PARCOACH RMA-Analyzer Manual

Release 0.0.1

Atos

Aug 09, 2022

CONTENTS

This guide describes the RMA-Analyzer, which is a module based on PARCOACH[1]. While PARCOACH initially aims at MPI collective errors detection in parallel applications, the RMA-Analyzer aims at implementing an error detection mechanism for MPI-RMA programs.

[1] PARallel COntrol flow Anomaly CHecker. https://parcoach.github.io/index.html

CHAPTER

ONE

INTRODUCTION

PARCOACH is a framework that aims at helping users programming MPI codes. It proposes an advanced aid for detecting errors when using MPI collective communications, non-blocking communications (i.e. `MPI_Isend/Irecv`), and correct usage of MPI routines in MPI + threads programs. It leverages a coupled static and dynamic analysis to detect conditions that may lead to deadlock due to MPI usage in parallel programs.

The RMA-Analyzer aims to extend the PARCOACH framework as a new module that addresses specifically the MPI-RMA section of the MPI standard. MPI-RMA seems promising in terms of performance, especially for overlapping communications with computations, but coding such programs can be quite challenging. The asynchronous nature of those communications raises however several types of memory consistency issues that the programmer is left to deal with. To help the programmer, the RMA-Analyzer module checks potential race conditions that can occur during an MPI-RMA program. It stops the program immediately if an error is detected, with a user-friendly message to help programmers locating the issue in their code, compared to other approaches that perform post-mortem analysis of errors.

This guide is organized in two parts:

- The *Implementation overview* section presents how the RMA-Analyzer is organized and how we instrument the user code to detect errors;
- The *Using the RMA-Analyzer* section presents how to use the RMA-Analyzer, from building your source code with it to an example on an erroneous code to fix.

