Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Master Thesis

Enabling Single Process Testing of MPI in Massive Parallel Applications

Scientific Computing Group
Department of Informatics

submitted by

Tareq Kellyeh
from Syria

March 20, 2018

Matr. No.:	6618727
Course of Study:	M.Sc. Informatics
Supervised by:	Dr. Julian Kunkel & Christian Hovy
1st evaluator:	Prof. Dr. Thomas Ludwig
2nd evaluator:	Dr. Julian Kunkel

Abstract

While many parallel programming models exist, the dominant model is MPI. It has been considered as the *de facto* standard for building parallel programs that use message passing. In spite of this popularity, there is a lack of tools that support testing of MPI programs. When considering unit testing, it is not widely applied to scientific programs, even though it is an established practice in professional software development. However, with MPI, the communicated data comes from different processes which increases the effort of creating small test units.

In this thesis, a solution to reduce the effort of testing massive parallel applications is developed. By applying this solution, any selected piece of MPI parallelized code that forms a part of such applications can be tested. The used method is based on the technique: *Capture & Replay*. This technique extracts data while executing the application and uses this data as an input for the MPI communications in the test phase. The structures, that contain the extracted data, are generated automatically.

As a step towards enabling *Unit Testing* of MPI applications, this thesis supports the user in writing appropriate test units and executing them by a single process solely. In this way, repeating the expensive parallel execution of MPI programs can be avoided. This step is considered as the most important contribution of this thesis.

Acknowledgments

My grateful thanks to my main supervisor Dr. Julian Kunkel for his kind supervision, helpful advice, and proofreading.

Many thanks to my secondary supervisor Christian Hovy for his valuable feedback, especially regarding the drawn figures.

Many thanks to Kevin Dwenger for reviewing this thesis and correcting some English mistakes.

I also want to appreciate the valuable support of my parents: Nahed (Emmi) and Mustafa, my siblings: Muhammad Ali (Hend), Ahmad (Sarah), and Nour (Anwar), and Hala.

During the whole period of writing this thesis, I haven't forgotten my homeland. I have always thought of the families and children there. If you read these words, please pray for *Syria*.

Contents

1 Introduction

This chapter motivates the thesis by giving an overview of how testing can be useful in the context of parallel applications. Section 1.1 provides an in-depth look into the obstacles to writing test units for MPI programs. Section 1.2 provides a primitive code example in order to demonstrate some issues when creating a test for an MPI program. Different possibilities to test MPI applications are discussed in Section 1.3. The goals of this work are presented in Section 1.4. Section 1.5 illustrates the terms used in this thesis. Finally, the structure of the thesis is listed in Section 1.6.

1.1 Motivation

When developing software applications, *testing* is seen as a key to assess their correctness. One of the most important benefits of testing is to reveal bugs in the code under test (CUT[1]). This is usually done by observing the response of the code to a certain input and then comparing this response to the expected one. Normally, the expected response is given by a *Test Oracle*. Such a test oracle can be a tester or the developer himself, the result of another implementation, or a described solution [8].

Besides testing expected behavior, a test can trigger exceptional scenarios that are likely to lead to a failure. For instance, it can be discovered if a biased input produces an unintended output. Even if the intended output is unknown, it can be checked if the code terminates badly, or even crashes [7].

Tests are also considered as a documentation for the CUT, because it shows the aim of this piece of code in addition to the cases that are already accounted. Therefore, tests are also another way to describe a problem and its solution. For these reasons, testing is an important method to ensure code quality.

However, testing parallel programs is considered to be difficult. This difficulty comes from the fact that failures in the parallel programs are caused not only by all the bugs known from the serial programming but also by bugs emerging from the interaction of several parallel processes [6]. This interaction may lead to deadlocks or race-conditions which are difficult or impossible to capture at compile time. Programming models such as MPI for distributed memory cause transferring data from process to process and maybe from node to node. This transformation implicitly means that each process receives data from another process and may send data to the others. As a result, it

[1] Code Under Test

would not be possible to test part of an MPI program with one process, because the input data may not be available, which prevents knowing the right output [1].

Beyond that, MPI programs may contain *user-defined datatypes* that specify a sequence of basic datatypes and integer displacements describing the data layout in memory. These datatypes are passed from subroutine to subroutine. When writing a test for a single subroutine, it is necessary to initialize these datatypes but it can be complicated to manually determine their layout in order to reconstruct them. For the mentioned reasons, testing of MPI programs is not straightforward.

1.2 Example MPI Program

Figure 1.1 shows a representation of a compiled MPI program which is executed by three processes. These processes exchange data using single point-to-point communications (`MPI_Send()`, `MPI_Recv()`) where `Rank 0` receives data from `Rank 2` and sends data to `Rank 1`. `Rank 2`, in turn, sends data to `Rank 0` and `Rank 1` after receiving data from `Rank 1`. Since the processes communicate with each other, it is not possible to test a part of the code that is executed by some of them. For instance: the code responsible for the execution of the red areas in the figure.

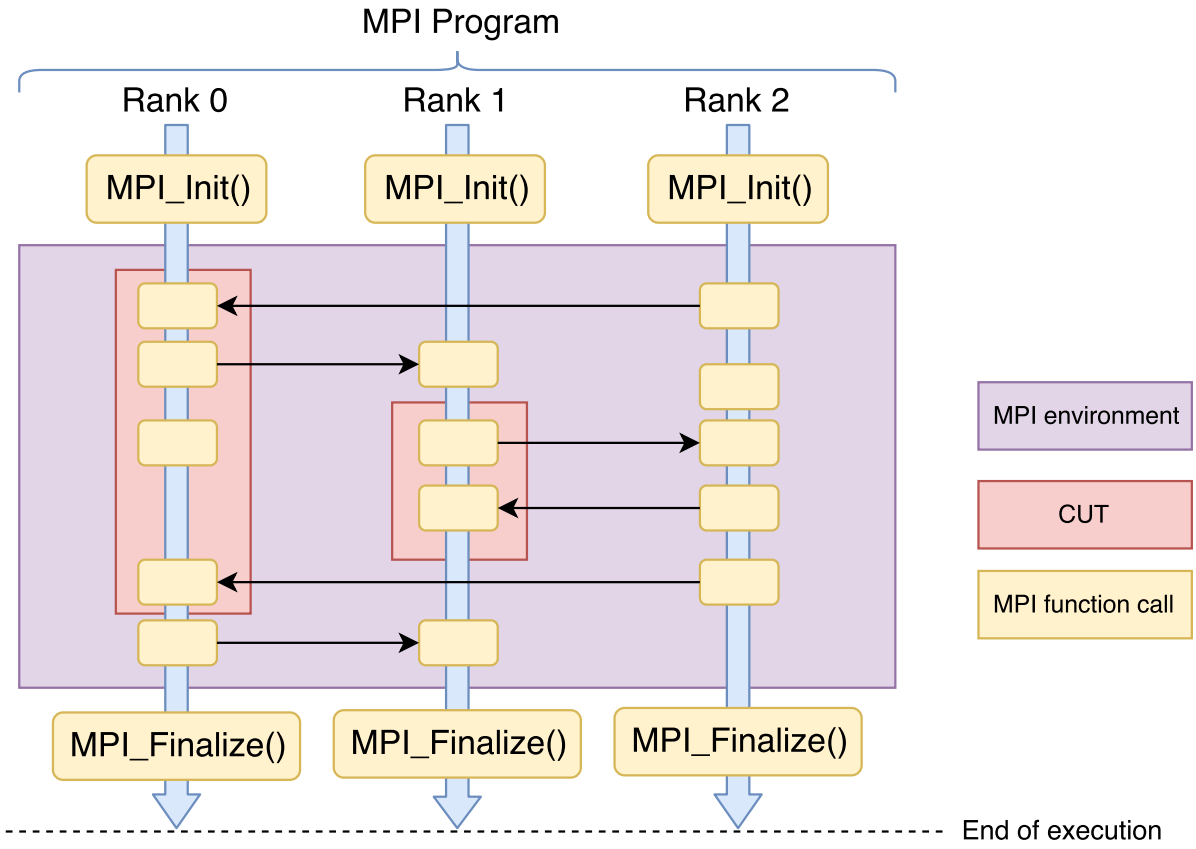


Figure 1.1: Representation of an MPI program.

Unit Testing When considering unit testing with MPI, some subroutines may require input that stems from another process. Testing such subroutines with one process is not possible as the important input is not available and checking simply the output would not have sufficed. To illustrate this, Listing 1.1 shows a section of an MPI program that models **Conwayś Game of Life** [13]. This section is a subroutine that applies the rules of the game repeatedly to create further generations depending on the initial state given by the user. To accelerate the generation process, this subroutine exchanges data from connected regions between processes.

```
1  void model_timestep(model_data_t * data)
2  {
3    // communication
4    // send row with: data->y_start to the process before (potentially wrap)
5    // send row with: data->y_end to the process after (potentially wrap)
6    int proc_before = (o.rank == 0) ? (o.size -1) : (o.rank - 1);
7    int proc_after = (o.rank + 1) % o.size;
8    MPI_Request request[4];
9    MPI_Status status[4];
10
11   MPI_Isend(d[1], o.x, MPI_CHAR, proc_before, 4712, MPI_COMM_WORLD, &
         ↪ request[0]);
12   MPI_Isend(d[data->y_count], o.x, MPI_CHAR, proc_after, 4711,
         ↪ MPI_COMM_WORLD, & request[1]);
13   // recv row data->y_start - 1 from the process before
14   // recv row data->y_end + 1 from the process before
15   MPI_Irecv(d[data->y_count + 1], o.x, MPI_CHAR, proc_after, 4712,
         ↪ MPI_COMM_WORLD, & request[3]);
16   MPI_Irecv(d[0], o.x, MPI_CHAR, proc_before, 4711, MPI_COMM_WORLD, &
         ↪ request[2]);
17   MPI_Waitall(4, request, status);
18   // compute next generation
19     .......
20     ......
21 }
```

Listing 1.1: Example: A subroutine that issues MPI calls.

Listing 1.2 shows a pseudocode for creating a test unit for this subroutine. Since each process that executes this subroutine calls the routines `MPI_Isend()` and `MPI_Irecv()` in order to communicate with other processes, running this test unit by a single process is not possible. However, this test unit does not test the data sent within the subroutine. To achieve that, the code that receives this data has to be tested.

```
1  model_data_t* data = prepare_test_data();
2
3  if (checksum(data) != abc){ // Check that the input data is correct
4      exit(1);                // Test failed!
5  }
6
7  model_timestep(data);
8
9  if (checksum(data) != xyz){ // Check that the output data is correct
10     exit(1);                // Test failed!
11 }
```

Listing 1.2: Pseudocode for creating a test unit for a subroutine that issues MPI calls.

1.3 MPI Testing Alternatives

In general, there are three alternatives for testing MPI programs:

1. **Creating test units manually**: when creating test units manually, the structure of the input data that has to be passed to the targeted subroutine as an argument may be large and complicated. Hence, reproducing and initializing it manually could be a very difficult and time-consuming task. Moreover, this approach does not solve the problem of calling MPI functions in the code.

2. **Code analysis**: analyzing the code may be a difficult task because of the possible complexity of the communication structure among processes. Above that, executing MPI code by more than one process increases the difficulty of analyzing this code.

3. **Capture/Replay**: this technique extracts data exchanged via MPI during the original execution of the application. Afterwards, it uses the extracted data as test input data to replay the original execution. Hence, it would be possible to use this data to test any MPI function individually. Moreover, since the captured data will be available after capturing, inspecting this data can be used to test the data sent by the tested code.

1.4 Goals

This work aims to build a library that significantly reduces the effort of creating test units for MPI programs. To achieve this goal, a technique for capturing and replaying the execution of a selected section from the MPI program is explored. Given an MPI program, the technique allows selecting a piece of code for testing, capturing the dynamic interactions between the processes that execute this code, and replaying the execution of any specific process in isolation.

This task can roughly be split into the following steps:

1. Building an instrumentation library for intercepting the MPI calls and recording the interactions performed in the CUT.

2. Building a replay library to emulate the original MPI interactions by using the data recorded from the previously intercepted execution.

Capturing the interactions between processes impacts storing the data exchanged in these interactions. This data should be saved in a standard file format. This format should be platform-independent and flexible and offer the ability to store data of any type or layout.

1.5 Terminology

In this thesis, the piece of code selected by the user for testing is referred to as the *code under test (CUT)*. Whereas the term *external code* indicates the rest of the application including its code and used libraries. The term *original execution* denotes the execution of the targeted program in the normal mode, i.e., without instrumenting. The output of this execution is referred to as the *original output*.

Replaying the execution from the perspective of a specific process means replaying the interactions made by this process solely. The term *replayed execution* refers to this execution. This process, in turn, is referred to as the *replayer process*.

1.6 Thesis Outline

This chapter motivated the topic of the thesis and provided an overview of the current challenges in constructing unit tests for MPI programs in particular. In Chapter 2, background information that is needed for understanding the thesis is provided. Armed with the information from the previous chapters, the suggested design for achieving the goals of the thesis is proposed in Chapter 3. Implementation details are covered in Chapter 4. In Chapter 5, the quality of the result is evaluated. The thesis concludes with Chapter 6 that summarizes the results and provides an outlook on future work.

Chapter Summary

This introductory chapter outlined the scope and importance of testing the parallel applications as well as highlighted some obstacles and challenges which occur when testing such applications. The basic alternatives for creating tests for MPI programs were briefly discussed, and the reasons for choosing one of these alternatives to ease the testing procedure have been sketched.

Next, the background information for this work is presented.

2 Background

Testing parallel applications and, therefore, MPI programs is considered to be difficult. Section 2.1 covers the challenges and shortcomings that one faces when testing such applications. Section 2.2 is dedicated to give a background of the MPI library, including some code examples and a view of the various implementations of this standard. Section 2.3 takes a look at the HDF5 file format which is used for storing and exchanging test data in this work. It is followed by an overview of the most popular performance analyzing tools that are related to this work in Section 2.4

2.1 Test Practice in Parallel Applications Development

The dominant parallel programming model for communication in high-performance computing (HPC) is message-passing. The interface that is commonly used for implementing it is MPI. Despite the diversity of the uses of the libraries that implement the MPI standard, conducting scientific experiments by scientists is the typical use case. For instance, simulation of physical phenomena, such as global climate change, or nuclear reactions [11].

Scientists have become dependent on computer-aided research, where other forms of experimentation have become difficult, e.g. manual experimentation. Despite this fact, they still develop their software without utilizing software engineering practices like testing [10]. One reason for this is the complexity of this software. According to Heaton et al., [10], testing scientific software is much more complicated than testing of traditional software. The characteristics of scientific software itself and the differences between scientific software developers and traditional software developers stand behind this complexity [9].

Kanewala et al. mentioned the challenges that arise when testing scientific software [9]. The first category of these challenges is related to the development of test cases. For instance, choosing a sufficient set of test cases is complicated due to the large number of input parameters needed by some scientific software. The large number of possibilities necessary to test the software on the system level and the lack of real world data that can be used for testing complicate the testing process more and more.

The absence of the *Test Oracle* is another obstacle that occurs while testing scientific software according to Kanewala et al. [9]. This test oracle can be absent during the development phase, since sometimes scientific software is written to find answers that

are yet unknown. Sometimes there is no single correct output for a given set of input. This increases the difficulty of testing.

Testing can also be a time-consuming task because of the long execution time of some scientific software which does not make it possible to run a large number of test cases to cover a certain criteria [9].

Additionally, scientists aim to do science, not software engineering [11]. They focus on scientific results rather than the quality of the code. Their lack of understanding of software engineering leads to underestimation of the value of software quality. The difference in the purpose of the software developed by the scientific community and the IT industry forms the second category of challenges in testing scientific software.

2.2 Message Passing Interface (MPI)

Parallel applications are distributed across large systems. They consist of separate processes, each with its own address space. These processes are decentralized and communicate using messages. The Message Passing Interface [3] is a standard for performing message passing. It allows programmers to write parallel applications where an application passes messages among processes in order to perform a task. It defines syntax and semantics for a set of interface functions which can be accessed from C or Fortran.

In this section, the focus is placed on the general aspects and concepts imposed by the interface. Hence, features are only listed if they are relevant to this thesis. This section is based on [2].

2.2.1 Basic concepts

Some classic concepts form the basis of the MPI design. The notion of a **communicator** is one of them. A communicator defines a group of processes that have the ability to communicate with each other. In this group, each process is assigned a so called rank, which is an integer number. Each rank is a unique identifier for the process inside the group and used explicitly by the other processes to address it for the purpose of communication.

The predefined `MPI_COMM_WORLD` is the initial communicator of all processes, i.e., each process can communicate within it after the initialization. This communicator is defined once `MPI_Init()` has been called. In addition, the communicator `MPI_COMM_SELF` is provided, which includes only the process itself.

Every newly created communicator is derived from another existed communicator. The function commonly used for this purpose is `MPI_Comm_split()` which has the following prototype [3]:

```
MPI_Comm_split(
    MPI_Comm comm,       // The parent communicator
    int color,           // All processes with the same color
    int key,
    MPI_Comm* newcomm)   // output
```

This function partitions the group associated with **comm** into disjoint subgroups, one for each value of **color**. Each subgroup contains all processes of the same **color**. Within each subgroup, the processes are ranked in the order defined by the value of the argument **key**, with ties broken independently of their rank in the old group. A new communicator is created for each subgroup and returned in **newcomm**. It is important to know that the original communicator does not go away, but a new communicator is created by each process.

Point-to-Point communication

The foundation of communication in MPI is built upon sending and receiving data among processes. There are two types of such communications. The basic type is known as point-to-point communication where data is transmitted between two, and only two, different processes: one process sends and the other one receives.

The respective communication routines can be used in blocking and non-blocking mode. In the first mode, the routine does not return until the communication is finished, while in the second one, the routine returns immediately even if the communication is not finished yet. Only the second mode can be used to perform further computation during the transmission of data. However, in this mode, the programmer must worry about whether the sent data is out of the send buffer, and whether the received data has finished arriving.

Assume two processes **A** and **B** exist. MPI send and receive calls operate in the following manner: first, **A** decides to send a message to **B**. **A** then packs up all the necessary data into the buffer. After that, the communication device (which is typically a network in a distributed memory architecture or a memory in a shared memory architecture) is responsible for routing the message to **B** depending on its rank.

In the blocking mode, **B** still has to acknowledge that it wants to receive data from **A**. Once it does, the data is transmitted. **A** is notified that the data has been transmitted and may continue computation.

MPI also allows senders and receivers to specify a message ID with each message (known as tag). That is useful to differentiate messages in cases where a process has to send many different messages to another process. When the destination process only wants to receive a message with a certain tag, messages with different tags are buffered by the network until this process is ready for them. The prototypes for the MPI sending and receiving functions are [3] :

```
MPI_Send(
    void* data,              // The address of data buffer
    int count,               // The count of sent elements
    MPI_Datatype datatype,   // The type of the transmitted elements
    int destination,         // The rank of the receiving process
    int tag,                 // The message ID
    MPI_Comm communicator    // The communicator, in which the operation occurs
    )
```

```
MPI_Recv(
    void* data,
    int count,           // The max. number of elements that can be received
    MPI_Datatype datatype,
    int source,          // The rank of the sending process
    int tag,
    MPI_Comm communicator,
    MPI_Status* status   // It provides information about the received message
    )
```

Collective Communication

In contrast to point-to-point communication, which is communication between two processes, collective communication is a method of communication which involves participation of all processes in a group specified by a communicator.

MPI provides many collective communication functions. For the purpose of synchronization it provides the function `MPI_Barrier`, which has the following prototype [3]:

```
MPI_Barrier(MPI_Comm communicator)
```

This function blocks until all processes in the communicator call it. After that, they can all resume execution again.

