pt2pt_comm_histogram;
  mpiPi_histogram_t pt2pt_size_histogram;
  double pt2pt_send_stats[(mpiPi_DEF_END - mpiPi_BASE)][32][32];

}
mpiPi_t;
```

7.5 Refactored MpiP Wrapper Example

Listing 7.4: Implementation of Simple tool wrapper

```
static int mpiPif_MPI_Attr_get( jmp_buf * base_jbuf, MPI_Comm * comm, \
int * keyval, void *attr_value, int *flag,int i, vector* v) {
 // ---------------------//
 gst* gs_ptr = (( gst*)(QMPI_GET_CONTEXT);
 // ---------------------//
 int rc, enabledState;
```

```
 double dur;
 int tsize;
 double messSize = 0.;
 double ioSize = 0.;
 double rmaSize =0.;
 mpiPi_TIME start, end;
 void *call_stack[MPIP_CALLSITE_STACK_DEPTH_MAX] = { NULL };

if (gs_ptr->mpiPi.enabled) {
mpiPi_GETTIME (&start);
if ( gs_ptr->mpiPi.stackDepth > 0 )
mpiPi_RecordTraceBack((*base_jbuf),call_stack, gs_ptr->mpiPi.stackDepth);
}

enabledState = gs_ptr->mpiPi.enabled;
gs_ptr->mpiPi.enabled = 0;

// ---------------------//
//rc = PMPI_Attr_get( * comm, * keyval, attr_value, flag);
void* f=NULL;
QMPI_Table_query( 14, &f, (*vector_get(v,i)).table);
exec_func(f,i,14,v, * comm, * keyval, attr_value, flag);
// ---------------------//

gs_ptr->mpiPi.enabled = enabledState;
if (gs_ptr->mpiPi.enabled) {

mpiPi_GETTIME (&end);
dur = mpiPi_GETTIMEDIFF (&end, &start);

if ( dur < 0 )
  mpiPi_msg_warn(i,v,"Rank %5d:Negative time difference:%11.9f in %s\n",\
  gs_ptr->mpiPi.rank, dur, "MPI_Attr_get");
else
  mpiPi_update_callsite_stats(mpiPi_MPI_Attr_get, gs_ptr->mpiPi.rank,\
  call_stack, dur,(double)messSize,(double)ioSize,(double)rmaSize,i,v);
}
```

```
return rc;
}

extern int TMPI_Attr_get ( MPI_Comm comm, int keyval, void *attr_value,\
int *flag,int i,vector * v)
{
int rc;
jmp_buf jbuf;

rc = mpiPif_MPI_Attr_get( &jbuf, & comm, & keyval, attr_value, flag,i,v);

return rc;
} /* MPI_Attr_get */


extern void F77_MPI_ATTR_GET(MPI_Fint * comm, int * keyval,\
void *attr_value, int *flag, MPI_Fint *ierr, int i, vector* v){
int rc;
jmp_buf jbuf;
MPI_Comm c_comm;

c_comm = MPI_Comm_f2c(*comm);

rc = mpiPif_MPI_Attr_get( &jbuf, &c_comm, keyval, attr_value, flag ,i,v);

*ierr = (MPI_Fint)rc;
return;
} /* mpi_attr_get */
```

Bibliography

[1] *AMG Summary v1.0*. June 2013. URL: https://asc.llnl.gov/sequoia/benchmarks/AMG_summary_v1.0.pdf.

[2] J. Dongarra, P. Luszczek, and M. A. Heroux. *HPCG Technical Specification*. Nov. 2013. URL: https://prod.sandia.gov/techlib-noauth/access-control.cgi/2013/138752.pdf.

[3] M. P. I. Forum. *MPI: A Message-Passing Interface Standard - Version 3.1*. June 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[4] *HPCG*. July 2014. URL: http://www.icl.utk.edu/files/print/2017/hpcg-sc17.pdf.

[5] *HPCG Performance Benchmark and Profiling*. July 2014. URL: http://www.hpcadvisorycouncil.com/pdf/HPCG_Analysis_and_Profiling.pdf.

[6] *IBM Spectrum MPI V10.1 documentation*. 2017. URL: https://www.ibm.com/support/knowledgecenter/en/SSZTET_10.1/smpi_welcome/smpi_welcome.html.

[7] *MPI: A Message-Passing Interface Standard*. 1994, pp. 1–4. URL: https://www.mpi-forum.org/docs/.

[8] *mpiP: Lightweight, Scalable MPI Profiling*. Mar. 2014. URL: http://mpip.sourceforge.net/#Using.

[9] S. S. Sawilowsky. "New effect size rules of thumb." In: (2009).

[10] M. Schulz and B. D. Supinski. "A Flexible and Dynamic Infrastructure for MPI Tool Interoperability." In: *2006 International Conference on Parallel Processing (ICPP06)* (Oct. 2006). DOI: 10.1109/icpp.2006.6.

[11] M. Schulz and B. R. D. Supinski. "PNMPI tools: A Whole Lot Greater Than the Sum of Their Parts." In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC 07* (2007). DOI: 10.1145/1362622.1362663.

[12] *SuperMUC Petascale System*. Nov. 2017. URL: https://www.lrz.de/services/compute/supermuc/systemdescription/.