MPI Broadcasting

The standard collective communication technique is **Broadcasting**. When broadcasting, the same data from one process is sent to all processes in a communicator including the sending process itself. In MPI, broadcasting can be accomplished by using `MPI_Bcast()` which has the following prototype [3]:

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)
```

The root process and the receiver processes call this function using the same arguments, although they do different jobs. On return, the contents of the root's communication buffer has been copied to all processes.

MPI Reduce and Allreduce

The function `MPI_Reduce()` combines the elements provided in the input buffer of each process in the group, using a *Combiner* to build up a result and returns it in the output buffer `recv_data` of the process with rank `root`. This Combiner is a function that applies an operation `op` on two elements and returns one element. This function applies the operation again on the returned element and the following elements from the list respectively. `MPI_Reduce()` has the following prototype [3]:

```
1 MPI_Reduce(
2     void* send_data,         // The input data
3     void* recv_data,         // The result
4     int count,
5     MPI_Datatype datatype,
6     MPI_Op op,
7     int root,
8     MPI_Comm communicator)
```

Sometimes it may be required to access the reduced results across all processes rather than the root process. The function `MPI_Allreduce()` is used for this purpose. In other words, `MPI_Allreduce()` is the equivalent of doing `MPI_Reduce()` followed by an `MPI_Bcast()`. It has the following prototype [3]:

```
1 MPI_Allreduce(
2     void* send_data,
3     void* recv_data,
4     int count,
5     MPI_Datatype datatype,
6     MPI_Op op,
7     MPI_Comm communicator)
```

2.2.2 MPI Profiling Interface

The MPI standard defines a profiling interface (PMPI) that allows the programmers to create profiling libraries by wrapping any of the standard MPI routines. This interface has the following properties:

- It allows selective replacement of MPI routines at link time, which means that there is no need to recompile the program.
- Every MPI function also exists under the name `PMPI_`.

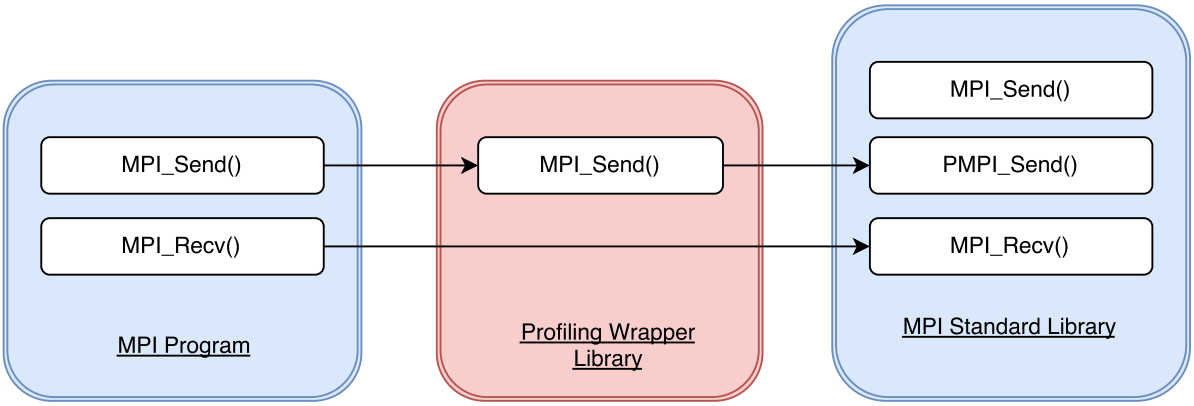


Figure 2.1: MPI Profiling Interface

Figure 2.1 shows the execution flow of the MPI routines in the presence of a profiling wrapper library. This library contains a subset of redefined `MPI_*` entry points, and inside these redefinitions, a combination of both `MPI_*` and `PMPI_*` symbols are called. This means that the programmer can write functions with the `MPI_*` prefix that call the equivalent `PMPI_*` function. Functions that are written in this manner behave like the standard MPI function, but can also exhibit any other behavior that may be added by the programmer of the wrapper library.

2.2.3 MPI Implementations

Multiple implementations for the MPI standard have been developed. Despite these different implementations, MPI defines a behavior, i.e., the semantics of operations, like the guarantee that the underlying transmission of messages is reliable. An MPI program is executed by a set of autonomous processes; each process executes the code in his own environment. The processes communicate via routines which are provided by the MPI interface. Following are the most popular MPI implementations:

- **Open MPI** [4]
  This implementation attempts to create the best MPI library available. It aims to provide support for the majority of established high performance interconnects, TCP/IP, and shared memory.
- **MPICH** [5]
  It is a high-performance portable implementation of the MPI standard developed by the Argonne National Laboratory. The primary goal of it is to be adaptable for any underlying system, i.e., to provide some kind of portability.

2.3 HDF5

The Hierarchical Data Formal (HDF) [12] is a technology designed to manage and store data collections of any size and any level of complexity. It was specifically designed for

flexible and efficient storage and I/O as well as for every type and size of system which encourages portability.

HDF5 has a few outstanding features in comparison to other file formats. For instance, HDF5 files are self-described and allow users to specify complex data relationships and dependencies like XML. In contrast to it, the access to any part of an HDF5 file is achieved directly without having to parse its whole content.

Similar to the tables in the relational database model, HDF5 supports n-dimensional datasets and each element in the dataset may itself be a complex object. In addition, HDF5 represents data objects in a hierarchical manner, similar to directories and files, in contrast to the flat organization of data objects in relational databases.

HDF5 consists of:

- An Abstract Storage Model.
- A Logical Data Model.
- A Physical Data Format.
- Libraries and tools for working with this format.

2.3.1 Logical Data Model

The Logical Data Model represents the HDF5 file as it appears to the user. According to this model, an HDF5 file seems like a rooted, directed graph. The nodes of this graph are named data objects whereas links are the directed arcs. In other words, an HDF5 file is a container that holds data objects. Each file has at least one object, the root group. All objects are members of the root group or descendants of it. These objects are groups, datasets, and other objects.

Group

The group and its members are similar to directories and files in UNIX. Every object in an HDF5 file has a unique identity and can be accessed only by its full path names within the hierarchy of the file. Figure 2.2 shows that an object, such as a dataset in a group, is defined by its group path. The objects can also be shared, so there can be multiple paths to the same objects. For instance, both paths `/A/D` and `/B/E` point to the same dataset.

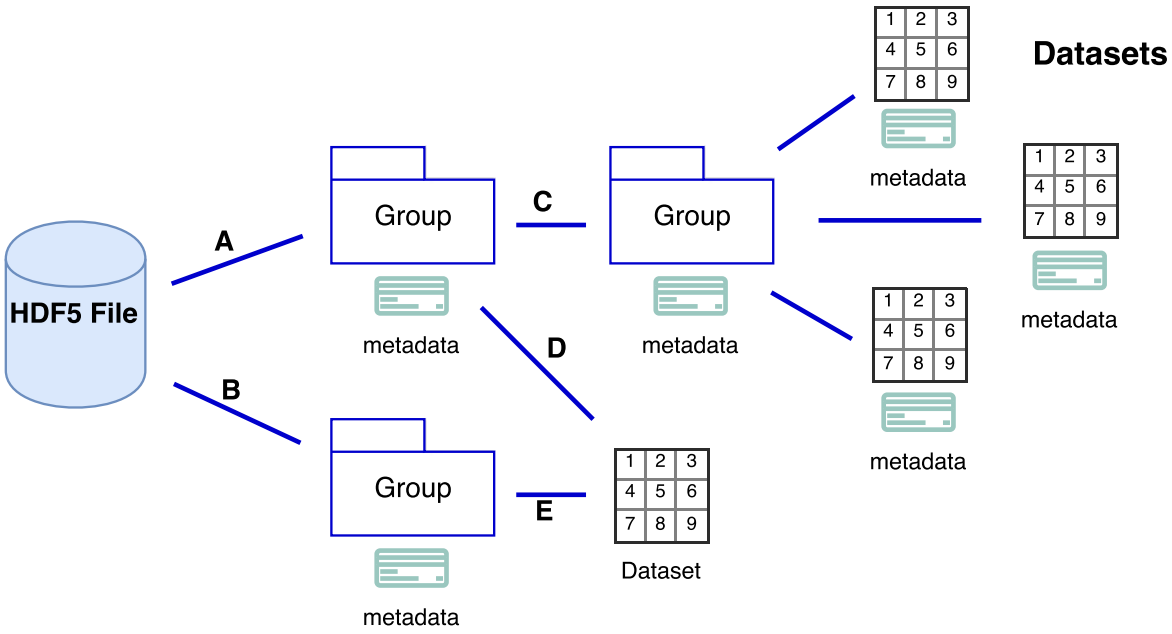


Figure 2.2: Groups in HDF5.

Dataset

The Dataset is a multidimensional array which contains and organizes identically typed data elements. It is made up of metadata that describes the data, in addition to the data itself. The minimum metadata necessary to describe any dataset consists of the following HDF5 objects:

- **Dataspace**: a description of the layout of the dataset's data elements.
- **Datatype**: a description of the individual data elements in the dataset.
- **Property List**: a collection of parameters controlling the options of the dataset.

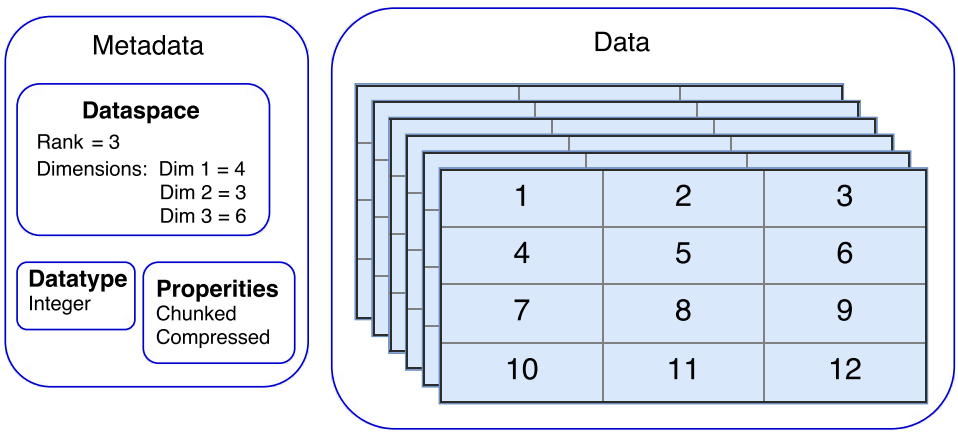


Figure 2.3: An instance of a dataset in HDF5.

Figure 2.3 shows an instance of a dataset where the data is stored as a three dimensional dataset having a size of 4 x 3 x 6 with an integer datatype. This dataset is chunked and compressed.

2.3.2 Programming Model

The programming model manipulates the objects from the abstract data model. The general paradigm for working with any HDF5 object is to:

- Open the object.
- Access the object.
- Close the object.

The library forces this order when working with any object by argument dependencies which means that no object can be accessed without opening it and once this object is closed it can no longer be accessed. In fact, it imposes an order when working with different objects. For example, a file must be opened before a dataset because the dataset open call requires a file handle as an argument. However, objects can be closed in any order.

2.4 Performance Analyzing Tools

Profiling is a technique for determining how a program uses (processor) resources. Principally, it is employed to determine how much of an application's run-time is attributable to the various sections of its code, how often a routine is called, and the identities of its calling routines. In this section, an overview of the most popular MPI profiling tools is introduced.

2.4.1 Intel Trace Analyzer and Collector

Intel Trace Analyzer and Collector [14] is used to find bottlenecks in parallel cluster applications quickly which allows to achieve a high performance for such applications. It evaluates profiling statistics and load balancing, identifies communication hotspots, and increases application efficiency.

2.4.2 PGPROF

PGPROF [15] is a performance profiler for MPI applications. It is used to visualize and diagnose the performance of an MPI program. It associates execution time with source code. PGPROF allows profiling at the function, source code line and assembly instruction level for Fortran, C and C++ programs. It provides views of the performance data for analysis of MPI communication, multi-process and multi-thread load balancing

and scalability. It collects counts of the number of messages and bytes sent and received. Figure 2.4 illustrates an MPI profile. This sample shows an example MPI profile with maximum times and counts in the Statistics Table, and per-process measurements in the Parallelism tab.

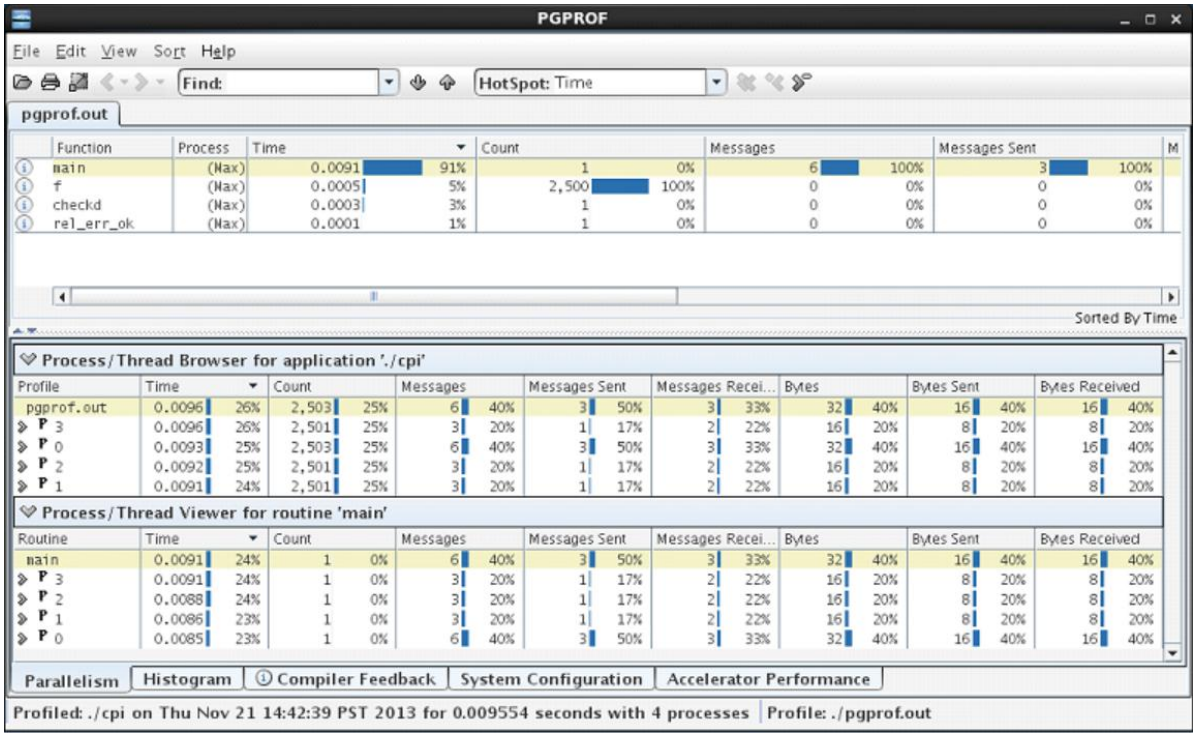


Figure 2.4: Sample MPI Profile [16]

The send and receive counts can be used for messages, the byte counts to identify potential communication bottlenecks, and the process-specific data to find load imbalances.

2.4.3 PGDBG

PGDBG [17] is a graphical debugger from the Portland Group. It may be used to debug parallel programs including MPI, OpenMP and hybrid MPI/OpenMP applications written in C, C++ or Fortran.

2.4.4 Marmot

Marmot [18] is an MPI checker which allows parallel programmers to isolate MPI-related programming errors in large scale parallel applications. It surveys the MPI calls made and automatically checks the correct usage of these calls and their arguments during run-time. It does not replace classical debuggers, but can be used in addition to them.

2.4.5 Vampir

Vampir [19] is an easy-to-use framework used to analyze arbitrary program behavior at any level of detail. The tool suite implements optimized event analysis algorithms and customizable displays that enable fast and interactive rendering of very complex performance monitoring data. It focuses on parallel applications where performance data is collected from multi-process (MPI, SHMEM), thread-parallel (OpenMP, Pthreads), as well as accelerator-based paradigms (CUDA, OpenCL, OpenACC).

Conclusion While all of these tools record the application behavior, they serve the purpose of analysis. None of them supports the writing/generating of test units for MPI code.

Chapter Summary

In this chapter, Some background to MPI library and its various implementations was presented. Moreover, an overview of the HDF5 data format was presented. The different performance analyzing tools were also introduced in nutshell.

The next chapter introduces the approach proposed to build easy-to-use capture and replay libraries for testing MPI programs.

3 Design

This chapter is dedicated to show the design of the solution developed to achieve the goals of this work. It does not focus on the actual implementation which will be discussed in Chapter 4. From the objectives of this work, some high-level user requirements can be derived. Section 3.1 specifies these requirements and views the solution from the user perspective. Section 3.2 recalls the requirements and objectives the solution should satisfy and states the constraints to which it has to remain committed. To realize these intentions, a technique for capture and replay executions is defined. This technique can be implemented in different ways. Section 3.3 is devoted to introduce the general approach in order to show these alternatives. Depending on the characteristics of each alternative, one solution is selected to implement the approach. This solution and the followed technique are described in detail in Section 3.4. After describing the selected solution, a general assessment of the desired requirements is performed in Section 3.5. In Section 3.6, an alternative solution, which applies the same approach, is introduced.

3.1 User Perspective

The following high-level user requirements can be derived from the goals of this work. As mentioned in Chapter 1, this goal is reducing the effort of testing MPI programs. The solution intended to achieve this goal should allow the user to select a piece of code to test it, the CUT. During run-time, the solution has to capture the execution of this piece solely. It should also be possible to capture the execution of the whole program.

Correspondingly, in the replay phase, only the execution of the selected CUT has to be replayed. Moreover, the solution has to replay the execution by using one single process which will enable the user to test the CUT from the perspective of any chosen process. In this way, repeating the expensive parallel execution of MPI programs can be avoided.

In addition, after capturing, the user has to be able to modify the captured data in order to produce different testing scenarios.

Above that, it should be possible to operate the solution on any system or platform and hence, the user will be able to create a test on one system and run it on another. Moreover, the user should not be restricted by any type or layout of data, i.e., the tool should be able to deal with data of any type or layout.

3.2 Requirements

As a result of the objectives mentioned above, the following requirements are important for the design and have to be satisfied:

1. **Usability**: since the proposed solution aims to ease testing of MPI programs, it should be practical and easy to use. When usability is concerned, three aspects have to be taken into consideration:

   a) Creating the test: the creation of the test data during the capture phase should not require any modification of the MPI library. Additionally, selecting the CUT should be simple so that it does not require significant modification of the application's source code. Calling a function to announce starting the CUT and calling another one to notify its ending should be adequate.

   b) Running the test: during replay, running large configurations has to be avoided. In addition, the applied solution should enable the studying of only one specific process in isolation.

   c) Deriving different test scenarios: after capturing, manipulating the captured data in order to produce different test scenarios has to be straightforward.

2. **Efficiency**: the intended solution has to be efficient during capturing from two perspectives:

   a) Storage usage: it has to capture merely the information related to the CUT during the original execution.

   b) Performance: when capturing the execution, the run-time of the application should not increase significantly.

   On other side, the solution should also be efficient in the replay phase, in that it replays just the execution of the CUT.

3. **Platform independency**: the solution should use a standard file format for exchanging the test data. This format has to be platform-independent and flexible and offer the ability to store data of any type or layout. Consequently, the created tests can be transferred between different systems and executed on any one of them.

3.3 Approach

As discussed in the introduction, the proposed approach depends on a technique for *capturing and replaying the executions* of MPI programs. *Capture & Replay* implies extracting data while running the program and using the extracted data afterwards as test input data to emulate the original execution. This technique is divided into two main phases: *capture* and *replay*. Figure 3.1 informally depicts the two phases.

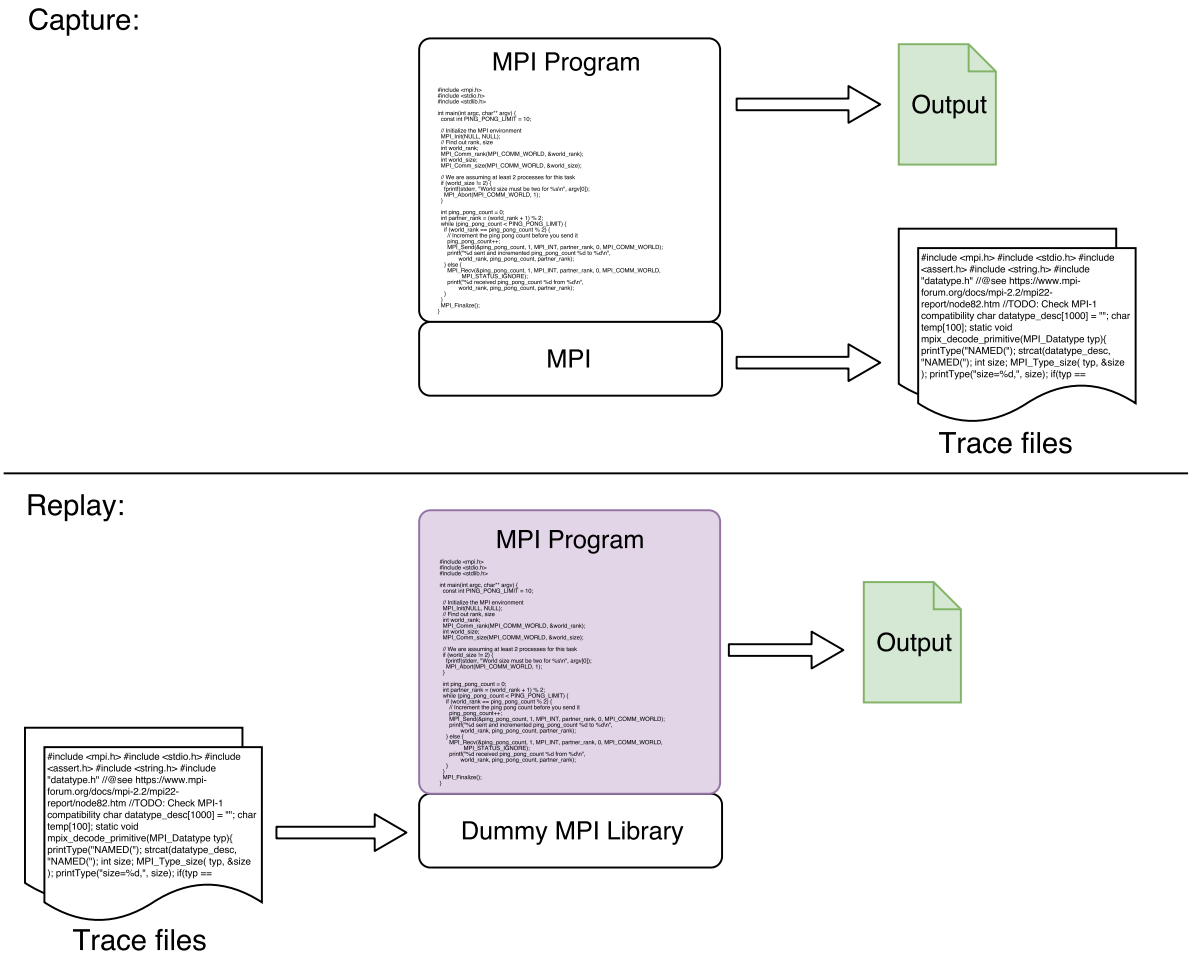


Figure 3.1: Overview of *Capture/Replay* technique.

Capturing The capture phase works by intercepting the MPI function calls in the program code. It serializes and stores the arguments of each function in trace files. Additionally, it stores the data transmitted between the processes when calling these functions. Since the capture takes place while the application is running (e.g., in the field or during testing), the output is identical to the original output in addition to the gained trace information.

Replay When the capturing is done, the trace information recorded in the capture phase is used as an input in the replay phase. To achieve this purpose, the technique provides a *replay scaffolding*. This scaffolding is made up of a dummy MPI library in addition to the trace information gained from the capture phase. The proposed MPI library does not conduct any communication but fetches the data from the recorded trace information.

Since MPI programs can be executed by multiple processes, replaying the execution from the point of view of any specific one of them eases the testing procedure. It reduces the

complexity of interacting multiple processes and shows the output in which this process has contributed. Replaying the execution from the perspective of one process implies imitating the interactions in which this process participated, without the need for the involvement of other processes.

3.3.1 Storing Trace Information

The trace information gained from the capture phase may be stored in different ways depending on the implementation of the proposed approach. Figure 3.2 shows alternative solutions that can be applied to implement this approach, each with different merits and flaws.

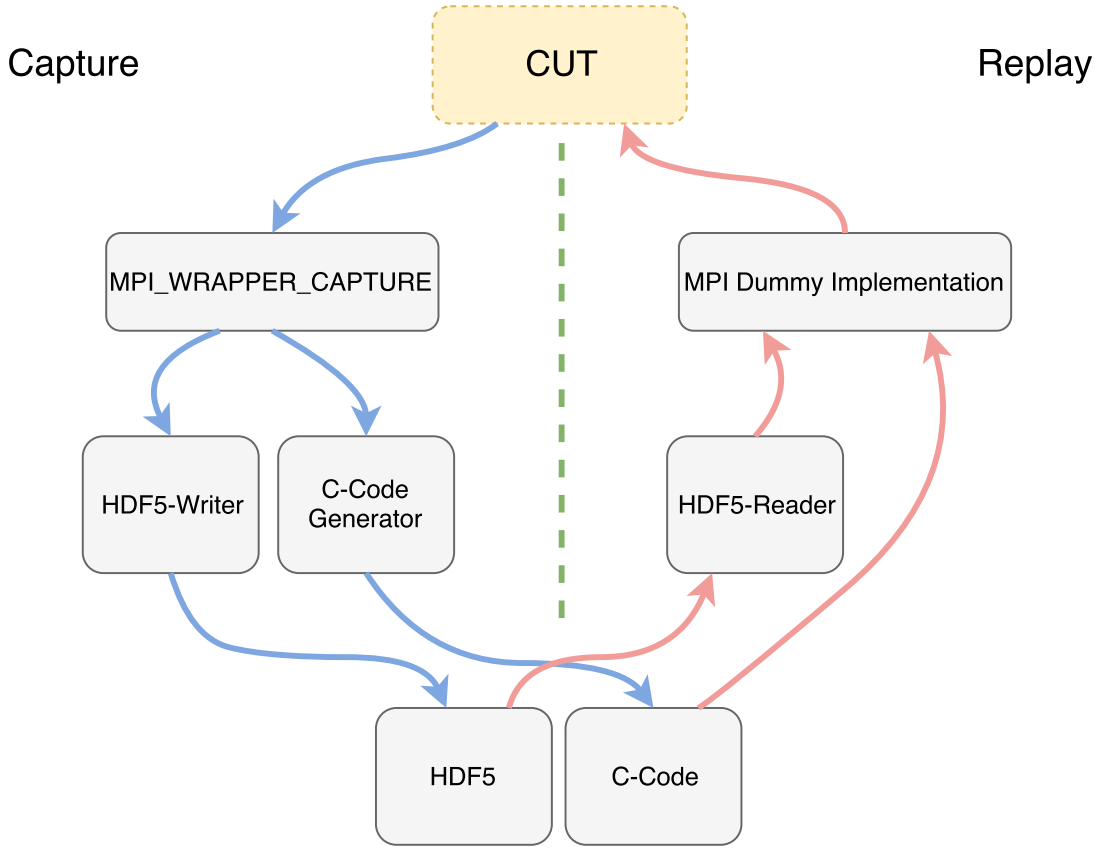


Figure 3.2: Implementation alternatives

Writing to/Reading from HDF5 data structures

In this solution, the trace information is stored in HDF5 data structures during run-time. For replay, an `HDF5-Reader` is required to fetch the data from these data structures. While replaying the execution, this reader acts as a data provider and is used by a dummy MPI library to respond to the MPI calls. The user can modify the data recorded in the HDF5 data structures to derive different testing scenarios. This solution is used to implement the approach in this work and is introduced in detail in the next section.

Source Code Generation

This solution generates source code and records the trace information as arguments in this code. In this solution, a dummy MPI implementation acts as an MPI-test library which is able to programmatically register the communicated messages. When executing the generated code, the MPI-test library receives the requests from this code and responds to them. To produce different testing scenarios, the user is able to modify the arguments in the generated code directly. A specific data reader is not required. The horizon of this solution is discussed in more detail in Section 3.6.

3.4 Technical Design

The solution introduced in this section is the one used to implement the approach discussed in Section 3.3. This solution consists of two libraries: one for capture and another for replay. The first library intercepts the MPI calls in any piece of code selected by the user (CUT). It records the captured information in HDF5 data structures which are automatically generated. These structures are used afterwards by the second library as a data provider in the replay phase. Since these structures are available after the capture phase, the user is able to manipulate the data contained in them in order to derive different testing scenarios. If the user changes the CUT to issue combinations for which no input exist, or to receive more messages than have been recorded, the implementation returns an error.

However, there are two alternatives for storing the captured information:

File per process In order to keep the information captured by each process isolated from that captured by the others, each process writes its information in a process-specific file. Hence, replaying the execution by using one single process implies fetching the data from the file written by this process appropriately.

Shared file An alternative to dedicate a file to each process could be to dedicate a file to each communicator. Thereby, all processes which participated in the communicator could write their trace information in this file. In this case, the collective `Parallel HDF5 API` imposes participation in operations on the file like create, open, and close, on all these processes. Further, it imposes the collaboration of these processes in any operation on any dataset stored in the file like creating and extending. The only exception from that is the array data transfer operation which can be collective or independent. For instance, if processes `A`, `B` and `C` were the participants of communicator `comm`, when `A` sends data to `B`, `A` will need the participation of `B` and `C` to store this data. That will produce both overhead and complexity to the data model which can be avoided when implementing the first suggested method. Besides performance, a drawback of this method is that data of all processes is kept while a test unit will run on individual processes only.

To better illustrate the above issues, Figure 3.3 shows an MPI program. A piece of its code is selected by the user for testing (CUT). Given this program is executed by n processes and the n processes executed the selected CUT, then the output will be n trace files in addition to the original output. Each trace file is process-specific and contains information about each interaction in which the process participated. Depending on this information, a developed dummy MPI library is able to emulate the original interactions of each process individually.

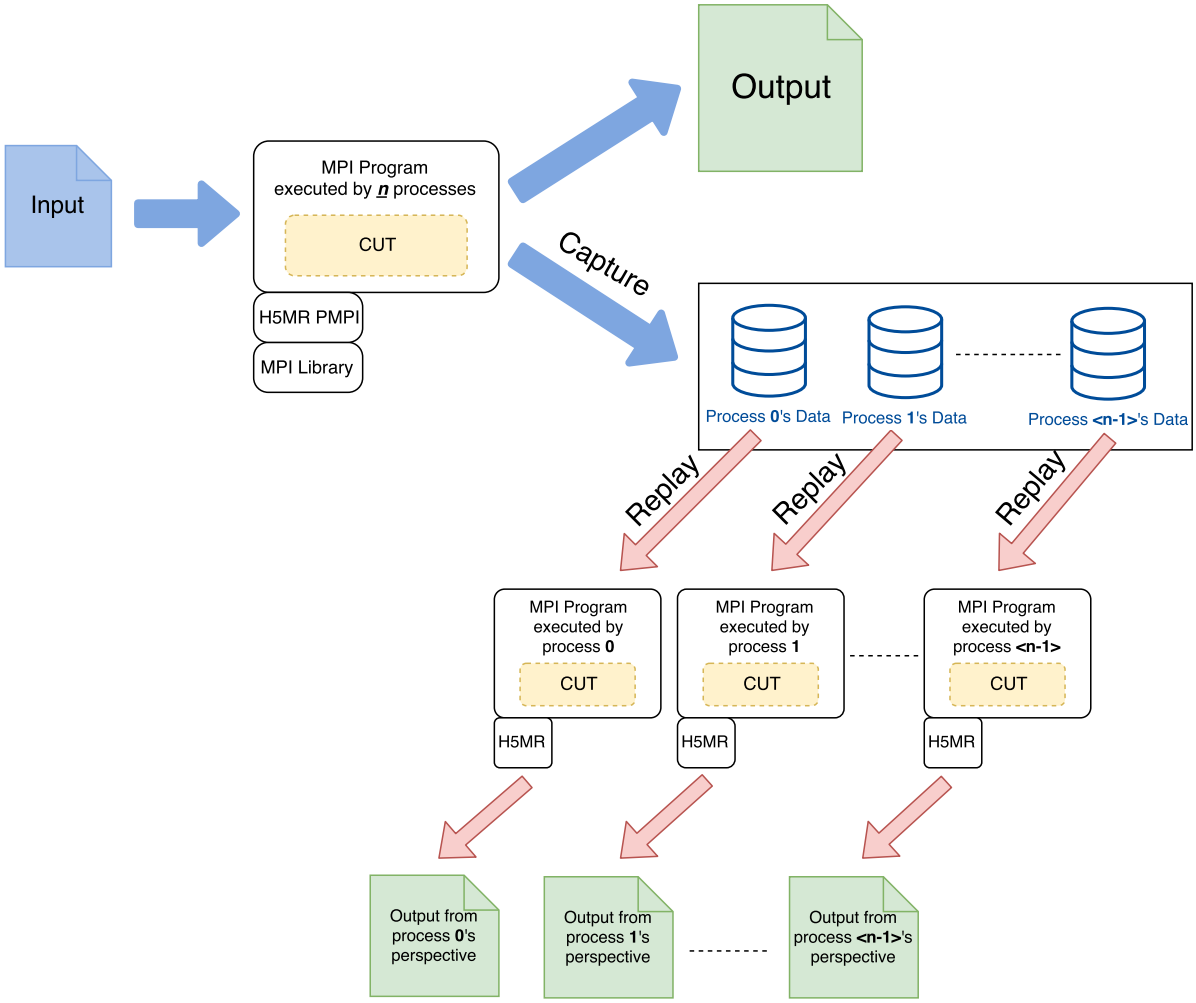


Figure 3.3: Technical Design

In general, the proposed solution works by:

1. Letting the user select a piece of code to test (CUT); he/she may specify it depending on run-time conditions like process rank or iteration number.

2. Automatically capturing at run-time all the interactions among all processes in the CUT.

3. Replaying the recorded interactions of any desired process in isolation.

3.4.1 Selecting CUT

Selecting a piece of code to test is the first step in the presented solution. After linking the developed library to the targeted program, three functions have to be called at most. A function to detect if the execution has to be captured completely or partially. If it is captured partially, two additional functions are required. The first one indicates the beginning of the CUT and the second one determines its ending.

Since the developed library replays the captured interactions solely, any attempt to change the CUT after capturing in order to include different interactions will result in a run-time error during replay. Replaying another interactions imposes capturing them beforehand. Listing 3.1 shows an example for selecting CUT.

```
1  #include <mpi.h>
2  #include <h5mr.h>  // Linking the library
3
4  int main(int argc, char** argv)
5  {
6      // h5mr_init(1); To capture the whole execution
7      h5mr_init(0);  // Capture the execution in manual mode
8      .....
9      MPI_Send(send_buffer, 3, v_datatype, 8, 53, MPI_COMM_WORLD);
10     h5mr_start_recording(); // Start capturing
11     MPI_Send(send_buffer, 3, c_datatype, 5, 52, MPI_COMM_WORLD);
12     MPI_Recv(receive_buffer, 3, v_datatype, 0, 53, MPI_COMM_WORLD, &status);
13     MPI_Recv(receive_buffer, 3, c_datatype, 0, 52, MPI_COMM_WORLD, &status);
14     h5mr_stop_recording(); // Stop capturing
15     ....
16 }
```

Listing 3.1: Pseudocode for selecting CUT.

3.4.2 Capture Phase

In this phase, all the communications during the execution of the selected CUT are captured. In order to capture the communication entirely, it is not sufficient to record the data transmitted among the processes that participated in the execution, it is also necessary to consider the arguments of each MPI function called to perform any operation in order to transmit this data. These arguments, among others, serve as metadata that describes the captured communication. As a consequence thereof, when the capture library intercepts an MPI call, it records trace information which includes two kinds of information:

1. The arguments in the intercepted MPI call.
2. The data transmitted by this call.

As mentioned previously, each process writes its captured information in a dedicated file. The files are named by the pattern `mpi-hdf5-recorder_<rank>.h5` where `<rank>` is the process's rank in the main communicator `MPI_COMM_WORLD`.

Capture Data Model

When transmitting data by MPI, the type of this data may range from primitive to composite and user-defined. Whereas capturing the data of primitive types can be done inexpensively, capturing the data of composite and user-defined types is computationally and space expensive and considered to be hard as well as complicated.

Figure 3.4 shows the data model proposed to capture trace information. This information includes the transmitted data in addition to metadata which describes each operation performed to achieve the transmission. This schema depicts the model from the process's perspective. A process's perspective describes which data structures are created by the process in order to trace all communications in which this process participated.

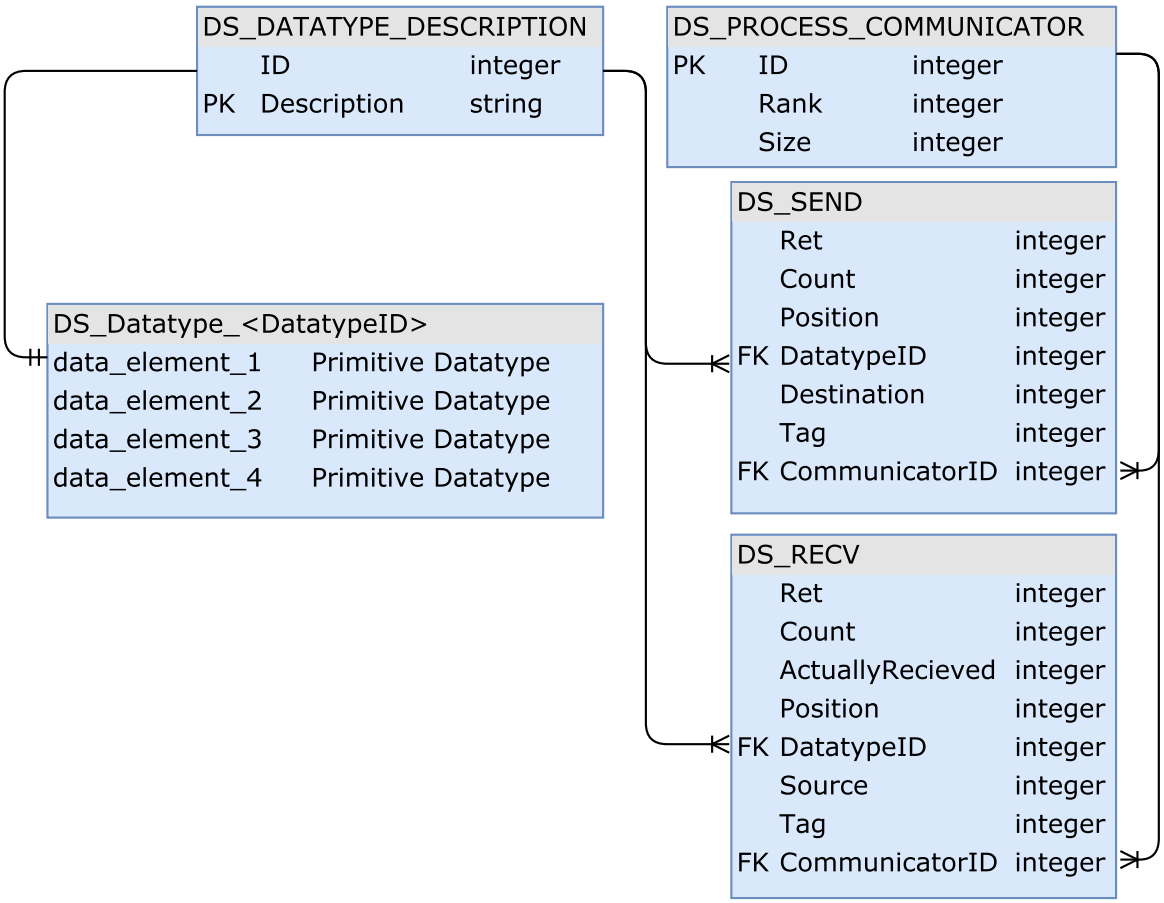


Figure 3.4: Capture Data Model

Storing Function Arguments Storing the arguments of each intercepted function in the capture phase is fundamental for applying the intended approach. These arguments are considered, among others, to be metadata that describes the operation performed by an MPI function call. According to these arguments, the necessary HDF5 data structures are created and extended over and over again to store the data transmitted by this call.

According to the proposed data model shown above, each operation has its own dataset that maintains metadata describing this operation. For instance, when intercepting an `MPI_Send()` call, metadata is stored in the dataset `DS_SEND`. Likewise, the dataset `DS_RECV` maintains metadata acquired when intercepting `MPI_Recv()` calls.

Basically, the metadata recorded when intercepting the asynchronous calls `MPI_Isend()` and `MPI_Irecv()` could be stored in the same datasets as `MPI_Send()` and `MPI_Recv()`. But for the aim of simplicity, this data is stored in different datasets, namely `DS_ISEND` and `DS_IRECV`.

For instance, when capturing a send operation, the metadata that describes this operation includes the following attributes:

1. The arguments of the function `MPI_Send()`, except the sent data itself.
2. `Ret`: an attribute that indicates whether the send operation performed successfully.
3. `Position`: an integer that indicates the position of the sent data in the dedicated dataset if the operation performed successfully. A negative integer otherwise. This is described in Section 4.5 on page 44.

For `MPI_Recv()`, a special case is that its semantic allows to receive less elements than specified with `count`. Therefore, the structure `status` has to be checked to reveal the number of elements that have been actually received. This number is stored in the field `ActuallyReceived`.

Data Model Normalization To keep the data model normalized, the scalar arguments are recorded in the datasets directly, whereas each struct argument is represented by a foreign key. For instance, when saving the arguments of the function `MPI_Send()`, the arguments `datatype` and `communicator` are represented in the dataset `DS_SEND` by the fields `DatatypeID` and `CommID` respectively. Each one, in turn, refers to the field `ID` in the datasets `DS_DATATYPE_DESCRIPTION` and `DS_COMMUNICATOR` accordingly.

The dataset `DS_DATATYPE_DESCRIPTION` maintains a description for each datatype used by the process in the program. This description is critical to distinguish between the different datatypes. In this dataset, each description (`Description`) is associated with a unique integer `ID`. Hence, this integer is used to represent the relevant datatype in the other datasets. Using the datatype description to identify it prevents double representation of those which have the same elements laid out in the same order with the same displacements from each other. As a result, the `DS_DATATYPE_DESCRIPTION` dataset serves as a reference for the datatypes used in the other datasets.

The same thing applies to the dataset `DS_COMMUNICATOR` where each communicator, in which the process participated, is represented by a unique identifier (`ID`). In addition, the size of this communicator (`Size`) and the rank of the process in it (`Rank`) are also recorded in this dataset. More details about the algorithm used to identify these communicators are given in Section 4.4 on page 44.

Storing Transmitted Data Coming back to the data model seen in Figure 3.4, the transmitted data is saved in different datasets according to its type. For instance, every time the process sends or receives data of the type `datatype_xy`, this data is saved in the dataset `DS_DATATYPE_123`, where 123 is the `ID` related to the description of the datatype `datatype_xy` in the dataset `DS_DATATYPE_DESCRIPTION`. The fact that the different datatypes have different structures is the main reason for that. Since HDF5 supports storing records of a well-defined datatype only and does not store records of generic types, it is a very difficult task to build a unified structure which is suitable for containing all these dissimilar datatypes.

Overview of the Capture Phase

Figure 3.5 depicts the capture phase entirely. It shows that each process executing the CUT produces a trace file. The other processes are not concerned with the capture procedure, and hence, do not need to capture any information.

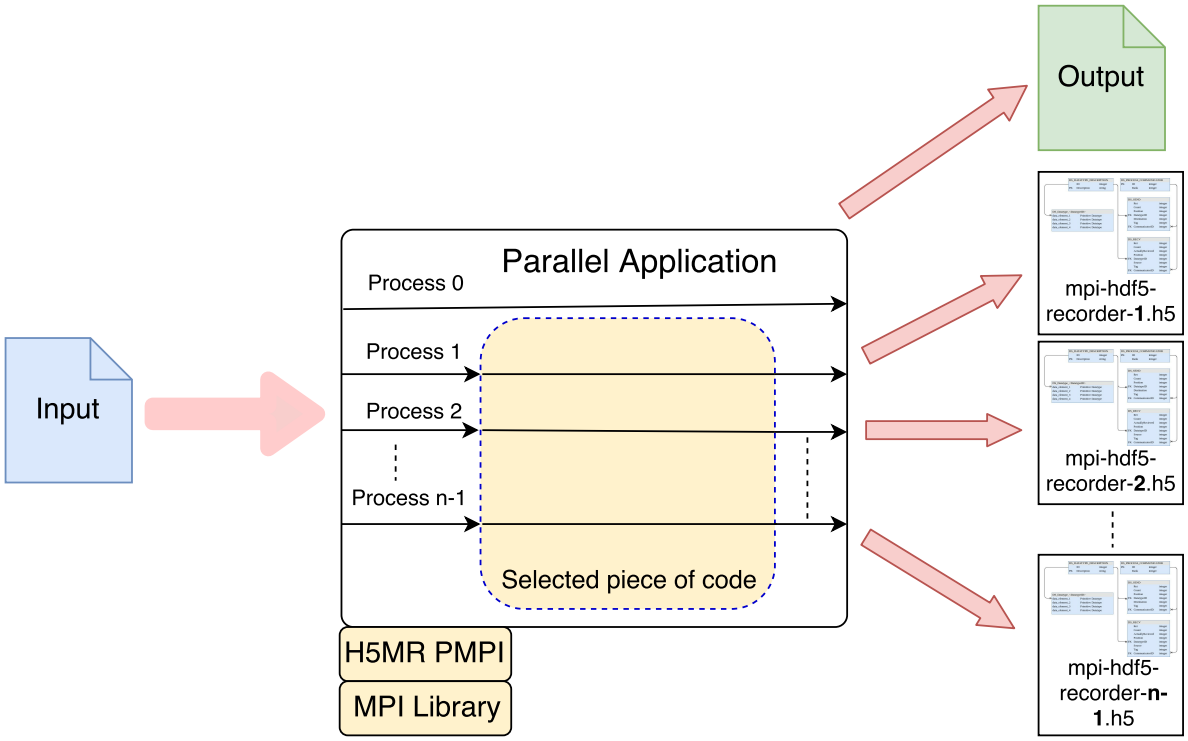


Figure 3.5: Capture Phase

To summarize, from the process's perspective the capture phase works by:

1. Creating a file to maintain the HDF5 data structures necessary for capturing.
2. Recording the arguments of each MPI call issued by the process.
3. Recording the data exchanged with the other processes.
4. Calling the corresponding regular MPI function.

3.4.3 Replay Phase

In this phase, the execution of the CUT is repeated. Whereas the original execution may be performed by one process or more, the replayed execution is always carried out by one process only. This process replays the execution of any specific one of those which participated in the original execution. In the replayed execution, the interactions with other processes performed by this process in the CUT are emulated. Hence, the output of this execution is the same output produced by the process in the original execution.

During replay, the followed technique acts as both a driver and a stub. It provides the scaffolding that imitates the communications between the replayer process and the other processes. This scaffolding is made up of a dummy MPI implementation in addition to the trace file that was written by the replayer process in the capture phase. In contrast to existing MPI implementations, the provided one does not conduct any communication. It fetches the data from the trace file appropriately.

First of all, the user has to choose a process to replay the execution from its perspective. This can be done by using the trace file created by this process in the capture phase as a data provider during the replay phase. After the application starts, the developed MPI implementation ignores all the MPI function calls made outside the CUT. It also ignores any call made by non relevant process in the CUT. Once it receives a call made by the replayer process in the CUT, it uses the data provider specified at the beginning to respond to this call.

Figure 3.6 depicts the replay phase from the perspective of `Process x`. It shows the trace file that was written by this process in the capture phase. This file is the source of the input data during the replay phase. It also shows that only one process is needed to replay the execution.

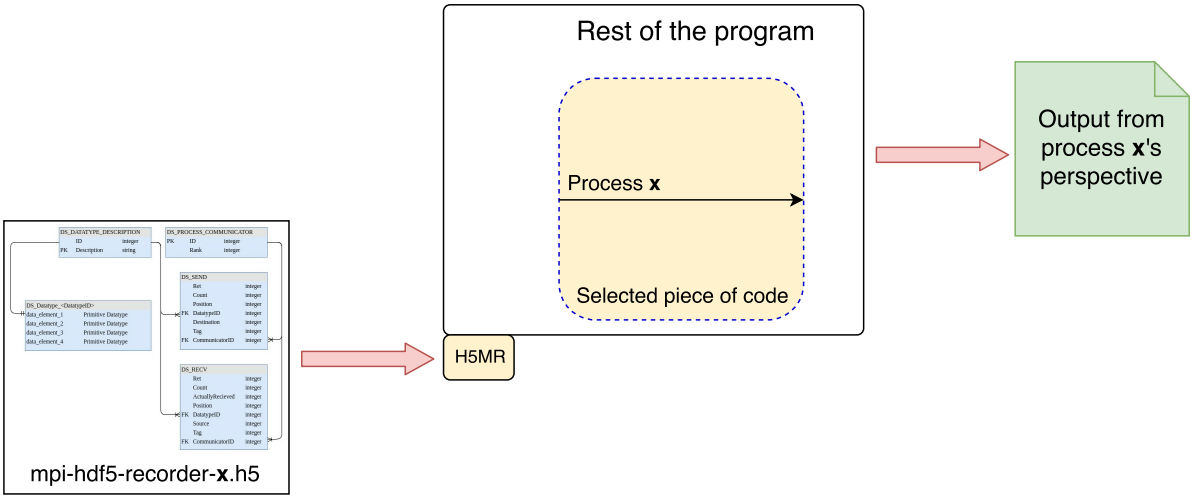


Figure 3.6: Replay phase from the perspective of `Process x`.

Reading the Trace Information

Handling the data maintained in the trace file appropriately is an essential step to replay the execution of the captured operations correctly. For each operation, there are data and metadata. When the provided MPI library receives an MPI call to perform a certain operation, it indicates the dataset that contains the metadata depending on the operation type. Subsequently, it searches for the tuple (source/destination, tag, communicator) in this dataset to get the related metadata.

According to this metadata, the transmitted data is fetched. First of all, the datatype of this data is decoded. This step is analogous to that in the capture phase where a string, which describes the datatype layout, is generated. This description is searched for in the dataset `DS_DATATYPE_DESCRIPTION` to get the `ID` assigned to it. This `ID` determines the dataset that contains the data transmitted by the operation. For instance, given that the transmitted data has the datatype `datatype_xy` which has the description associated with the `ID = 123` in the dataset `DS_DATATYPE_DESCRIPTION`. Then, the dataset that contains the transmitted data is `DS_DATATYPE_123`.

As mentioned in Chapter 1, the data that stems from another processes may be required as input data when testing an MPI program. That makes the received data is the data of interest in the replay phase. Hence, the replay phase aims to emulate the receive operations performed by the replayer process. It achieves this work by:

1. Opening the trace file.
2. Finding the metadata that describes the receive operation.
3. Fetching the received data depending on this metadata.

3.4.4 Recording User Data

The capture data model presented in this section is dedicated to capture data transmitted by processes, i.e., it does not allow the user to record other data like arguments of a subroutine that should be tested. Recording data on demand may be beneficial in other use cases. For instance, recording initial configurations of a program during capturing and retrieving them when replaying ascertains replaying the execution by using the same configurations. Moreover, recording the output of a captured execution allows to compare it with the output of a corresponding replayed execution. This point is essential when creating test units for MPI programs.

To satisfy this requirement, the data model depicted in Figure 3.4 is extended. Figure 3.7 shows it after extension.

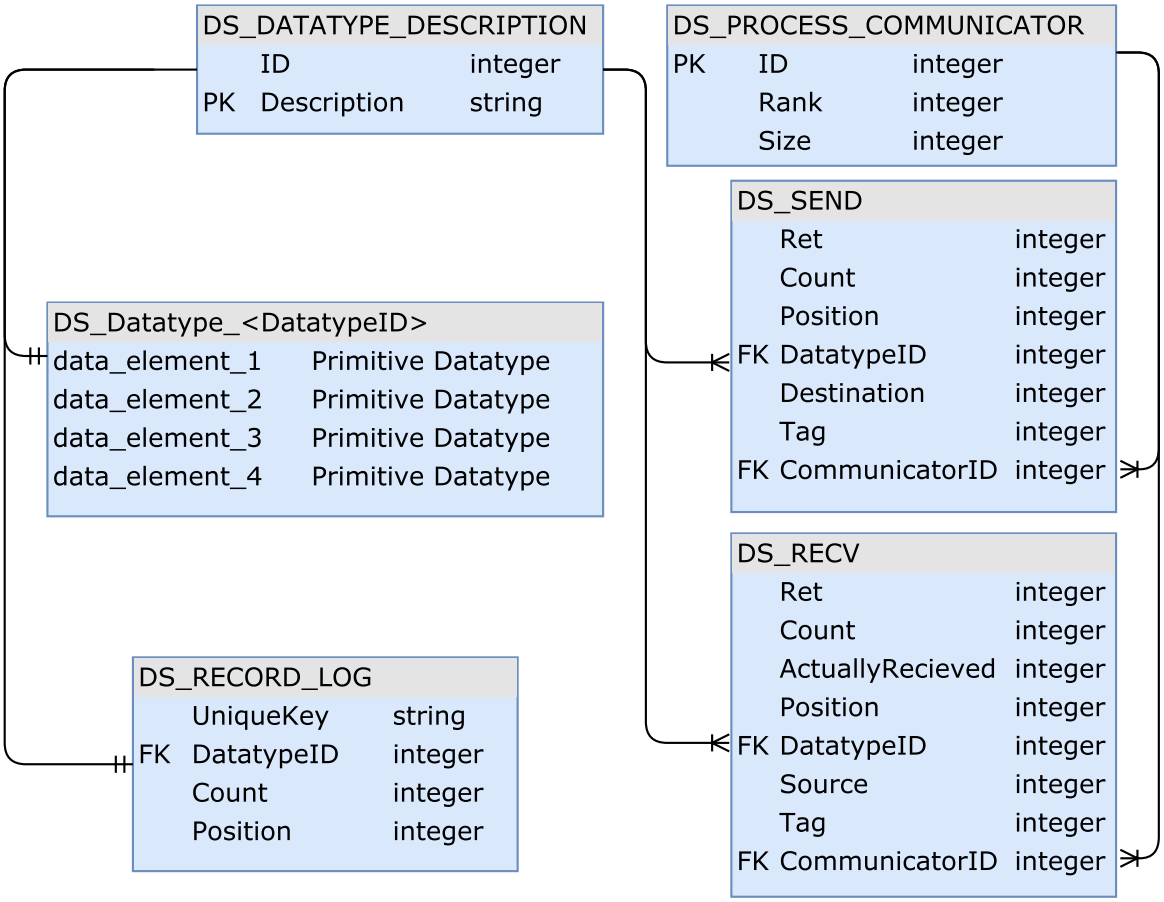


Figure 3.7: The extended capture data model.

The newly created dataset `DS_RECORD_LOG` is dedicated to store metadata about any data saved by the user. The string `UniqueKey` acts as an identifier for this data. Similar to the data transmitted between processes, the data recorded by the user at his/her own request is saved according to its type in a dedicated dataset as explained in Section 3.4.

3.5 Requirements Assessment

This section discusses briefly how the proposed solution fulfills the requirements described in Section 3.2. Defining a technique for capturing and replaying the executions of MPI programs that accounts for usability, efficiency, and platform independency issues involves a set of challenges. The presented technique allows overcoming these issues by providing a flexible and efficient way to capture and replay executions.

When usability is concerned, the use of the technique in the capture phase needs only to link a library to the application and select CUT. Likewise, the use of it in the replay phase requires linking the provided MPI implementation to the application in addition to the data recorded in the capture phase. Above that, this technique allows replaying

the execution by one process which reduces the operation costs. Moreover, different test scenarios can be derived as the captured data is available after the capture phase.

When efficiency is concerned, the technique allows for limiting the volume of the recorded data by suitably selecting the subset of the code on which the test can be conducted.

When platform independency is concerned, the technique uses a standard file format which makes it operable on any platform. It gives the users the ability to capture the execution of a program on any platform and replay its execution on a different one.

3.6 Alternative Solution: Source Code Generation

To facilitate the testing procedure of a subroutine that calls MPI functions, a test driver for it can be generated. For each data communication performed by an MPI call in this subroutine, a call of a corresponding function is generated in the test driver. Each generated call holds the communicated data as parameters, instead of storing it in a file like the previous solution. In this way, the user is able to derive different test scenarios by manipulating this data easily.

For instance, if the subroutine under test contains a call of `MPI_Recv()`, the generated test driver will contain a call of a function `mpi_add_receive()`. This function is able to programmatically register the message that is received via `MPI_Recv()`. In addition, it registers metadata that describes this operation. This subroutine is implemented like the following:

```
1  void mpi_add_receive(void *buf, int count, MPI_Datatype datatype, int source,
       ↪ int tag, MPI_Comm comm)
2  {
3      mpi_msg_t * msg = malloc(sizeof(mpi_msg_t));
4
5      // The amount of space is needed for packed data
6      int size;
7      int position = 0;
8      MPI_Pack_size(count, datatype, comm, & size);
9      msg->buf = malloc(size);
10
11     MPI_Pack(buf, count, datatype, msg->buf, size, & position, comm);
12     msg->buf_size   = position;
13     msg->count      = count;
14     msg->datatype   = datatype;
15     msg->source     = source;
16     msg->tag        = tag;
17     msg->comm       = comm; // evtl. MPI COMM DUP
18     g_array_append_vals(mpi.recvs, msg, 1);
19
20     mpi.total_pending_msgs++;
21 }
```

After generating a call of this function for each call of `MPI_Recv()`, a call of the subroutine under test is generated. From an MPI application, a simple user code example for a subroutine that calls `MPI_Recv()` could be like the following:

```
int testfunc()
{
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, & rank);
    assert(rank == 4711);

    MPI_Status status;
    int data[2];
    int ret;
    ret = MPI_Recv(data, 2, MPI_INT, 4700, 2244, MPI_COMM_WORLD, & status);
    assert(ret == MPI_SUCCESS);
    assert(data[0] == 2233);
    int count = 0;
    ret = MPI_Get_count(& status, MPI_INT, & count);
    assert(count == 1);

    float fdata[2];
    ret = MPI_Recv(fdata, 2, MPI_FLOAT, 4242, 2121, MPI_COMM_WORLD, & status);
    assert(ret == MPI_SUCCESS);
    assert(fdata[0] == 1.0f);
    count = 0;
    ret = MPI_Get_count(& status, MPI_FLOAT, & count);
    assert(count == 2);

    return 0;
}
```

The generated test driver for this subroutine could be as follows:

```
int main(int argc, char** argv)
{
    MPI_Init(& argc, & argv);
    // Method setup
    mpi_set_rank(MPI_COMM_WORLD, 4711);
    int data = 2233;
    float fdata[] = {1.0, 2.0};
    mpi_add_receive(fdata, 2, MPI_FLOAT, 4242, 2121, MPI_COMM_WORLD);
    mpi_add_receive(& data, 1, MPI_INT, 4700, 2244, MPI_COMM_WORLD);

    testfunc(); // Subroutine under test

    assert(mpi_dummy_check_pending_messages() == 0);

    MPI_Finalize();
    return 0;
}
```

Chapter Summary

In this chapter, some design decisions and rationales were unfolded to guide the implementation of the Capture & Replay technique. The defined technique is specifically designed to be used on legacy software as well as on software under development. It has three main characteristics:

First, the technique captures and replays executions selectively. Users can specify any CUT, and the technique captures merely the operations performed by the processes that run through this CUT during the execution. In the replay phase, just the execution of the selected piece of code will be replayed.

Second, the technique captures and replays executions in terms of interactions between processes, i.e., MPI operations like send and receive. During capture, the technique records every interaction performed by any relevant process as an operation with a set of attributes. These attributes include metadata about this operation in addition to the data transmitted by it. During replay, the technique uses the recorded attributes of each captured interaction to replay the execution.

Third, when capturing interactions, the technique records the data transmitted between the processes that execute the CUT. In addition, it records the data that traverses the boundary between this CUT and the rest of the application. That allows the intended test to deal with code that requires input that stems from any process. And hence, it is possible to run such a test with one process as the important input is available.

Highlights of implementation aspects are discussed in the next chapter.

4 Implementation

This chapter covers the technical aspects and the implementation of the targeted solution. Some specifics that were deemed interesting are discussed in detail. Section 4.1 justifies the choice of the programming language. Selecting a piece of code to test is introduced in Section 4.2. Section 4.3 discusses the decoding of MPI datatypes. Section 4.4 reveals the procedure used to identify any communicator in which a captured interaction is performed. The tracing library used to intercept the MPI calls in the capture phase is presented in Section 4.5, whereas the MPI library used to respond to these calls during replay is introduced in Section 4.6. The algorithm used to capture non-blocking point-to-point communications is presented in Section 4.7. Enabling the user to record data during capture on demand in order to retrieve it during replay is outlined in Section 4.8.

4.1 Programming Language

Choosing the programming language is an early decisions made in any software project. Since C and Fortran are languages from which MPI can be interfaced directly, i.e., without additional bindings, and C is a long-term popular choice for software developers, C is a good candidate. An additional argument in favor of C is that a lot of scientists develop their software in C and MPI. They are, as mentioned in the introduction to this thesis, among those who have a need for tools supporting them in testing their software, therefore the code will be easier to understand for them. In addition, the data within C programs can be managed efficiently and elegantly by using *GLib collections* [21]. The structures provided by these collections are necessary for caching purposes. These are the main reasons for which C was chosen.

4.2 Selecting CUT

Selecting a piece of code to test (CUT) is the first aspect in facilitating the testing of MPI applications. It allows the user to focus on one specific section of the code execution during run-time. This step is the first action taken by the user. For the purpose of configuring the capture scope, the function `h5mr_init()` has to be called. Passing `1` as an argument to this function leads to capture the execution of the whole program, whereas passing `0` gives the user the ability to choose the CUT manually by using the functions `h5mr_start_recording()` and `h5mr_stop_recording()`. Calling the first one declares the beginning of the CUT, whereas calling the second one announces its ending. Selecting many CUTs is also allowed.

Listing 4.1 shows a code example that includes calls to initialize and shutdown MPI (`MPI_Init()`, `MPI_Finalize()`) and to identify the process number (`MPI_Comm_rank()`). When the compiled program is executed with two processes, certain branches are executed by both processes and some only by `Rank 0` or `Rank 1`. Then, the processes exchange information using a single point-to-point communication (`MPI_Send()`, `MPI_Recv()`) where `Rank 0` sends data of user-defined datatype to the other process. Since the function `h5mr_init()` is called with `0` as an argument, the CUT is determined by calling the functions `h5mr_start_recording()` and `h5mr_stop_recording()`. As can be seen, the selected CUT is a piece of a branch executed by `Rank 1` where these functions are called.

```
1  #include <stdio.h>
2  #include <mpi.h>
3  #include "h5mr.h"  // Linking the Library
4
5  int main(int argc, char* argv[]) {
6      h5mr_init(0);  // Capture the execution in manual mode
7      int rank, i;
8      MPI_Status status;
9      MPI_Init(&argc,&argv);
10     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
11
12     const int count              = 2;
13     const int blocklength        = 3;
14     const int stride             = 4;
15
16     MPI_Datatype c_datatype, v_datatype;
17     MPI_Type_vector(count, blocklength, stride, MPI_INT, &v_datatype);
18     MPI_Type_commit(&v_datatype);
19
20     if(rank == 0){
21         int buffer[21];
22         for (i=0; i<21; i++)
23             send_buffer[i] = get_grade(i);
24
25         MPI_Send(send_buffer, 3, v_datatype, 1, 53, MPI_COMM_WORLD);
26
27     } else if(rank == 1) {
28         int receive_buffer[21];
29         for (i=0; i<21; i++)
30             receive_buffer[i] = 0;
31
32         h5mr_start_recording(); // Start capturing
33         MPI_Recv(receive_buffer, 3, v_datatype, 0, 53, MPI_COMM_WORLD,
               ↪ &status);
34         // Compute GPA
35         int gpa = Compute_gpa(receive_buffer);
36         h5mr_stop_recording(); // Stop capturing
37
38         printf("GPA: %d", gpa);
39     }
40
41     MPI_Type_free(&v_datatype);
42
43     MPI_Finalize();
44     return 0;
45 }
```

Listing 4.1: Selecting a piece of code for testing.

According to the discussion in Section 3.4, the output of this execution is the original output in addition to a trace file created by the `Rank 1` which is named `mpi-hdf5-recorder_1.h5`.

4.3 Decoding MPI Datatypes

As mentioned in Chapter 3, the proposed solution works by writing the data transmitted between the processes during the capture phase and fetching it in the replay phase. Consequently, detecting the type of this data in each phase is fundamental for the implementation.

In fact, MPI predefines its primitive datatypes. In addition, it provides datatype objects, which allows users to derive any datatype and specify any layout of data in memory. The layout information, once put in a datatype, could not be decoded from this datatype. Above that, these derived datatypes may be constructed from existing datatypes, which may be, in turn, primitive or even derived. This makes the decoding procedure very hard and therefore, the most effort has been made to decode these opaque datatypes. However, decoding each used datatype, primitive or derived, is useful in two cases:

1. To create a description for this datatype which will used to identify this datatype.
2. To create a datatype-specific dataset which will maintain data of this datatype.

4.3.1 Creating a Datatype Description

As mentioned in Section 3.3, a description for each datatype of the data transmitted among the processes acts as a unique identifier for this datatype. Figure 4.1 shows the activity diagram of the algorithm used to create a description for any given MPI datatype.

At the beginning, the description is initialized with an empty string. After that, the function `decode_datatype()` is used to decode the datatype in two steps. In the first step, the MPI function `MPI_Type_get_envelope()` is called to detect if the datatype is a primitive one. If it is a primitive datatype, the function `decode_primitive_datatype()` reveals it and append the appropriate description to the description string initialized at the beginning. If the datatype is derived, an additional step is needed. In this step, the MPI function `MPI_Type_get_contents()` is called to expose more information about this datatype. The crucial information is the number of the sub-datatypes that this datatype consists of. Subsequently, the function `decode_datatype()` is called *recursively* for each revealed sub-datatype. At the end, the description lists all the primitive sub-datatypes and illustrates their layout in memory.

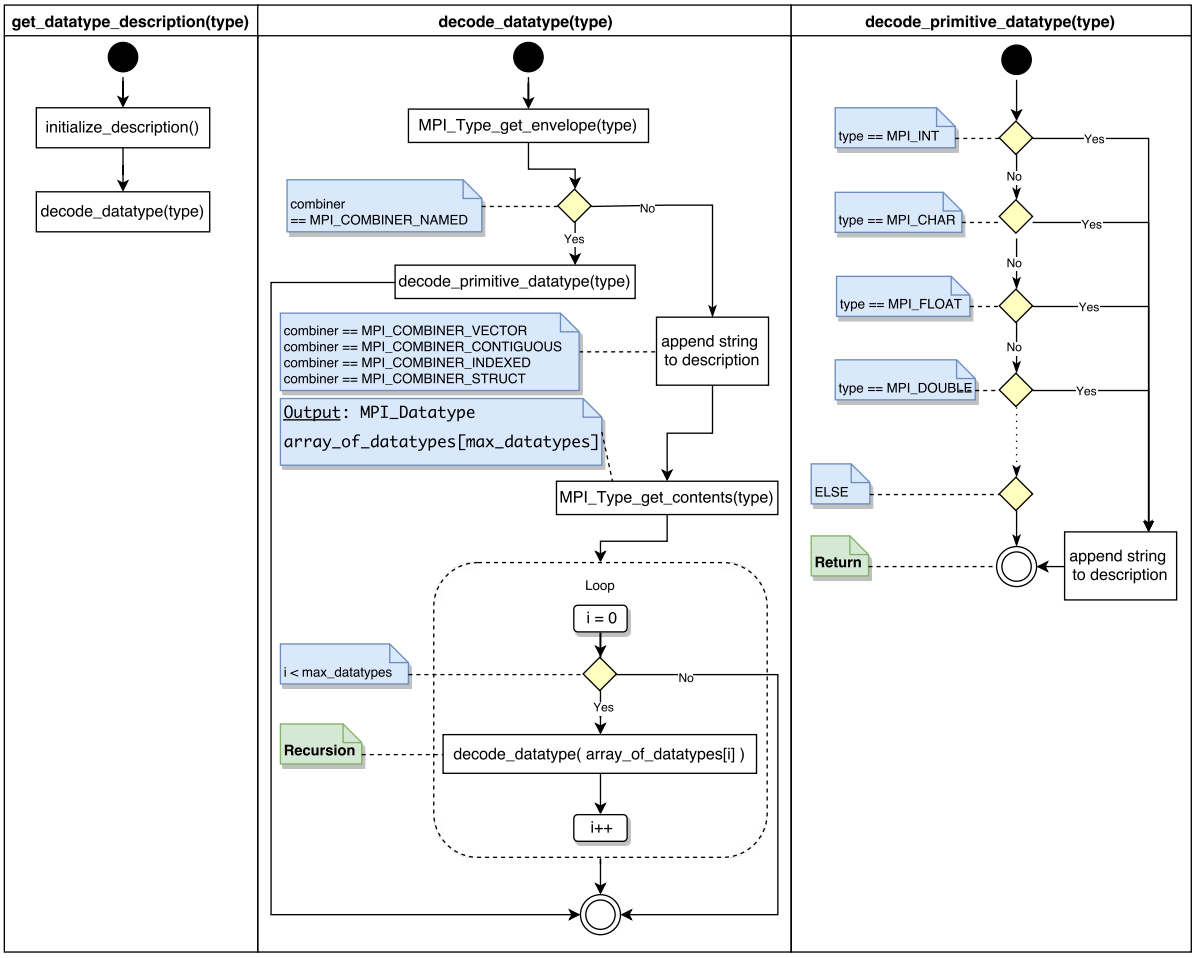


Figure 4.1: Creating a Datatype Description: Activity Diagram

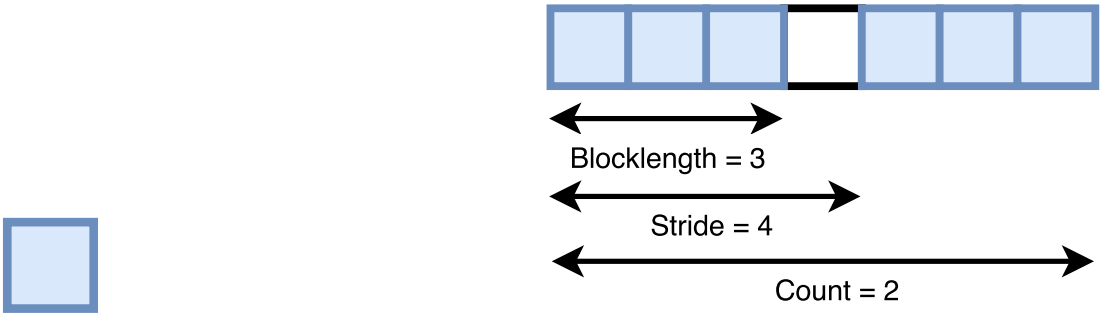
Decoding Example

In Listing 4.2, two user-defined MPI datatypes are created.

```
1     const int count          = 2;
2     const int blocklength    = 3;
3     const int stride         = 4;
4
5     MPI_Datatype c_datatype, v_datatype;
6     MPI_Type_vector(count, blocklength, stride, MPI_INT, &v_datatype);
7     MPI_Type_commit(&v_datatype);
8     MPI_Type_contiguous(3, v_datatype, &c_datatype);
9     MPI_Type_commit(&c_datatype);
```

Listing 4.2: Decoding Example.

The first datatype is a vector which consists of two blocks. Each one concatenates three items from the primitive type `MPI_INT`. The spacing between these two blocks is four multiples of the extent of the type `MPI_INT`. This user-defined datatype has been given the name `v_datatype`. It is shown in Figure 4.2b.



(a) `MPI_INT` (b) Vector Datatype: `v_datatype`

Figure 4.2: Primitive and derived MPI datatypes.

When decoding this datatype by using the algorithm stated above, it returns the following description:

```
VECTOR(count=2,blocklength=3,stride=4,typ=NAMED(INT),size=24,extent=28)
```

The second derived datatype is a contiguous one which concatenates three copies of the datatype `v_datatype` defined above. This user-defined datatype has been given the name `c_datatype`. It is seen in Figure 4.3.

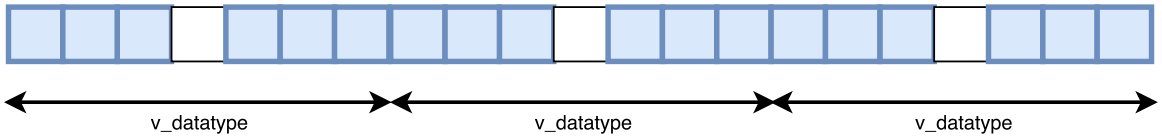


Figure 4.3: Contiguous Datatype: `c_datatype`.

When decoding this datatype, the algorithm returns the following description:

```
CONTIGUOUS(count=3,typ=VECTOR(count=2,blocklength=3,stride=4,
typ=NAMED(INT),size=24,extent=28),size=72,extent=84)
```

When comparing both descriptions, the first one appears completely in the second one. This proves the recursiveness used in the implemented algorithm.

4.3.2 Creating a Datatype Dataset

As illustrated in Section 3.3, for each datatype, a dataset is created. This dataset maintains the data of this datatype. Therefore, decoding each datatype is essential for creating the appropriate storage layouts (in memory and on disk) of the data elements from this datatype. The created storage layout is used to save these elements in the dedicated dataset appropriately.

By using an algorithm similar to that used to create a description for any given MPI datatype, these storage layouts are created, in the afore-mentioned locations. These layouts are eventually used to create the datasets. The activity diagram shown in Figure 4.4 visualizes the steps through this algorithm.

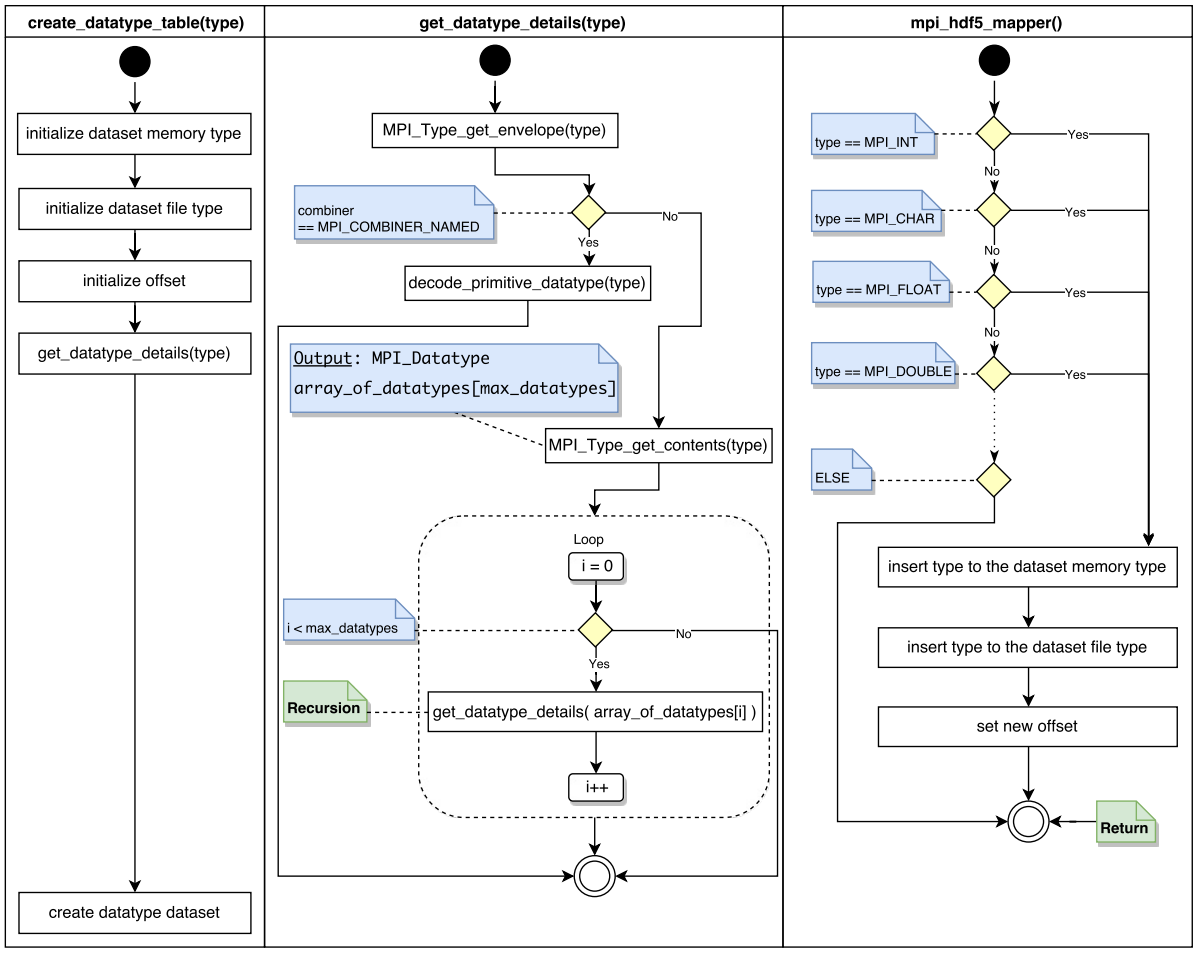


Figure 4.4: Creating a Datatype Dataset: Activity Diagram

Initially, both layouts are initialized. Next, the function `get_datatype_details()` is used to decode the datatype in two steps. In the first step, it detects if the datatype is a primitive by using the function `MPI_Type_get_envelope()`. If it is primitive, the function `decode_primitive_datatype()` reveals its actual type and appends the appropriate member to each layout created at the beginning. If the datatype is derived, the MPI function `MPI_Type_get_contents()` is called to discover more information about it. The crucial information is the number of the sub-datatypes that consist this derived datatype. Subsequently, the function `get_datatype_details()` is called *recursively* for each revealed sub-datatype. At the end, the storage layouts is completed and adequate to be used for creating the targeted dataset.

4.4 Communicators Identification

According to the design principles discussed in Chapter 3, each communicator is represented by a unique identifier (`CommID`) in the data model. This identifier is used to refer to this communicator when capturing any communication performed in it. To identify each communicator in the data model, a unique name is given to each one by the provided MPI library. For this purpose, the function `MPI_Comm_set_name()` is used. The given name is returned, when needed, by using the function `MPI_Comm_get_name()`. For each name, a unique integer is assigned. This integer is the attribute `CommID` seen in the data model.

If the function `MPI_Comm_set_name()` is called by the user to set a name for any communicator, this name is not assigned to the communicator directly but rather stored in a temporal hash table. When the user calls the function `MPI_Comm_get_name()`, it gets the name from this hash table.

In other words, during the capture phase, each communicator has two names:

1. An actual name which is given and used by the provided MPI library.
2. A nick name which may be given by the user.

This assignment/remapping is not limited to the CUT but applied to the whole program.

4.5 MPI Calls Interception

As discussed in Chapter 3, the capture phase aims to record the interactions between the processes in the CUT. Each process records all the information related to any operation in which it participates. This involves creating the different data structures required to maintain this information. According to the discussion in Section 3.3, this information includes:

1. The metadata that describes any performed MPI operation.
2. The data transmitted by this operation.

To achieve this goal, an MPI Tracing Library, which outlined in Figure 4.5, is developed. This library can be linked against any MPI program in order to instrument it and record trace information for each MPI function called in it. The library intercepts these calls by using the *Profiled MPI (PMPI) interface* and obtains the trace information it needs.

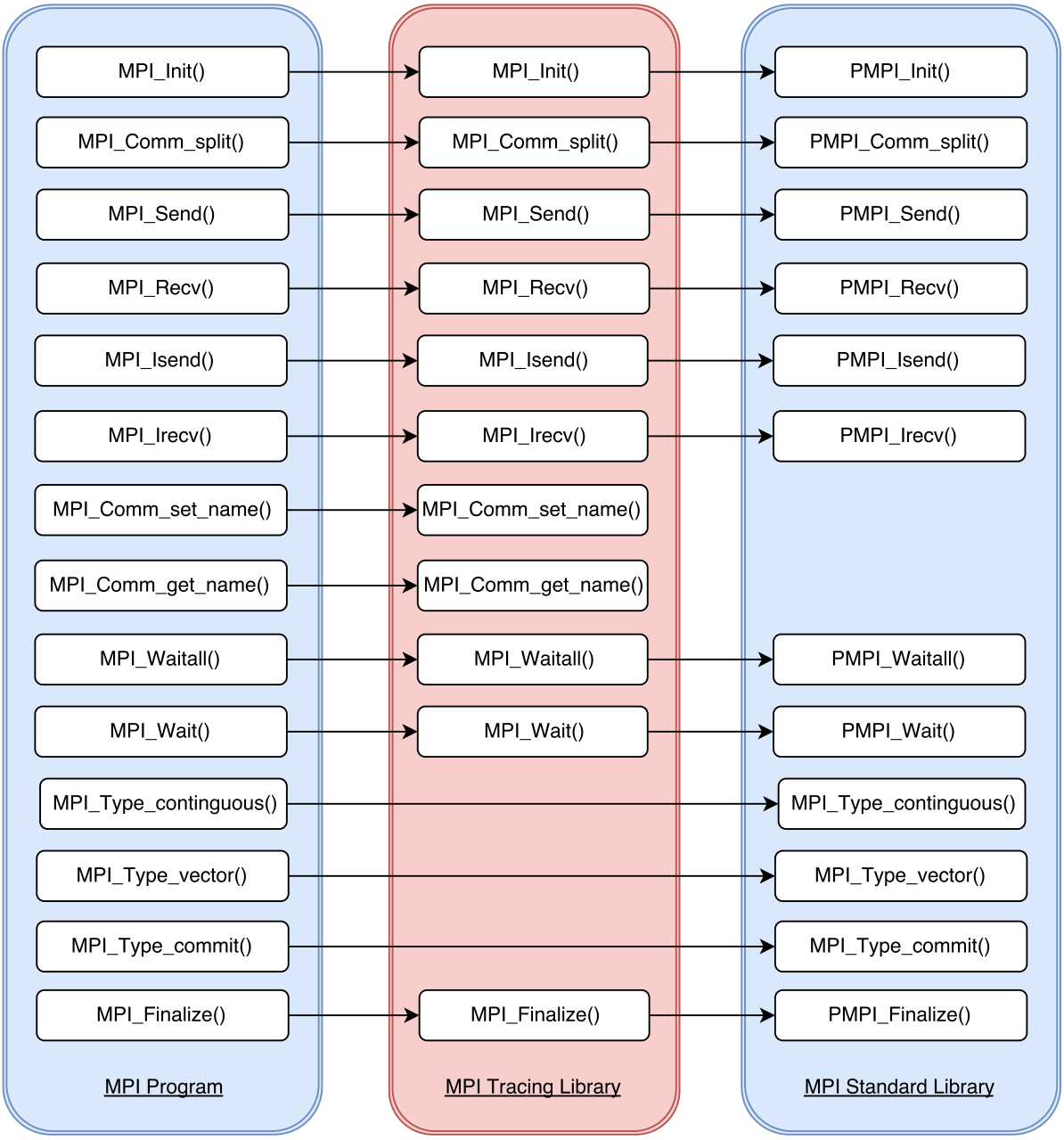


Figure 4.5: MPI Tracing Library

In the following, the functions that form the building blocks of this library and which are used to intercept the MPI calls are introduced. These functions behave like the standard MPI functions, but also exhibit the behavior necessary for the capturing purposes.

MPI_Init()

In this function the MPI execution environment is initialized. After that, each process initializes its own temporary *GLib Hash Tables*. These hash tables are used to cache

information during the capture phase. For instance, if a process participated in an operation on data of any datatype, it has to perform certain tasks to get information about this datatype. The process writes the acquired information in a permanent data structure. Since the datatype may be used again and again, it is efficient to cache its related information in a hash table to use it once more if needed without demand to access it within the permanent data structure. Listing 4.3 shows the function `MPI_Init()`.

```
1 int MPI_Init(int * argc, char *** args)
2 {
3     PMPI_Init(argc, args);
4     h5mr_intitialize_dummy_data_structures();
5
6     return 0;
7 }
```

Listing 4.3: `MPI_Init()`

MPI_Comm_split()

As discussed in Chapter 3, in the permanent data structures, each communicator is represented by a unique identifier `CommID`. Calling the function `MPI_Comm_split()` to create new communicators imposes assigning a new `CommID` for each newly created one.

Figure 4.6 depicts this concept. In this figure, eight processes participate in the default communicator `MPI_COMM_WORLD` wherein each one is given a rank. Further, each one is participated in its own default communicator `MPI_COMM_SELF`. When a new communicator `new_comm_1` is derived from `MPI_COMM_WORLD`, it is given the identifier `3` and each process participated in it is given a new rank. The same procedure applies if another new communicator `new_comm_2` is created.

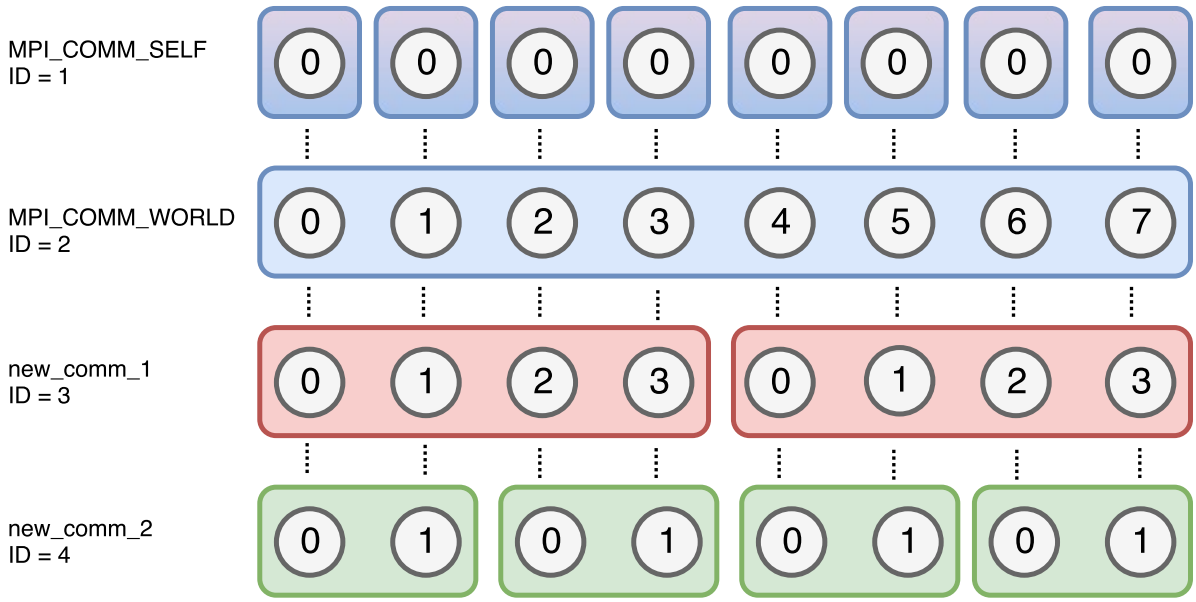


Figure 4.6: New ID for each rank in each communicator.

If the process participates in any interaction within a communicator, the information related to this communicator is saved in the dataset `DS_COMMUNICATOR`. Listing 4.4 shows the code executed when splitting a communicator.

```
1  int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
2  {
3      int ret = PMPI_Comm_split(comm, color, key, newcomm);
4
5      if (ret == MPI_SUCCESS){
6          h5mr_register_communicator( *newcomm );
7      }
8
9      return ret;
10 }
```

Listing 4.4: `MPI_Comm_split()`

As a result, when the capturing is finished, information about each relevant communicator in which the process interacted is maintained in the dataset `DS_COMMUNICATOR`. This information includes the following:

1. The `ID` assigned to the communicator.
2. The new rank given to the process in this communicator.
3. The size of the communicator.

Table 4.1 shows an instance of this dataset where the process interacted with processes in four communicators. The communicators with the `IDs` `1` and `2` are the predefined communicators `MPI_COMM_SELF` and `MPI_COMM_WORLD` respectively. The communicators with the `IDs` `3` and `4` are two user-defined communicators. In the first one, the process get the `Rank` number `2`, whereas in the second one, it was given the `Rank` number `0`.

ID	Rank	Size
1	0	1
2	6	8
3	2	4
4	0	2

Table 4.1: An instance of the `DS_COMMUNICATOR` dataset.

MPI_Comm_set_name()

As discussed in Section 4.4, each communicator may have two names: an actual name and a dummy name. This function is dedicated to set the dummy name provided by the user. This dummy name is saved in a dedicated hash table and is not assigned to the communicator itself. Listing 4.5 shows the code devoted to achieve this goal.

```
1 int MPI_Comm_set_name (MPI_Comm comm, const char *comm_name)
2 {
3     int ret = h5mr_comm_change_name(comm, comm_name);
4     assert (ret == 0);
5
6     return ret;
7 }
```

Listing 4.5: `MPI_Comm_set_name()`

MPI_Comm_get_name()

This function is dedicated to return the dummy name given by the user. Listing 4.6 shows the code that accomplishes this task.

```
1 int MPI_Comm_get_name (MPI_Comm comm, char *comm_name, int *resultlen)
2 {
3     int ret = h5mr_comm_get_dummy_name(comm, comm_name, resultlen);
4     assert (ret == 0);
5
6     return ret;
7 }
```

Listing 4.6: `MPI_Comm_get_name()`

MPI_Send()

When performing a send operation, metadata that describes this operation is saved. If the operation was performed successfully, a copy of the sent data is also saved. Listing 4.7 shows the code devoted to carry out this task.

```
1  int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int
       ↪ tag, MPI_Comm comm)
2  {
3      int ret = PMPI_Send(buf, count, datatype, dest, tag, comm);
4
5      if ( ! h5mr_is_enabled() ){
6          return ret;
7      }
8
9      int position = -1;
10     if (ret == MPI_SUCCESS){
11         position = h5mr_append_buffer_to_datatype_ds( datatype, buf, count );
12         assert ( position >= 0 );
13     }
14
15     int success = h5mr_write_send_log(0, ret, count, position, datatype,
           ↪ dest, tag, comm);
16     assert (success == 0);
17
18     return ret;
19 }
```

Listing 4.7: `MPI_Send()`

Saving the sent data

As mentioned above, if the send operation performed successfully, a copy of the sent data contained in the buffer `buf` is saved in the dataset `DS_DATATYPE_<ID>`. This `ID` refers to the attribute `ID` in the dataset `DS_DATATYPE_DESCRIPTION`. Table 4.2 shows an instance of this dataset.

ID	Description
1	`CONTIGUOUS(count=3,typ=VECTOR(count=2,blocklength=3,stride=4, typ=NAMED(INT),size=24,extent=28),size=72,extent=84)`
2	`VECTOR(count=2,blocklength=3,stride=4, typ=NAMED(INT),size=24,extent=28)`
3	`STRUCT(count=2,blocklength=[1;1],displacement=[0;8], typ=[NAMED(INT);NAMED(DOUBLE)],size=12,extent=16)`
4	`STRUCT(count=2,blocklength=[1;1],displacement=[0;28], typ=[VECTOR(count=2,blocklength=3,stride=4, typ=NAMED(INT),size=24,extent=28);NAMED(DOUBLE)] ,size=32,extent=36)`

Table 4.2: An instance of the `DS_DATATYPE_DESCRIPTION` dataset.

For instance, when sending data of the datatype that has the description associated with the `ID = 2` seen in Table 4.2, this data is saved in the dataset `DS_DATATYPE_2`. Table 4.3 shows an instance of this dataset.

INT_1	INT_2	INT_3	INT_4	INT_5	INT_6
3	2	3	4	5	7
36	22	31	2	15	8
1	9	2	2	4	10

Table 4.3: An instance of the `DS_DATATYPE_2` dataset.

For each row in this dataset, a unique number starting with `0` is assigned. Moreover, each row maintains one data element. Namely, when capturing a send operation of three data elements, a copy of these elements is stored in three rows within this dataset. The number of the first row represents the attribute `Position` that is used as metadata to describe the captured operation. This attribute in addition to the count of the data elements transmitted per operation are used to fetch this data in the replay phase correctly.

Saving the operation metadata

The operation metadata includes the arguments of the function `MPI_Send()`. Additionally, it includes the attribute `Ret` that indicates whether the send operation was performed successfully. Moreover, it contains the attribute `Position`. As stated above, this attribute

represents the number of the row in the dataset where the data transmitted previously is stored. If the send operation failed, the attribute `Position` will contain the value `-1`. This metadata is saved in the dataset `DS_SEND`. Table 4.4 shows an instance of this dataset.

Ret	Count	Position	DatatypeID	Destination	Tag	CommID
0	3	0	1	1	1147	3
0	2	0	2	1	1148	3
0	2	3	1	1	1150	4
0	1	2	2	1	1152	4

Table 4.4: An instance of the `DS_SEND` dataset.

In this dataset, the argument `datatype` is represented in the field `DatatypeID` by a foreign key that refers to the attribute `ID` in the dataset `DS_DATATYPE_DESCRIPTION` (see Table 4.2). The same thing applied to the argument `comm` which is also represented in the column `CommID` by a foreign key. This key refers to the attribute `ID` in the dataset `DS_COMMUNICATOR` introduced previously.

Example The second record in the dataset `DS_SEND` indicates the following:

1. A sending operation is performed successfully.
2. In this operation, two data elements are transmitted.
3. The data transmitted in this operation is pf the type that has the description associated with the `ID = 2` in the dataset `DS_DATATYPE_DESCRIPTION`.
4. A copy of this data is saved in the dataset `DS_DATATYPE_2` starting with the row number `0`.

MPI_Recv()

Similar to the procedure followed when capturing a send operation, when capturing a receive operation, metadata that describes this operation is stored in the dataset `DS_RECV`. If this operation performed successfully, a copy of the received data is saved in a dedicated dataset. Listing 4.8 shows the code that realizes this task.

```
1 int MPI_Recv(const void *buf, int count, MPI_Datatype datatype, int dest, int
      ↪ tag, MPI_Status *stat_out)
2 {
3     int ret;
4     if ( ! h5mr_is_enabled() ){
5         ret = PMPI_Recv(buf, count, datatype, source, tag, comm, stat_out);
6         return ret;
7     }
8
9     MPI_Status* stat = stat_out;
```

```
10      if (stat_out == MPI_STATUS_IGNORE) {
11          // use a fake status
12          MPI_Status actual_status;
13          stat = &actual_status;
14      }
15
16      ret = PMPI_Recv(buf, count, datatype, source, tag, comm, stat);
17
18      int actually_received;
19      MPI_Get_count(stat, datatype, &actually_received);
20      if (ret != MPI_SUCCESS){
21          actually_received = 0;
22      }
23
24      int position = -1;
25      if (actually_received > 0){
26          position = h5mr_append_buffer_to_datatype_ds( datatype, buf,
                ↪ actually_received );
27          assert ( position >= 0 );
28      }
29
30      int success = h5mr_write_recv_log(ret, count, actually_received,
            ↪ position, datatype, source, tag, comm);
31      assert (success == 0);
32      return ret;
33  }
```

Listing 4.8: `MPI_Recv()`

In fact, if an `MPI_Status` structure was passed to the `MPI_Recv()` function, it will be populated with additional information about the receive operation after it completes. If the user intended to ignore this information in his program, a fake `MPI_Status` is used alternatively. However, this information is important to get the total number of `datatype` elements that were actually received. This number is maintained in the field `ActuallyReceived`. As seen in Table 4.5, the other attributes that describe a captured receive operation are similar to those recorded when capturing a send operation.

Ret	Count	ActuallyReceived	Position	DatatypeID	Source	Tag	CommID
0	3	1	0	1	1	1147	3
0	3	2	1	2	1	1148	3
0	3	3	3	1	1	1150	4
1	3	0	-1	2	1	1152	4

Table 4.5: An instance of the `DS_RECV` dataset.

Saving the received data

If the data received successfully, a copy of this data is saved. Similar to the captured sent data, the received data is saved in a dedicated dataset depending on its type. For instance, when considering the data of the datatype that has the description associated with the `ID = 2` seen in Table 4.2, this data is saved in the dataset `DS_DATATYPE_2`. Table 4.6 shows an instance of this dataset.

INT_1	INT_2	INT_3	INT_4	INT_5	INT_6
3	2	3	4	5	7
1	9	2	2	4	10

Table 4.6: An instance of the DS_DATATYPE_2 dataset.

MPI_Isend()

Since the data intended to send is available when starting non-blocking send operation, capturing this data is similar to capturing the send operation in blocking mode. Listing 4.9 shows the code devoted to achieve this purpose.

```
1  int MPI_Isend (const void *buf, int count, MPI_Datatype datatype, int dest,
       ↪ int tag, MPI_Comm comm, MPI_Request *request)
2  {
3      int ret = PMPI_Isend (buf, count, datatype, dest, tag, comm, request);
4
5      if ( ! h5mr_is_enabled() ){
6          return ret;
7      }
8
9      int position = -1;
10     if (ret == MPI_SUCCESS){
11         position = h5mr_append_buffer_to_datatype_ds (datatype, buf, count);
12         assert ( position >= 0 );
13     }
14
15     int success = h5mr_write_send_log (2, ret, count, position, datatype,
           ↪ dest, tag, comm);
16     assert (success == 0);
17
18     return ret;
19 }
```

Listing 4.9: MPI_Isend()

MPI_Irecv()

In contrast to capturing the blocking receive operations introduced previously, capturing data which is received in non-blocking mode does not occur immediately. When a receive request object is created by calling the function MPI_Irecv(), it is stored temporarily in a hash table. Subsequently, when a call of the function MPI_Wait() with the same receive object request is intercepted, this object is used to determine if the receive operation is completed and to store the related trace information which is saved in the hash table in the permanent data structures. This procedure is presented in detail in Section 4.7 on page 56. Listing 4.10 shows the code executed when intercepting a call of the function MPI_Irecv().

```
1  int MPI_Irecv (void *buf, int count, MPI_Datatype datatype, int source, int
       ↪ tag, MPI_Comm comm, MPI_Request *request)
```

```
2  {
3      int ret = PMPI_Irecv(buf, count, datatype, source, tag, comm, request);
4
5      if ( ! h5mr_is_enabled() ){
6          return ret;
7      }
8
9      int success = h5mr_register_receive_request(buf, ret, count, datatype,
           ↪ source, tag, comm, request);
10     assert (success == 0);
11
12     return ret;
13 }
```

Listing 4.10: `MPI_Irecv()`

MPI_Wait()

The call of this function is intercepted in order to get the information associated with the request object `request`. When this request is completed, it checks if this request is a receive request. In this case, it saves metadata about the receive operation and a copy of the received data if received successfully. Listing 4.11 shows the code dedicated to achieve this goal. More details about this interception is introduced in Section 4.7 on page 56.

```
1  int MPI_Wait (MPI_Request *request, MPI_Status *status)
2  {
3      if ( h5mr_is_enabled() ){
4          int flag = 0;
5          while (flag == 0){
6              MPI_Request_get_status(*request, &flag, status);
7          }
8
9          int is_consistent = h5mr_consist_i_recv (request);
10         assert( is_consistent == 0 );
11     }
12
13     int ret = PMPI_Wait (request, status);
14
15     return ret;
16 }
```

Listing 4.11: `MPI_Wait()`

MPI_Waitall()

The error-free interception of `MPI_WAITALL(count, array_of_requests, array_of_statuses)` has the same effect as the interception of `MPI_Wait(&array_of_request[i], &array_of_statuses[i])`, for `i=0` and `count=1`. `MPI_Waitall()` with an array of length one is equivalent to `MPI_Wait()`. Listing 4.12 shows the code devoted to make this equivalence.

```
1  int MPI_Waitall (int count, MPI_Request* array_of_requests, MPI_Status*
       ↪ array_of_statuses)
2  {
3      if ( h5mr_is_enabled() ){
4          int flag;
5          for(int i=0; i < count; i++){
6              flag = 0;
7              while (flag == 0){
8                  MPI_Request_get_status(array_of_requests[i], &flag,
                       ↪ &array_of_statuses[i]);
9              }
10             int is_consistent = h5mr_consist_i_recv (&array_of_requests[i]);
11             assert( is_consistent == 0 );
12         }
13     }
14
15     int ret = PMPI_Waitall(count, array_of_requests, array_of_statuses);
16
17     return ret;
18 }
```

Listing 4.12: `MPI_Waitall()`

MPI_Finalize()

Listing 4.13 shows the function `MPI_Finalize()` where the MPI environment is cleaned up. But before that, each process releases the resources it used to create the permanent data structures needed in the capture phase. The file, wherein these structures created, is also closed. In addition, temporary data structures like GLib Hash Tables are destroyed.

```
1  int MPI_Finalize()
2  {
3      h5mr_free_dummy_data_structures();
4
5      if ( h5mr_is_enabled() ){
6          h5mr_free_resources();
7      }
8
9      PMPI_Finalize();
10
11     return 0;
12 }
```

Listing 4.13: `MPI_Finalize()`

4.6 Dummy MPI Library

As mentioned in Chapter 3, replaying the execution from the perspective of any specific process that participated in the original execution is the goal of the replay phase. This implies emulating the interactions in which this process participated. To achieve this goal, a dummy MPI library is provided. Therefore, when creating a test driver for the CUT, this test driver is linked against this library instead of a conventional MPI

implementation. This library uses one of the trace files produced in the capture phase as a data provider during the replay phase. When an MPI function call is issued, the data stems from the trace file instead of the run-time environment. In the following, the functions that this library consists of are introduced.

MPI_Init()

This function assumes that a trace file is passed to the program as a last argument. After that, it parses the file name to detect the replayer process. If no argument is passed, the execution is aborted. Listing 4.14 shows this function.

```
1  int MPI_Init(int * argc, char *** argv)
2  {
3      // the last argument of the program is expected to be the trace file
4      if (*argc > 1){
5          (*argc)--;
6          h5mr_initialize_param((*argv)[*argc]);
7
8          printf("H5MR Replaying the trace file: %s for the process: %d\n",
               ↪ tracefile, world_rank);
9          h5mr_initialize_hashtables();
10     } else {
11         printf("H5MR: please use the trace file as last parameter\n");
12         MPI_Abort(MPI_COMM_WORLD, 2);
13     }
14     PMPI_Init(argc, argv);
15
16     return MPI_SUCCESS;
17 }
```

Listing 4.14: `MPI_Init()`

MPI_Comm_rank()

Similar to the original implementation of the function `MPI_Comm_rank()`, the dummy implementation determines the rank of the replayer process in the given communicator `comm`. Listing 4.15 shows this function.

```
1  int MPI_Comm_rank(MPI_Comm comm, int *rank)
2  {
3      // find rank based on the communicator
4      int ret = h5mr_find_rank_in_comm( comm, rank );
5      assert(ret == 0);
6
7      return MPI_SUCCESS;
8  }
```

Listing 4.15: `MPI_Comm_rank()`

MPI_Recv()

Providing the data that stems from other processes is the goal of this function. It depends on the datatype of the received data to detect the dataset where the received data is stored. Listing 4.16 shows this function.

```
1  int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int
       ↪ tag, MPI_Comm comm, MPI_Status *stat_out)
2  {
3      int count = h5mr_fetch_transmitted_data( buf, datatype );
4      assert(count > 0);
5
6      if (stat_out != MPI_STATUS_IGNORE){
7          h5mr_populate_status(source, tag, count, stat_out);
8      }
9
10     return MPI_SUCCESS;
11 }
```

Listing 4.16: `MPI_Recv()`

MPI_Finalize()

This function destroys the resources that were used to replay the execution are destroyed. Listing 4.17 shows this function.

```
1  int MPI_Finalize()
2  {
3      h5mr_free_resources();
4      PMPI_Finalize();
5
6      return MPI_SUCCESS;
7  }
```

Listing 4.17: `MPI_Finalize()`

4.7 Capturing Non-blocking Communications

In a parallel program, blocking point-to-point communication is done by using `MPI_Send()` and `MPI_Recv()`, whereas `MPI_Isend()` and `MPI_Irecv()` are used to communicate in non-blocking mode.

In blocking mode, the sending process returns from the `MPI_Send()` call only when the data from the send buffer can be safely overwritten again, whereas the receiving process returns from the `MPI_Recv()` call when the receive buffer has been filled with valid data from a sending process. In contrast, when a process calls a non-blocking function, i.e., `MPI_Isend()` or `MPI_Irecv()`, it returns immediately from it. But in this case, the communication buffer may still be in use, because the message has not necessarily already been sent/received.

Since the relevant data will be available after blocking communication, capturing such communications is straightforward. As seen in the previous section, the process stores the trace information after is returned from the relevant PMPI call. For the same reason, capturing non-blocking sending is done after it is started by the process. Conversely, capturing non-blocking receive is not trivial. This section is dedicated to give more details about this procedure.

In fact, when a process starts a non-blocking communication, it allocates a communication request object and associates it with the request handle (the argument `request`). This request can be used later with other MPI calls like `MPI_Request_get_status()` to query the status of the communication.

When the function `MPI_Irecv()` is called, the trace information, i.e., data and metadata, is captured by intercepting the calls of the functions: `MPI_Irecv()` and `MPI_Wait()`. This is summarized as follows:

1. Firstly, when a call of the function `MPI_Irecv()` is intercepted, a tuple (Key: pointer to request object, Value: pointer to trace information) is stored in a temporary hash table. Instead of the received data itself, the trace information at this time contains a pointer to the buffer which is allocated by the user to receive this data later.

2. Secondly, if the request object allocated by the function `MPI_Irecv()` in the previous step is passed as an argument to the function `MPI_Wait()`, the function `MPI_Request_get_status()` is called with this object to check if the related communication is completed (see Listing 4.11). Whenever it is completed, the receive buffer will be filled with data. Hence, the trace information associated with the request in the hash table can be stored permanently in the dedicated data structures. Finally, the request object with its associated value is removed from the temporary hash table. However, if the function `MPI_Wait()` was not called, inspecting the temporary hash table before the finalization of the MPI program is conducted to determine if the request is completed, and hence, trace information needs be recorded.

Basically, the function `PMPI_Wait()` could be called to wait for the request to be completed without implementing this waiting manually in Listing 4.11. But the destructive behavior of this function prevents tracking the request object later on. In other words, after the call to `PMPI_Wait()` is returned, the request object will be deallocated and the request handle associated with it will be set to `MPI_REQUEST_NULL`. However, the function `PMPI_Wait()` is called at the end in order to destruct the request object.

4.8 Recording/Retrieving User Data

The extended capture data model presented in the previous chapter allows the user to record data on demand, particularly, the input data needed for the CUT. The function `h5mr_record_data()` is dedicated for this purpose. Listing 4.18 shows this function.

```
1  void h5mr_record_data(const char* unique_key, MPI_Datatype datatype, int
       ↪ count, void* buf)
2  {
3      if ( ! h5mr_is_enabled() ){
4          return;
5      }
6
7      int position = -1;
8      position = h5mr_append_buffer_to_datatype_ds (datatype, buf, count);
9      assert ( position >= 0 );
10
11     int success = h5m4_write_record_log (unique_key, datatype, count,
           ↪ position);
12     assert (success == 0);
13
14     return;
15 }
```

Listing 4.18: `h5mr_record_data()`

The data recorded by this function can be retrieved by calling the function `h5mr_read_data()`. Listing 4.19 shows this function:

```
1  int h5mr_read_data(const char* unique_key, MPI_Datatype datatype, int count,
       ↪ void* buf)
2  {
3      int ret = h5mr_fetch_captured_data( unique_key, datatype, count, buf );
4      return ret;
5  }
```

Listing 4.19: `h5mr_read_data()`

Chapter Summary

In this chapter, highlights of different implementation aspects were introduced. Firstly, the choice of programming language was justified. After that, the algorithm used to decode any given MPI datatypes was explained. The goals of this decoding were also clarified. Following that, the procedure used to identify the communicators in which a captured interaction is performed was outlined. An overview of the MPI libraries used during capture and replay was also given. Allowing the user to record data during capture and retrieve it during replay was illustrated at the end of this chapter.

Next, measured capture and replay executions are evaluated for a set of experiments.

5 Evaluation

A thorough evaluation of H5MR includes an assessment of requirements. This covers the correctness of the functional requirements and the acceptance of the non-functional requirements like performance. To measure the performance, the overhead of H5MR is determined. Overhead in terms of computation time and storage space is briefly assessed in Section 5.1. Validation of the solution and a correct implementation of it are very important to provide sufficient results. To validate the results, the output of the capture phase is examined in Section 5.2. Section 5.3 shows how to use the developed solution to create test units for an MPI program. The created test unit is used to demonstrate that the developed solution replays the execution correctly by using merely one process.

5.1 Overhead of H5MR

Testing with H5MR intercepts regular MPI function calls and records the transmitted data between the interacted processes during the capture phase, hence, this testing influences the application performance. A qualitative analysis of the caused overhead is important to verify that this influence is neglectable in comparison to the results that are presented in this thesis.

In the following small experiment, the overhead caused by H5MR in the capture phase is determined. For this purpose, a simple test program is used. In this program, an even number of processes perform point-to-point communications. More precisely, each process with an odd rank receives data from the process with the previous rank. For instance, Rank `1` receives data from Rank `0`. Likewise, Rank `2` sends data to Rank `3`. In addition, each operation is repeated `100000` times by a loop. In order to introduce a non trivial run-time overhead, data of a user-defined datatype has been transmitted by these operations. Consequently, the difference between the execution times of each MPI operation with and without capturing the MPI data is measured. The source code of this program is provided in Listing 5.1.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <time.h>
5  #include <mpi.h>
6
7  #define TIME_ACCURACY   1E9
8  #define COUNT           100000
9
10 int main(int argc, char* argv[]) {
```

```
11
12      int rank, i;
13      MPI_Status status;
14
15      struct timespec program_start, program_stop;
16      double total_program_time;
17
18      MPI_Init(&argc,&argv);
19
20      struct timespec start, stop;
21      double total_time;
22
23      if( clock_gettime( CLOCK_MONOTONIC , &program_start) == -1 ) {
24          exit( EXIT_FAILURE );
25      }
26
27      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
28
29      const int count       = 2;
30      const int blocklength = 3;
31      const int stride      = 4;
32
33      // Create two user-defind datatypes to use them in transmission which
            ↪ increases the overhead
34      MPI_Datatype c_datatype, v_datatype;
35      MPI_Type_vector(count, blocklength, stride, MPI_INT, &v_datatype);
36      MPI_Type_commit(&v_datatype);
37      MPI_Type_contiguous(3, v_datatype, &c_datatype);
38      MPI_Type_commit(&c_datatype);
39
40      if(rank % 2 == 0){
41          // Even processes send the data and measure the run-time of this
                ↪ operation
42          int send_buffer[63];
43          for (i=0; i<63; i++) {
44              buffer[i] = i;
45          }
46
47          if( clock_gettime( CLOCK_MONOTONIC , &start) == -1 ) {
48              exit( EXIT_FAILURE );
49          }
50
51          for(int i=0; i < COUNT; i++){
52              MPI_Send(send_buffer, 3, c_datatype, rank + 1, 52,
                    ↪ MPI_COMM_WORLD);
53          }
54
55          if( clock_gettime( CLOCK_MONOTONIC , &stop) == -1 ) {
56              exit( EXIT_FAILURE );
57          }
58
59          total_time = (( stop.tv_sec - start.tv_sec ) + ( stop.tv_nsec -
                ↪ start.tv_nsec ) / TIME_ACCURACY)/ COUNT;
60          printf( "Run-time of send operation call: %lf\n", total_time );
61
62      } else if(rank % 2 == 1) {
63          // Odd processes receive the data and measure the run-time of this
                ↪ operation
64          int receive_buffer[63];
65          for (i=0; i<63; i++){
66              receive_buffer[i] = -1;
67          }
68
```

```
69          if( clock_gettime( CLOCK_MONOTONIC , &start) == -1 ) {
70              exit( EXIT_FAILURE );
71          }
72
73          for(int i=0; i < COUNT; i++){
74              MPI_Recv(receive_buffer, 3, c_datatype, rank -1, 52,
                    ↪ MPI_COMM_WORLD, &status);
75          }
76
77          if( clock_gettime( CLOCK_MONOTONIC , &stop) == -1 ) {
78              exit( EXIT_FAILURE );
79          }
80
81          total_time = (( stop.tv_sec - start.tv_sec ) + ( stop.tv_nsec -
                ↪ start.tv_nsec ) / TIME_ACCURACY)/ COUNT;
82          printf( "Run-time of receive operation call: %lf\n", total_time );
83      }
84
85      MPI_Type_free(&v_datatype);
86      MPI_Type_free(&c_datatype);
87
88      if( clock_gettime( CLOCK_MONOTONIC , &program_stop) == -1 ) {
89          exit( EXIT_FAILURE );
90      }
91
92      // Compute the total execution time of the program
93      total_program_time = (( program_stop.tv_sec - program_start.tv_sec ) + (
            ↪ program_stop.tv_nsec - program_start.tv_nsec ) / TIME_ACCURACY);
94      printf( "Run-time of process %d: %lf\n", rank, total_program_time );
95
96      MPI_Finalize();
97      return 0;
98  }
```

Listing 5.1: Source code used to evaluate the capturing overhead.

Test system

This program is executed on a computer that is equipped with an **Intel Xeon Processor X5650** [20]. The processor clock-rate is 2.67 GHz.

In each experiment, the test program is executed by two processes. First of all, it is executed without linking against H5MR. After linking it against H5MR, it is executed twice: with and without capturing the MPI data.

Measured times are shown in Table 5.1. The wall-clock time gives the total execution time of the program. When capturing is enabled, initialization and finalization take some time. During the initialization, resources used to create the data structures that maintain the captured data must be allocated. During the finalization, these resources must be deallocated. This additional overhead is ignored. The time per operation is computed by dividing the execution time of the loop by the number of iterations. Computing the time of sending appears in Line 59 in Listing 5.1, whereas measuring the time of receiving is done in Line 81.

	Unlinked H5MR	Linked H5MR	
		Capturing	
		deactivated	activated
Wall-clock time	0.093377 s	0.097977 s	26.607060 s
Time for `MPI_Send()`	0.000001 s	0.000001 s	0.000266 s
Time for `MPI_Recv()`	0.000001 s	0.000001 s	0.000266 s

Table 5.1: Time overhead caused by H5MR.

Discussion of the results The overhead of the tracing delays execution about 270 times. The produced overhead is high, because the trace information is written out to the file system directly. This overhead can be reduced by using the *Double Buffering* strategy where two buffers are dedicated to cache the captured information. When the first buffer capacity is reached, its content is written out to a file system in the background while the second buffer can be filled at the same time. Hence, the first one will be ready for usage again.

However, when the test program is linked against H5MR without capturing any MPI data, no overhead is produced. Since the user is expected to record the interactions selectively only in a small CUT, most time during developing a scientific application, there shall not be any overhead in the final application.

Storage requirements

The testing environment must output trace information to HDF5 files. Table 5.2 shows the size of the two files produced when executing the considered test program by two processes.

Rank	Size of the produced trace file	Bytes per MPI operation
0	43.7 MiB	458.23
1	45.2 MiB	473.96

Table 5.2: Size of the produced trace files.

Dividing the size of the file written by `Rank 0` by the number of the captured `MPI_Send()` calls, i.e., the number of iterations, gives the amount of data required to record one data entry. For the considered test program, capturing one send operation needs about 458.23 Bytes. Applying the same approach on the file written by `Rank 1` shows that about 473.96 Bytes are required to capture one receive operation.

5.2 Demonstration

According to the design principles discussed in Chapter 3, capturing the execution of an MPI application with H5MR does not change the output of this application, i.e., it is

the same when running without H5MR. In addition to the original output, a file per process that executed the CUT is produced. Each file contains information about the interactions in which the process participated. Hence, fetching the data contained in each file correctly is the essential goal of the replay phase. In this section, the results of the capture phase are examined, whereas Section 5.3 is dedicated to prove the correctness of the results of the replay phase.

To examine the results of the capture phase, the execution of an MPI program is completely captured. After that, the output of this execution is investigated. The test oracle will be the solution described in Section 3.4. Listing 5.2 shows this program. In this program, two user-defined datatypes are created. Data of these datatypes are exchanged between processes. Firstly, in the default communicator `MPI_COMM_WORLD`, `Rank 0` sends data to `Rank 1`. After that, the communicator `MPI_COMM_WORLD` is split into two new communicators and the same transmission is repeated, i.e., `Rank 0` sends data to `Rank 1` in each new communicator. Once again, each newly created communicator is split into two new communicators and the same transmission is repeated. In this program, each process participated in two newly created communicators in addition to the two build-in communicators `MPI_COMM_WORLD` and `MPI_COMM_SELF`. This program should be executed by an even number of processes.

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  #include "h5mr.h"  // Linking the Library
5
6  int main( int argc, char* argv[] )
7  {
8      h5mr_init(1);  // Capture the whole execution
9      int i;
10     MPI_Status status;
11     MPI_Init(&argc,&argv);
12
13     const int count       = 2;
14     const int blocklength = 2;
15     const int stride      = 4;
16
17     // Creating two user-defined datatypes
18     MPI_Datatype c_datatype, v_datatype;
19     MPI_Type_vector(count, blocklength, stride, MPI_INT, &v_datatype);
20     MPI_Type_commit(&v_datatype);
21     MPI_Type_contiguous(2, v_datatype, &c_datatype);
22     MPI_Type_commit(&c_datatype);
23
24     int world_rank, world_size;
25     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
26     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
27
28     if (world_rank==0) {
29         int send_buffer[63];
30         for (i=0; i<63; i++)
31             send_buffer[i] = i;
32         // In MPI_COMM_WORLD, Rank 0 sends data to Rank 1 twice.
33         // Every time, data of the user-defined datatype created recently.
34         MPI_Send(send_buffer, 3, v_datatype, 1, 1148, MPI_COMM_WORLD);
35         MPI_Send(send_buffer, 2, c_datatype, 1, 1147, MPI_COMM_WORLD);
```

```
36
37      } else if (world_rank==1) {
38          int receive_buffer[63];
39          for (i=0; i<63; i++)
40              receive_buffer[i] = -1;
41          // In MPI_COMM_WORLD, Rank 1 receives the data sent by Rank 0.
42          MPI_Recv(receive_buffer, 3, v_datatype, 0, 1148, MPI_COMM_WORLD,
                ↪ MPI_STATUS_IGNORE);
43          MPI_Recv(receive_buffer, 3, c_datatype, 0, 1147, MPI_COMM_WORLD,
                ↪ MPI_STATUS_IGNORE);
44
45          for (i=0; i<63; i++)
46              printf("Received: buffer[%d] = %d\n", i, receive_buffer[i]);
47
48          fflush(stdout);
49      }
50
51      int color = world_rank / 4; // Determine color based on row
52
53      // Split the communicator MPI_COMM_WORLD based on the color and use the
            ↪ original rank for ordering
54      MPI_Comm row_comm;
55      MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);
56
57      int row_rank, row_size;
58      MPI_Comm_rank(row_comm, &row_rank);
59      MPI_Comm_size(row_comm, &row_size);
60
61      if(row_rank==0){
62          int send_buffer[63];
63          for (i=0; i<63; i++)
64              send_buffer[i] = i;
65          // In the new communicator row_comm, Rank 0 sends data to Rank 1
                ↪ twice.
66          // Every time, data of the user-defined datatype created in the
                ↪ beginning.
67          MPI_Send(send_buffer, 3, v_datatype, 1, 1150, row_comm);
68          MPI_Send(send_buffer, 2, c_datatype, 1, 1149, row_comm);
69
70      } else if(row_rank==1) {
71          int receive_buffer[63];
72          for (i=0; i<63; i++)
73              receive_buffer[i] = -1;
74          // In the new communicator row_comm, Rank 1 receives the data sent by
                ↪ Rank 0.
75          MPI_Recv(receive_buffer, 3, v_datatype, 0, 1150, row_comm,
                ↪ MPI_STATUS_IGNORE);
76          MPI_Recv(receive_buffer, 3, c_datatype, 0, 1149, row_comm,
                ↪ MPI_STATUS_IGNORE);
77
78          for (i=0; i<63; i++)
79              printf("Received: buffer[%d] = %d\n", i, receive_buffer[i]);
80
81          fflush(stdout);
82      }
83
84      // Split the communicator row_comm based on the color
85      int color_2 = row_rank / 2; // Determine color based on row
86      MPI_Comm row_comm_2;
87      MPI_Comm_split(row_comm, color_2, row_rank, &row_comm_2);
88
89      int row_rank_2, row_size_2;
90      MPI_Comm_rank(row_comm_2, &row_rank_2);
```

```
91      MPI_Comm_size(row_comm_2, &row_size_2);
92
93      if(row_rank_2==0){
94          int sent_buffer[63];
95          for (i=0; i<63; i++)
96              send_buffer[i] = i;
97          // In the new communicator row_comm_2, Rank 0 sends data to Rank 1
                ↪ twice.
98          // Every time, data of the user-defined datatype created in the
                ↪ beginning.
99          MPI_Send(send_buffer, 3, v_datatype, 1, 1152, row_comm_2);
100         MPI_Send(send_buffer, 2, c_datatype, 1, 1151, row_comm_2);
101     } else if(row_rank_2==1) {
102         int receive_buffer[63];
103         for (i=0; i<63; i++)
104             receive_buffer[i] = -1;
105         // In the new communicator row_comm_2, Rank 1 receives the data sent
                ↪ by Rank 0.
106         MPI_Recv(receive_buffer, 3, v_datatype, 0, 1152, row_comm_2,
                ↪ MPI_STATUS_IGNORE);
107         MPI_Recv(receive_buffer, 3, c_datatype, 0, 1151, row_comm_2,
                ↪ MPI_STATUS_IGNORE);
108
109         for (i=0; i<63; i++)
110             printf("Received: buffer[%d] = %d\n", i, receive_buffer[i]);
111
112         fflush(stdout);
113     }
114
115     MPI_Comm_free(&row_comm);
116     MPI_Comm_free(&row_comm_2);
117     MPI_Type_free(&v_datatype);
118     MPI_Type_free(&c_datatype);
119
120     MPI_Finalize();
121     return 0;
122 }
```

Listing 5.2: Source code used to evaluate the implementation.

After compiling this program and executing it by 8 processes, it will be checked if the captured execution outputs the same output as the original one. Subsequently, to demonstrate the correctness and implementation issues, the produced trace files are investigated. Note that this does not provide a proof of flawlessness of the code but demonstrates the general correctness of the approach by illustrating which data is generated.

Output Checking After the execution is terminated successfully, the captured execution prints out the same messages and results as the original one (not shown). It demonstrates that the captured execution outputs the same output as the original execution.

Trace Files Investigation According to the design principles introduced in Chapter 3, each process creates the data structures required to capture the operations it performed. Following, the structures created by `Rank 0` are examined. For the purpose of clarity, additional datasets which are created by other processes are also examined.

DS_DATATYPE_DESCRIPTION This dataset contains a description for each datatype used by `Rank 0` associated with a unique identifier `ID`. This identifier is used to represent the related datatype in the other datasets. Table 5.3 shows the content of this dataset:

ID	Description
1	`CONTIGUOUS(count=2,typ=VECTOR(count=2,blocklength=2,stride=4,typ=NAMED(INT),size=16,extent=24),size=32,extent=48)`
2	`VECTOR(count=2,blocklength=2,stride=4,typ=NAMED(INT),size=16,extent=24)`

Table 5.3: `DS_DATATYPE_DESCRIPTION` dataset created by `Rank 0`

DS_COMMUNICATOR This dataset contains a unique identifier of each communicator in which the process interacted. In addition, it contains the size of this communicator and the rank given to the process in it. Table 5.4 shows this information regarding `Rank 0`.

ID	Rank	Size
2	0	8
3	0	4
4	0	2

Table 5.4: `DS_COMMUNICATOR` dataset created by `Rank 0`

The `DS_COMMUNICATOR` created by `Rank 7` is shown in Table 5.5.

ID	Rank	Size
4	1	2

Table 5.5: `DS_COMMUNICATOR` dataset created by `Rank 7`

DS_SEND This dataset maintains metadata about all the blocking send operations performed by `Rank 0`. Table 5.6 shows its content.

Ret	Count	Position	DatatypeID	Destination	Tag	CommID
0	3	0	1	1	1147	2
0	2	0	2	1	1148	2
0	3	3	1	1	1149	3
0	2	2	2	1	1150	3
0	3	6	1	1	1151	4
0	2	4	2	1	1152	4

Table 5.6: `DS_SEND` dataset created by `Rank 0`

DS_RECV The information seen in the table above has to be consistent with the metadata contained in the dataset DS_RECV which is created by Rank 1. It is shown in Table 5.7.

Ret	Count	ActuallyReceived	Position	DatatypeID	Source	Tag	CommID
0	3	3	0	1	0	1147	2
0	3	2	0	2	0	1148	2
0	3	3	3	1	0	1149	3
0	3	2	2	2	0	1150	3
0	3	3	6	1	0	1151	4
0	3	2	4	2	0	1152	4

Table 5.7: DS_RECV dataset created by Rank 1

DS_DATATYPE_1 This dataset stores a copy of the data of the datatype represented by ID = 1 (see Table 5.3) and transmitted by Rank 0. Table 5.8 shows its content.

INT_1	INT_2	INT_3	INT_4	INT_5	INT_6	INT_7	INT_8
0	1	4	5	6	7	10	11
12	13	16	17	18	19	22	23
24	25	28	29	30	31	34	35
0	1	4	5	6	7	10	11
12	13	16	17	18	19	22	23
24	25	28	29	30	31	34	35
0	1	4	5	6	7	10	11
12	13	16	17	18	19	22	23
24	25	28	29	30	31	34	35

Table 5.8: DS_DATATYPE_1 dataset created by Rank 0

DS_DATATYPE_2 This dataset stores a copy of the data of the datatype represented by ID = 2 (see Table 5.3) and transmitted by Rank 0. Table 5.9 shows its content.

INT_1	INT_2	INT_3	INT_4
0	1	4	5
6	7	10	11
0	1	4	5
6	7	10	11
0	1	4	5
6	7	10	11

Table 5.9: DS_DATATYPE_2 dataset created by Rank 0

The other trace files will not be examined, however, they are correct as well.

5.3 Towards Unit Testing

As mentioned in the introduction, creating test units for MPI programs which can be executed by a single process is not possible when the code to be tested depends on data that stems from other processes. In this section, a test unit for a subroutine in an MPI program is created. This subroutine issues MPI calls to communicate with other processes. The created test unit is executed by one process solely.

The goal of this section is to prove the results of this work in the replaying phase. It proves that this solution replays the execution from the perspective of one specific process. That is, the code executed by this process in the capture phase is executed again by one process, wherein the interactions with other processes are emulated. If the outputs are identical, then the developed solution replayed the execution correctly.

For this purpose, two programs are created: one for capturing and the other for replaying. The first program is an MPI program that models **Conwayś Game of Life** [13]. The second program is a test driver for a subroutine used in the first program. This subroutine is `model_timestep()`. It accepts an instance of the structure `data_model_t` as an argument. This structure has the following definition:

```
1  typedef struct{
2      int                 y_start;
3      int                 y_end;
4      int                 y_count;
5      unsigned char*      buff;
6      unsigned char***    cells;
7  } model_data_t;
```

Listing 5.3: The definition of the structure `model_data_t`.

Each instance of this structure represents a state. After passing an initial state to the subroutine `model_timestep()`, it applies the rules of the game simultaneously in order to generate the next state. The desired number of states to be generated is given by the user. To accelerate the generation process, this subroutine exchanges data from connected regions between processes. Listing 5.4 shows this subroutine.

```
1  void model_timestep(model_data_t* data)
2  {
3      unsigned char** d       = data->cells[0];
4      unsigned char** d_new   = data->cells[1];
5      data->cells[1]          = d;
6      data->cells[0]          = d_new;
7
8      // communication
9      // send row with: data->y_start to the process before (potentially wrap)
10     // send row with: data->y_end to the process after (potentially wrap)
11     int proc_before = (o.rank == 0) ? (o.size -1) : (o.rank - 1);
12     int proc_after = (o.rank + 1) % o.size;
13
14     MPI_Request request[4];
15     MPI_Status status[4];
16     if(o.verbosity > 2)
```

```
17          printf("send %d %d %d %d\n", 1, data->y_count, 0, data->y_count + 1);
18      MPI_Isend(d[1], o.x, MPI_CHAR, proc_before, 4712, MPI_COMM_WORLD, &
            ↪ request[0]);
19      MPI_Isend(d[data->y_count], o.x, MPI_CHAR, proc_after, 4711,
            ↪ MPI_COMM_WORLD, & request[1]);
20      // recv row data->y_start - 1 from the process before
21      // recv row data->y_end + 1 from the process before
22      MPI_Irecv(d[data->y_count + 1], o.x, MPI_CHAR, proc_after, 4712,
            ↪ MPI_COMM_WORLD, & request[3]);
23      MPI_Irecv(d[0], o.x, MPI_CHAR, proc_before, 4711, MPI_COMM_WORLD, &
            ↪ request[2]);
24      MPI_Waitall(4, request, status);
25
26      // compute next generation according to the number of neighbors, see:
            Wikipedia for the rules.
27      for(  int y=1; y <= data->y_count; y++){
28          int yb = y - 1;
29          int yt = y + 1;
30
31          for(int x=0; x < o.x; x++){
32              int xr = (x + 1) % o.x;
33              int xl = x == 0 ? o.x - 1 : x - 1;
34
35              int count = d[yt][xl] + d[yb][xl] + d[y][xl] + d[yt][x] +
                    ↪ d[yb][x] + d[yt][xr] + d[yb][xr] + d[y][xr] ;
36              if(count == 3){
37                  d_new[y][x] = 1;
38              }else if(count < 2){
39                  d_new[y][x] = 0;
40              }else if(count > 3){
41                  d_new[y][x] = 0;
42              }else{
43                  d_new[y][x] = 1;
44              }
45          }
46      }
47  }
```

Listing 5.4: Subroutine `model_timestep()`

Listing 5.5 shows the `main()` function of the MPI program where this subroutine is called. The program accepts three command line arguments:

- `x,y`: the two dimensions of the universe of the Game of Life.
- `t`: the desired number of states to be generated by the program.

```
1   int main(int argc, char** argv)
2   {
3       int printhelp = 0;
4       init_options();
5
6       MPI_Init(&argc, &argv);
7       MPI_Comm_rank(MPI_COMM_WORLD, &o.rank);
8       MPI_Comm_size(MPI_COMM_WORLD, &o.size);
9       option_parseOptions(argc, argv, options, &printhelp, o.rank == 0);
10      if(printhelp != 0){
11          if (o.rank == 0){
12              printf("\nSynopsis: %s ", argv[0]);
13              option_print_help(options, 0);
14          }
```

```
15          MPI_Finalize();
16          if(printhelp == 1){
17              exit(0);
18          }else{
19              exit(1);
20          }
21      }
22
23      if (o.rank == 0){
24          printf("Model starttime: ");
25          print_current_time();
26          printf("\nOptions:\n");
27          option_print_current_values(options);
28      }
29
30      model_data_t model_data;
31      model_init(&model_data);
32
33      timer t_start;
34      start_timer(&t_start);
35
36      for(int t = 0; t < o.timesteps; t++){
37          if(o.verbosity){
38              printf("%d: timestep: %d\n", o.rank, t);
39          }
40          // Generating the next generation depending on the last one.
41          model_timestep(&model_data);
42      }
43
44      double runtime = stop_timer(t_start);
45      if(o.rank == 0){
46          printf("End run-time: %.3fs endtime: ", runtime);
47          print_current_time();
48          printf("\n");
49      }
50
51      model_finalize(&model_data);
52
53      MPI_Finalize();
54      return 0;
55  }
```

Listing 5.5: `main()` function of a program that models `Conwayś Game of Life`.

Capture Phase In order to test the subroutine `model_timestep()`, one call of it has to be captured. In addition, the argument passed to it has to be recorded. For this purpose, some user contribution is needed. In Listing 5.7, this contribution is written in red. The code in Lines 11 and 12 links this program against H5MR. For the intended experiment, the call of `model_timestep()` which is issued by `Rank 3` when generating the sixth state is captured. Basically, many calls which are issued by any other process could be captured. Since generating each state depends on the last generated one, the code between Lines 56 and 63 captures the instance of the structure `model_data_t` that represents the fifth state. Moreover, the arguments given by the user (x,y,t) are also recorded in this section.

Recording these values aims to use them as arguments for the subroutine `model_-timestep()` when replaying the execution, i.e., when executing the test driver created

later. Passing the same input for the subroutine during capturing and replaying aims to compare the output of both executions. To compare them, a function which derives a checksum datum from each output is implemented. This function aims to detect any lack of equality between these outputs. Listing 5.6 shows this function.

```
1  int model_checksum(model_data_t* data)
2  {
3      int sum = 0;
4      for(int y=1; y <= data->y_count; y++){
5          for(int x=0; x < o.x; x++){
6              sum = sum * 11 + x * 21 + data->cells[0][y][x];
7          }
8      }
9
10     return sum;
11 }
```

Listing 5.6: Checksum Function

Line 67 in Listing 5.7 shows the call of this function in the `main()` function, whereas Line 69 shows how to record the checksum datum that is derived from the value returned by this function.

```
1  #include <stdint.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <mpi.h>
6  #include <model.h>
7  #include <util-main.h>
8  #include <util-time.h>
9  #include <option-parser.h>
10
11 #include "h5mr.h"               // Linking the capturing library
12 #include "h5mr-serializer.h"    // Linking the library to capture data on demand
13
14 int main(int argc, char** argv)
15 {
16     int printhelp = 0;
17     init_options();
18     h5mr_init(0);  // Capture the execution in manual mode
19
20     MPI_Init(&argc, &argv);
21
22     MPI_Comm_rank(MPI_COMM_WORLD, &o.rank);
23     MPI_Comm_size(MPI_COMM_WORLD, &o.size);
24     option_parseOptions(argc, argv, options, &printhelp, o.rank == 0);
25     if(printhelp != 0){
26         if (o.rank == 0){
27             printf("\nSynopsis: %s ", argv[0]);
28             option_print_help(options, 0);
29         }
30         MPI_Finalize();
31         if(printhelp == 1){
32             exit(0);
33         }else{
34             exit(1);
35         }
36     }
```

```
37
38      if (o.rank == 0){
39          printf("Model starttime: ");
40          print_current_time();
41          printf("\nOptions:\n");
42          option_print_current_values(options);
43      }
44
45      model_data_t model_data;
46      model_init(&model_data);
47
48      // measure time for main program
49      timer t_start;
50      start_timer(&t_start);
51
52      for(int t = 0; t < o.timesteps; t++){
53          if(o.verbosity){
54              printf("%d: timestep: %d\n", o.rank, t);
55          }
56          if( t == 5 && o.rank == 3){
57              h5mr_start_recording();
58              h5mr_record_data("o.x", MPI_INT, 1, &o.x);
59              h5mr_record_data("o.y", MPI_INT, 1, &o.y);
60              h5mr_record_data("t", MPI_INT, 1, &t);
61              h5mr_record_data("data.y", MPI_INT, 3, &model_data.y_start);
62              h5mr_record_data("data.cells", MPI_CHAR,(model_data.y_count + 2) * o.x,
                    model_data.cells[0][0]);
63          }
64          // Generating the next generation depending on the last one.
65          model_timestep(&model_data);
66          if( t == 5 && o.rank == 3){
67              static int checksum = model_checksum(&model_data);
68              // Save the value to compare it when replaying the execution
69              h5mr_record_data("checksum", MPI_INT, 1, &checksum);
70              h5mr_stop_recording();
71          }
72      }
73      // end
74      double runtime = stop_timer(t_start);
75      if(o.rank == 0){
76          printf("End run-time: %.3fs endtime: ", runtime);
77          print_current_time();
78          printf("\n");
79      }
80
81      model_finalize(&model_data);
82
83      MPI_Finalize();
84      return 0;
85  }
```

Listing 5.7: The `main()` function of the targeted program with the user contribution.

After compiling this program, it has to be executed by four processes. After the executing is terminated successfully, one HDF5 file, which is created by `Rank 3`, is produced. In order to show the content of this file, the tool HDFVIEW [22] can be used. This tool enables the user to manipulate the captured data without previous knowledge of HDF. This tool has the drawback, in the case of H5MR, that it is not aware of the relation between the datasets that maintain the metadata and those which contain the relevant data. This may lead to an unintentional wrong linking of these datasets by the user.

Figure 5.1 shows the file `mpi-hdf5-recorder_3.h5` opened by HDFVIEW. It exhibits general information about this file (size, path, number of attributes) and displays the datasets maintained in it.

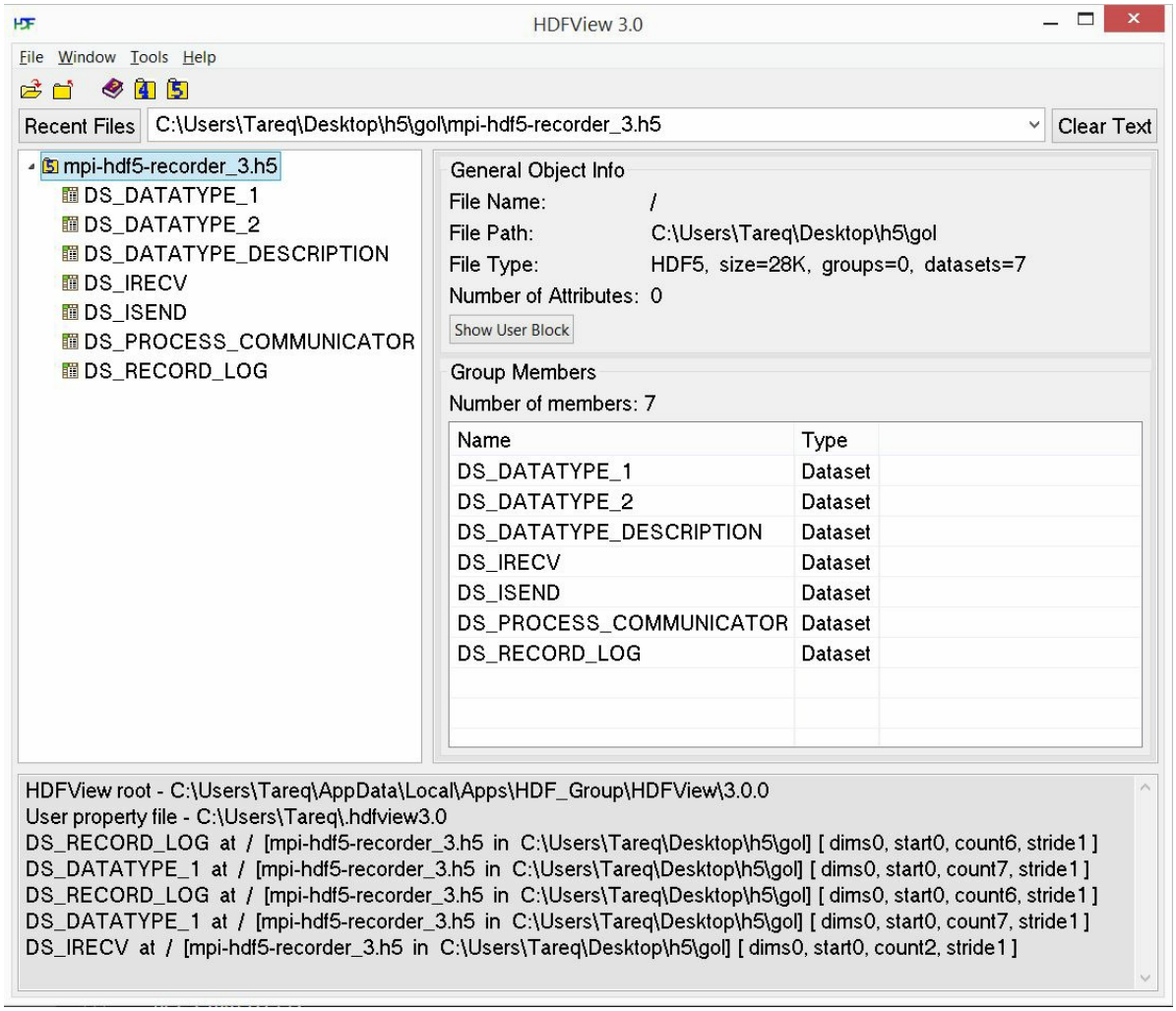
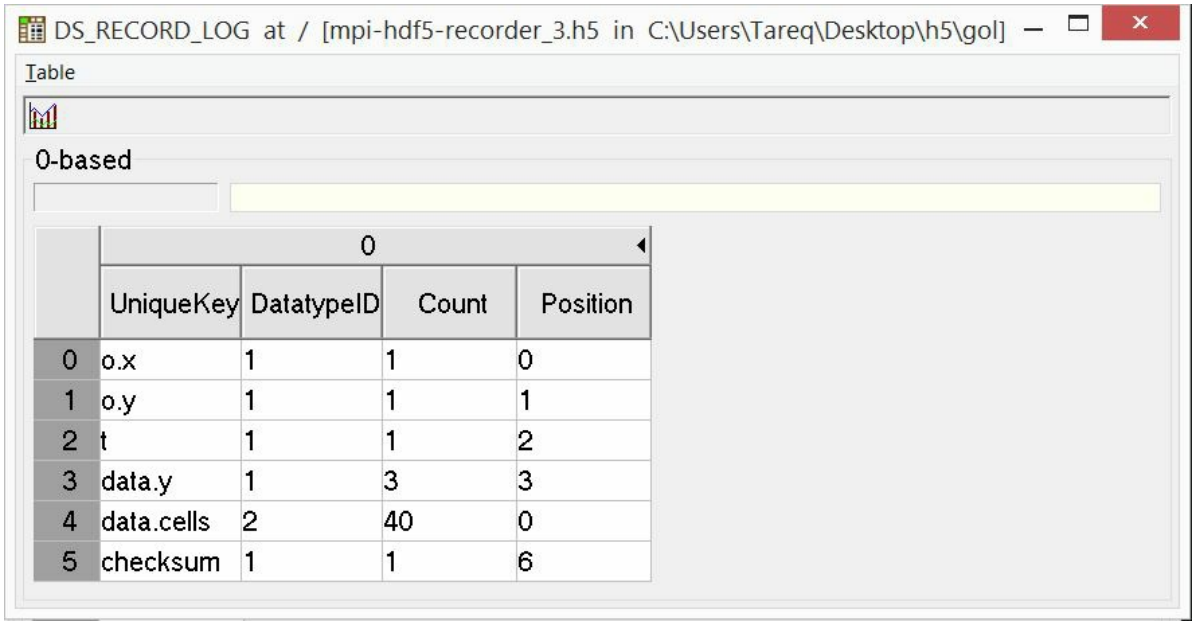


Figure 5.1: HDFVIEW: General information about the file `mpi-hdf5-recorder_3.h5`

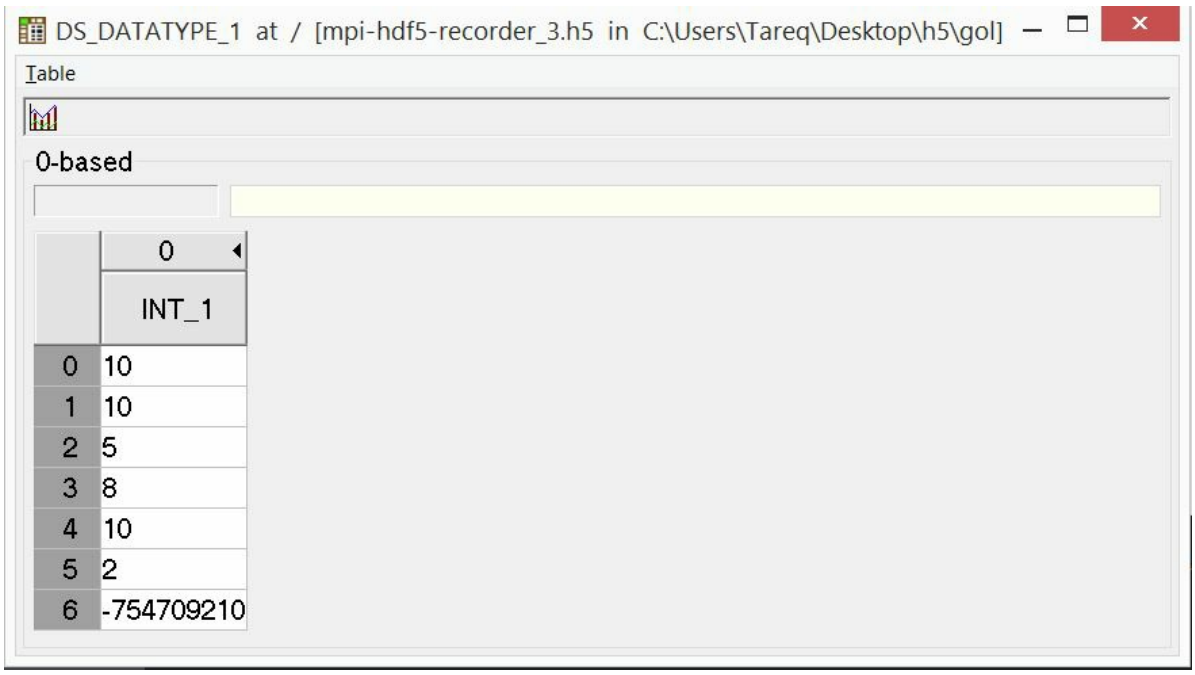
Figure 5.2 shows the dataset `DS_RECORD_LOG` opened by HDFVIEW. This dataset stores metadata describing the data recorded by the user. Regarding the capture program seen in Listing 5.7, this data includes the arguments passed to the program in addition to the checksum datum which is derived from the output of the subroutine `model_timpestep()`. According to this dataset, the checksum datum is stored in the dataset `DS_DATATYPE_1` in `Position = 6`. HDFVIEW is also used to open this dataset in Figure 5.3.

Like any other dataset opened by HDFVIEW, the content of these datasets can be manipulated directly. However, HDFVIEW does not realize the relation between the datasets in the data model. Therefore, it does not prevent any modification that violates this relationship and may lead to data inconsistency.



	0			
	UniqueKey	DatatypeID	Count	Position
0	o.x	1	1	0
1	o.y	1	1	1
2	t	1	1	2
3	data.y	1	3	3
4	data.cells	2	40	0
5	checksum	1	1	6

Figure 5.2: HDFVIEW: `DS_RECORD_LOG` Dataset



	0
	INT_1
0	10
1	10
2	5
3	8
4	10
5	2
6	-754709210

Figure 5.3: HDFVIEW: `DS_DATATYPE_1` Dataset

Replay Phase In this phase, the captured call of the subroutine `model_timestep()` is replayed. As stated in the beginning, this call is issued by `Rank 3` and generated the sixth state (generation) starting from the initial one. In contrast to the original

execution, the replayed execution is done merely by one process. The interactions with the other processes are emulated.

Listing 5.8 shows the test driver created for testing the subroutine `model_timestep()`. The code between Lines 21 and 24 and in Line 43 is dedicated to fetch the values captured during the original execution. These values are used to initialize an instance of the structure `model_data_t` (see Listing 5.3). This initialized instance is used as an argument for the subroutine `model_timestep()` when calling it in Line 46. After that, a checksum value is derived from its output in Line 48. The code in Line 51 fetches the checksum value recorded in the original execution. In Line 53, the code prints both checksum values.

```
1  #include <stdint.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <mpi.h>
6  #include <model.h>
7  #include <util-main.h>
8  #include <util-time.h>
9
10 #include "h5mr-serializer.h"    // Linking the replaying library
11
12 model_options_t o;
13
14 int main(int argc, char** argv){
15     MPI_Init(&argc, &argv);
16     MPI_Comm_rank(MPI_COMM_WORLD, &o.rank);
17     MPI_Comm_size(MPI_COMM_WORLD, &o.size);
18
19     int t;
20     model_data_t model_data;
21     h5mr_read_data("o.x", MPI_INT, 1, &o.x);
22     h5mr_read_data("o.y", MPI_INT, 1, &o.y);
23     h5mr_read_data("t", MPI_INT, 1, &t);
24     h5mr_read_data("data.y", MPI_INT, 3, &model_data.y_start);
25
26     // initialize model data
27     model_data.buff = (unsigned char*) malloc((model_data.y_count + 2) * o.x
           ↪ * 2);
28     memset(model_data.buff, 0, (model_data.y_count + 2) * o.x * 2);
29     // create pointer arrays and point them to buff
30     model_data.cells   = (unsigned char ***) malloc(sizeof(void *) * 2);
31     model_data.cells[0] = (unsigned char **) malloc(sizeof(void *) *
           ↪ (model_data.y_count + 2) * o.y);
32     model_data.cells[1] = (unsigned char **) malloc(sizeof(void *) *
           ↪ (model_data.y_count + 2) * o.y);
33
34     unsigned char* buff = model_data.buff;
35     for(int y=0; y < model_data.y_count + 2; y++){
36         model_data.cells[0][y] = buff;
37         buff += o.x;
38     }
39     for(int y=0; y < model_data.y_count + 2; y++){
40         model_data.cells[1][y] = buff;
41         buff += o.x;
42     }
43     h5mr_read_data("data.cells", MPI_CHAR, (model_data.y_count + 2) * o.x,
           ↪ model_data.cells[0][0]);
```

```
44
45      // CALLING THE SUBROUTINE UNDER TEST
46      model_timestep(&model_data);
47
48      int checksum = model_checksum(& model_data);}
49      int checksum_read;
50      // Get the checksum datum derived in the capture phase.
51      h5mr_read_data("checksum", MPI_INT, 1, &checksum_read);
52      // Comparing the derived checksum data in both phases.
53      printf("%d == %d", checksum, checksum_read);
54
55      MPI_Finalize();
56      return 0;
57  }
```

Listing 5.8: Test driver for the subroutine `model_timestep()`.

As mentioned before, executing the program seen in Listing 5.7 by four processes produces one trace file which is created by `Rank 3`. Executing the test driver by using this file as a data provider outputs the same checksum value that was derived from the model `model_data` in the first execution. In other words, by only one process, the developed library replayed the execution of a specific process and in which this process interacted with other processes. This demonstrates the correctness of the developed solution in the replay phase.

Chapter Summary

In this chapter, the software created for this thesis is evaluated by demonstrating its usefulness for testing MPI programs and by validating its feasibility.

The systematic evaluation was divided into three stages: First, the overhead of H5MR was determined which provided insight into its applicability. Then, an MPI program was executed twice: with and without tracing. The outputs of both executions were compared to prove that the developed solution does not present any variation from the original output of the targeted program. After that, a test unit for a subroutine was created. This test unit was executed by a single process although the tested subroutine uses data that stems from other processes.

In the following chapter, the thesis is summarized and concluded.

6 Conclusion

This chapter summarizes the findings from the previous chapters and covers future work. Discussing the results of the work is important to indicate their implications. Section 6.1 discusses those results and how they fulfill the aim of the work. In Section 6.2, limitations of the current approach are listed. It is followed by an assessment of future work in Section 6.3.

6.1 Discussion

With the increasing use of MPI to make computer-aided research by domain scientists without strong software engineering-background, facilitating testing of MPI programs by supporting users in creating test drivers will be beneficial to the HPC society.

In an effort to ease testing of MPI applications, the goal of the thesis was to develop a solution for capturing the execution of such applications and replaying the execution of any specific process in isolation. This approach was useful as it reduced the complexity of interacting parallel executions. Moreover, it kept the user away from running the expensive execution of such applications during testing. The interactions between this process and other processes were emulated. In this way, the user could focus on one branch of execution in the course of testing.

Avoiding running large configurations during testing was one of the goals of this work. The developed solution is applicable by linking a library against the targeted program and executing it. After that, the user may replay the execution by using the same source code but after linking it to a provided MPI library. This library does not conduct any communication but fetches data from a trace file. The user can also create a test driver for the code whose execution was captured. The created test driver has to be linked against this library too.

H5MR enables MPI users to test any section of their programs by selecting the piece of code that executes this section and testing it. Moreover, it allows inspecting the execution of the tested section from the perspective of any process that executed it.

In the context of refactoring MPI applications, H5MR can be useful. When capturing the execution of MPI code before and after refactoring, comparing the produced trace files from both executions can reveal any non-equivalence between the original code and the refactored one. This makes MPI applications easier to maintain, optimize, and

expand, because it prevents *code regression*, i.e., unintended side effects that emerge when changing code and break existing functionality.

When examining MPI code, some used datatypes may be complicated and difficult to be inspected. When capturing a data transmission operation in which this datatype is used, the developed solution creates a description for this datatype. It allows the user to discover the primitive datatypes that this datatype consists of. Moreover, it creates datasets dedicated to store the data of this datatype. It enables the user to examine the transmitted data of these datatypes accurately.

6.2 Limitations

Although the developed solution eases testing of MPI applications significantly, it suffers from some limitations. H5MR works only for MPI applications written in C. Moreover, in order to derive different testing scenarios, the user needs some knowledge of HDF5 in order to manipulate the data maintained in the generated HDF5 structures. The developed solution does not provide a build-in graphical user interface to enable the manipulation of the captured data in an easy way.

The user can use the tool HDFVIEW [22], which was presented in the previous chapter, in order to manipulate the captured data, but this still has a drawback. In the capture data model, the datasets that maintain the metadata are linked to those containing the relevant data, which is not understood by HDFVIEW. As a consequence thereof, modifying data may destroy the consistency or unintentionally link wrong data.

6.3 Future Work

With a prototype to proof the feasibility of the concept, only the first step is made towards a production tool. The broader vision is to gradually turn the current prototype into an actual open source tool that generates test units for existing MPI applications. In order for this vision to become a reality, some effort is necessary in terms of software development, in particular:

- Extending the MPI tracing/replaying libraries in order to intercept/emulate calls of more MPI functions.
- Enabling the user to manipulate the data captured in the HDF5 structures easily, for example, through a GUI like the tool HDFVIEW [22]. This GUI has to be aware of the relationship between datasets and maintain the data consistency.
- Using the developed software with FTG [23] to generate test units automatically.

The implementation of this work is published under **MIT** [24] license in a public repository[1] on **Github**.

[1]Repository: https://github.com/Tareq-Kellyeh/H5MR

Bibliography

[1] C. Hovy, J. Kunkel. *Towards Automatic and Flexible Unit Test Generation for Legacy HPC Code.* 2016.

[2] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference.* MIT press, 1995.

[3] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard – Version 3.1. Technical report, June 2015.
URL: http://www.mpi-forum.org/ (accessed on March 20, 2018).

[4] Open MPI.
URL: https://www.open-mpi.org/ (accessed on March 20, 2018).

[5] MPICH.
URL: https://www.mpich.org/ (accessed on March 20, 2018).

[6] B. Krammer, K. Bidmon, M. S. Müller, and M. M.Resch. *MARMOT: An MPI analysis and checking tool. Advances in Parallel Computing.* 13, 2004.

[7] J. Carver, R. Kendall, S. Squires, and D. Post. *Software Development Environments for Scientific and Engineering Software: A Series of Case Studies.* In 29th Int. Conf. on Software Engineering (ICSE), 2007.

[8] E. J. Weyuker. *On testing non-testable programs.* The Computer Journal, 25(4), 1982.

[9] U. Kanewala, J. M. Bieman. *Testing scientific software: A systematic literature review.* Information and Software Technology, Vol. 56, Nr. 10, 2014, S. 1219 - 1232.

[10] D. Heaton, J. C. Carver. *Claims about the use of software engineering practices in science: A systematic literature review.* Information and Software Technology, Vol. 67, Nr. C, 2015, S. 207-219.

[11] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz. *Understanding The High Performance Computing Community: A Software Engineer's Perspective.* IEEE Software, 25(4), 2008.

[12] The HDF Group. *High Level Introduction to HDF5.* The HDF Group, 2016.
URL: https://www.hdfgroup.org/ (accessed on March 20, 2018).

[13] Wikipedia. Conwayś Game of Life.
URL: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life (accessed on March 20, 2018).

[14] Intel Trace Analyzer and Collector.
URL: https://software.intel.com/en-us/intel-trace-analyzer (accessed on March 20, 2018).

[15] PGPROF: PGI Compilers and Tools. PGI Profiler User's Guide, 2017.
URL: http://www.pgroup.com/resources/docs/17.10/pdf/pgi17profug.pdf (accessed on March 20, 2018).

[16] PGI Compilers and Tools. PGI Profiler User Guide, 2015.
URL: https://www.softek.co.jp/SPG/Pgi/public/doc/pgprofug2015.pdf (accessed on March 20, 2018).

[17] PGDBG: PGI Compilers and Tools. PGI Profiler User's Guide, 2017.
URL: http://www.pgroup.com/resources/docs/17.10/pdf/pgi17dbug.pdf (accessed on March 20, 2018).

[18] Marmot.
URL: https://www.lrz.de/services/software/parallel/marmot/ (accessed on March 20, 2018).

[19] Vampir.
URL: https://www.vampir.eu/ (accessed on March 20, 2018).

[20] Intel Xeon Processor X5650.
URL: https://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI (accessed on March 20, 2018).

[21] GLib collections.
URL: https://developer.gnome.org/glib/stable/glib-data-types.html (accessed on March 20, 2018).

[22] HDFVIEW.
URL: https://support.hdfgroup.org/HDF5/Tutor/hdfview.html (accessed on March 20, 2018).

[23] FTG. A tool for automatically generating unit tests for subroutines of existing Fortran applications based on an approach called Capture & Replay.
URL: https://github.com/fortesg/fortrantestgenerator (accessed on March 20, 2018).

[24] MIT License. A permissive free software license originating at the Massachusetts Institute of Technology (MIT).
URL: https://opensource.org/licenses/MIT (accessed on March 20, 2018).

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ggf. streichen: Ich bin damit einverstanden, dass meine Abschlussarbeit in den Bestand der Fachbereichsbibliothek eingestellt wird.

Hamburg, den 20. März 2018

Tareq Kellyeh