CHAPTER

TWO

IMPLEMENTATION OVERVIEW

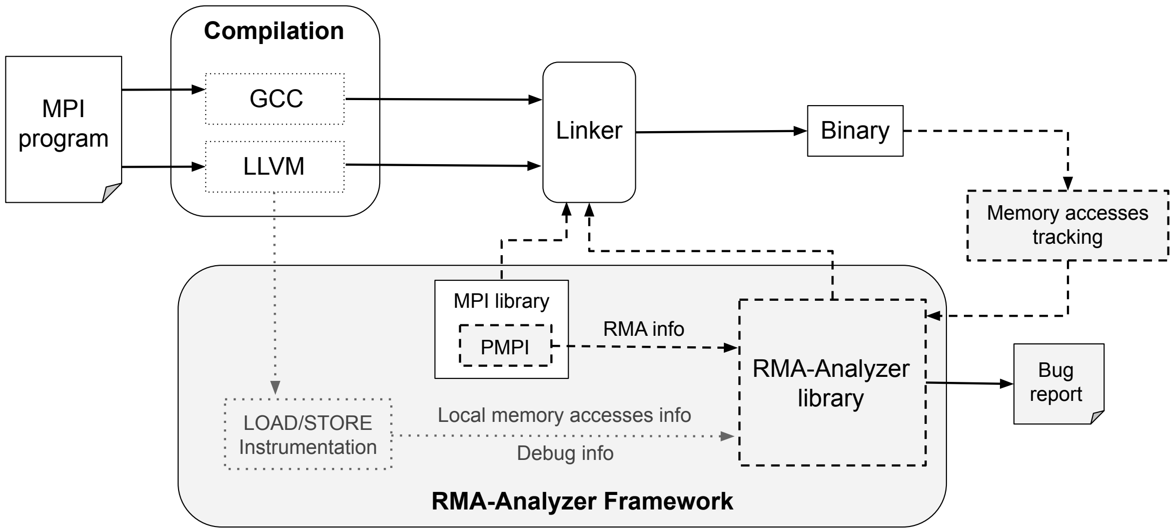

Fig. 1: *Figure 1: Overview of the RMA-Analyzer framework*

Figure 1: Overview of the RMA-Analyzer framework presents the workflow of the RMA-Analyzer. It is based on a dynamic analysis of the user code that allows to detect all the memory consistency errors that happen at runtime. To do so, the RMA-Analyzer implements an additional LLVM pass that is applied on the user code to instrument all the memory accesses that may interact with each other (i.e. local `load/store` accesses, remote MPI-RMA operations that lands on MPI windows, and local buffers used by MPI-RMA communications). By doing so, the RMA-Analyzer can detect at runtime when memory accesses may conflict, and immediately raise an error to the user before continuing the program with a potential silent memory corruption, so that programmers can fix the issue. More details can be found in[2].

The RMA-Analyzer supports user codes programmed in C and Fortran, and currently supports the following synchronization models and routines of the MPI-RMA standard:

- `MPI_Win_Lock_all/Unlock_all` synchronization model;
- `MPI_Win_fence` synchronization model;
- `MPI_Put/Get` RMA operations;
- `MPI_Put/Get/Test/Wait_notify` notified RMA operations

[2] "Static and Dynamic Data Race Detection for MPI-RMA Programs", Ait Kaci et al., EuroMPI'21

  - Only available when building with the Bull Open MPI software;

- `MPI_Barrier` global synchronization.

CHAPTER

THREE

USING THE RMA-ANALYZER

3.1 Prerequisites

To use the RMA-Analyzer tool, two pieces of software are needed:

- An LLVM-based toolchain to compile the user code so that the LLVM pass of the RMA-Analyzer can be used
  - For the Fortran support, an LLVM-based toolchain that has a Fortran front-end compiler that can generate LLVM IR is needed (e.g. Classic Flang[3]);
- An MPI implementation (e.g. Open MPI) that can use the LLVM-based toolchain as its internal compiler
  - For the Fortran support, the MPI implementation **must** be built with the LLVM-based toolchain to have the Fortran modules of the MPI implementation built with LLVM;
  - For the notified RMA support, the MPI implementation must be the Bull Open MPI.

WARNING

Only the Bull Open MPI implementation has been tested for now. While other MPI implementations should be compatible, please be aware that the support for the LLVM-based toolchain still must be ensured.

3.2 RMA-Analyzer workflow

Figure 2: Overview of the RMA-Analyzer usage workflow presents how users can benefit from the RMA-Analyzer. It spans in four main steps:

1. Compile the MPI-RMA application with the RMA-Analyzer, that will instrument all local and MPI-RMA memory accesses;
2. Run the program. During its execution, the RMA-Analyzer will analyze the memory accesses, searching for a potential memory consistency error;
3. If the RMA-Analyzer has not found any error, nothing is returned to the user. If there is an error, a user-friendly message is returned to the user so that it can identify the source of the memory consistency error;
4. After fixing its MPI-RMA application, the cycle restarts until the RMA-Analyzer has not found any error.

In the following sections, we present on a simple example called *rr_put_put* how to perform each step of the workflow. Its code is the following:

[3] Classic Flang. https://github.com/flang-compiler/flang

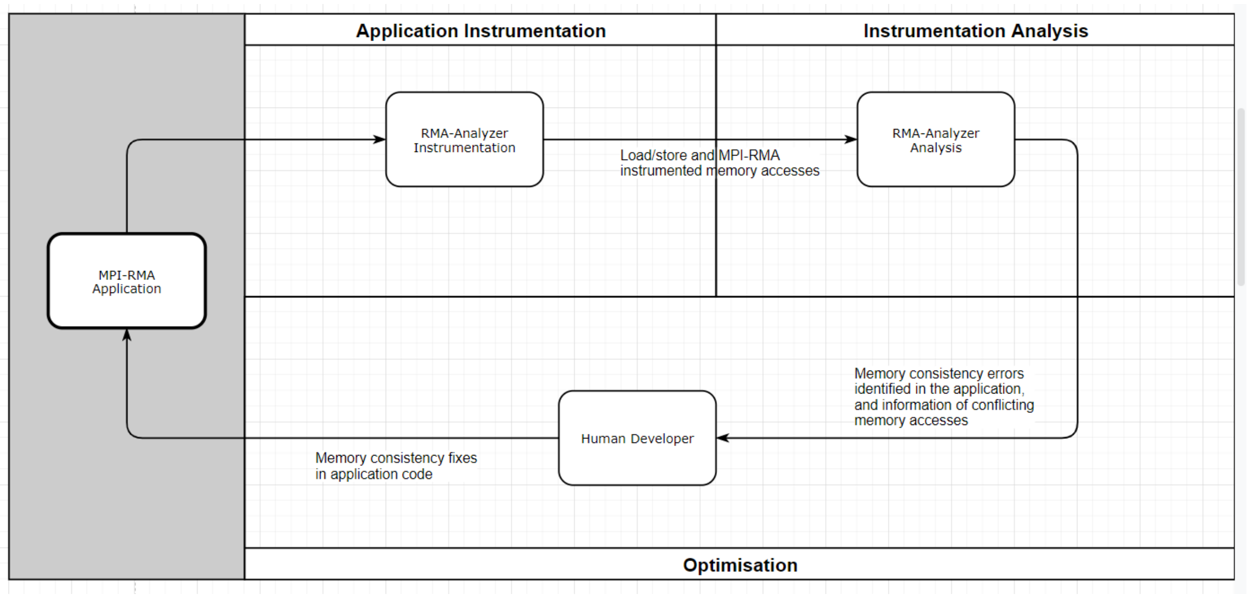
Application Instrumentation
Instrumentation Analysis
RMA-Analyzer Instrumentation
RMA-Analyzer Analysis
Load/store and MPI-RMA instrumented memory accesses
MPI-RMA Application
Memory consistency errors identified in the application, and information of conflicting memory accesses
Human Developer
Memory consistency fixes in application code
Optimisation

Fig. 1: *Figure 2: Overview of the RMA-Analyzer usage workflow*

```
################
# rr_put_put.c #
################

#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char* argv[])
{
  MPI_Win window;
  int provided;
  int comm_size;
  int my_rank;
  int value = 12345;
  int window_buffer[100] = {0};

  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

  MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
  if(comm_size != 3)
  {
    printf("This application is meant to be run with 3 MPI processes, not %d.\n",␣
↪comm_size);
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  MPI_Win_create(&window_buffer, 100*sizeof(int), sizeof(int), MPI_INFO_NULL,
                 MPI_COMM_WORLD, &window);
  MPI_Win_lock_all(0, window);

  if(my_rank == 0 || my_rank == 2)
```

(continues on next page)

(continued from previous page)

```
  {
    // Put value to target 1 at offset 80
    MPI_Put(&value, 1, MPI_INT, 1, 80, 1, MPI_INT, window);
  }

  MPI_Win_unlock_all(window);
  MPI_Win_free(&window);
  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

3.3 Compiling with the RMA-Analyzer

To build a program with the RMA-Analyzer, all of its files must be passed through the LLVM pass of the RMA-Analyzer. To do so, each file must be compiled in three steps:

1. First, generate the LLVM bytecode of the file by specifying `-c -emit-llvm` to the compiler;
2. Apply the LLVM pass of the RMA-Analyzer with the `opt -load` tool to build the instrumented LLVM bytecode;
3. Build the object file from the instrumented LLVM bytecode.

The last step to build with the RMA-Analyzer happens at linking time, when building the final executable. Since the routines instrumented by the LLVM pass are implemented in a library delivered with the RMA-Analyzer LLVM pass, it is also mandatory to link the binary with this RMA-Analyzer library.

If we build the *rr_put_put.c* file with an Open MPI implementation built with Clang, the chain of commands would look like this:

```
[... 1. Generate the LLVM bytecode ...]
$> clang -o rr_put_put.bc -c -emit-llvm rr_put_put.c
[... 2. Apply the LLVM pass of the RMA-Analyzer ...]
$> opt -load path/to/parcoachrma_install/lib/parcoachRMA.so rr_put_put.bc -o rr_put_
↪putINSTR.bc
[... 3. Build the object file from the instrumented bytecode ...]
$> clang -o rr_put_put.o -c rr_put_putINSTR.bc
[... 4. Build the final executable and link it to the RMA-Analyzer ...]
$> mpicc -Lpath/to/parcoachrma_install/lib -o rr_put_put rr_put_put.o -lrma_analyzer -
↪lpthread
```

Here is a Makefile example that performs the whole building step for the *rr_put_put.c* file:

```
CC=clang
MPICC=mpicc

PARCOACHRMA_ROOT?=path/to/parcoachrma_install
PARCOACH=$(PARCOACHRMA_ROOT)/lib/parcoachRMA.so
LIBS=-L$(PARCOACHRMA_ROOT)/lib -lrma_analyzer -lpthread

SRC_TEST= rr_put_put.c

OBJ_TEST=$(SRC_TEST:.c=.o)
EXEC_TEST=$(SRC_TEST:.c=)
```

(continues on next page)

(continued from previous page)

```
all: $(EXEC_TEST)

$(EXEC_TEST): $(OBJ_TEST)
        $(MPICC) -o $(shell basename $@) $@.o $(LIBS)

%.o: %.c
        $(CC) -o $*.bc -c -emit-llvm $<
        opt -load $(PARCOACH) -parcoach $*.bc -o $*INSTR.bc
        $(CC) -o $@ -c $*INSTR.bc
```

3.4 Launching a code built with the RMA-Analyzer

Now that the code is built with the RMA-Analyzer, we can try to launch it so see if an error is raised. If no error is raised, it means that the RMA-Analyzer has not found any memory consistency errors in the program, and thus it should be fine (from a memory consistency point of view).

However, in the case of the *rr_put_put* program, an error is happening, and the RMA-Analyzer returns an error that look like this:

```
$> mpirun -np 3 ./rr_put_put
[RMA-ANALYZER Process 1] Error when inserting memory access of type
RMA_WRITE from file rr_put_put.c at line 35 with already
inserted access of type RMA_WRITE from file rr_put_put.c
at line 35. The program will be exiting now with MPI_Abort.
```

The error that happens here is shown in *Figure 3: Example of erroneous execution of program rr_put_put*. The MPI process 0 and 2 both perform an `MPI_Put` targeted at MPI process 1 and at the same memory location without synchronization between the two, which is an error. We refer the readers that want to learn more about what kind of errors can happen within MPI-RMA programs to[?].

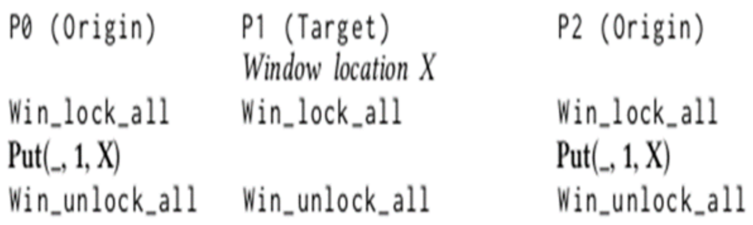

Fig. 2: *Figure 3: Example of erroneous execution of program rr_put_put*

One solution to solve this issue is to add a synchronization between the processes, to guarantee that the receiving process can perform the wanted operations on the data before the second communication arrives, and to order them. Other solutions are of course viable, depending of the wanted semantic of the program. Such fix could look like this:

```
$> diff rr_put_put.c rr_put_put_with_sync.c
32c32
<   if(my_rank == 0 || my_rank == 2)
---
>   if(my_rank == 0)
33a34,41
>     // Put value to target 1 at offset 80
```

(continues on next page)

(continued from previous page)

```
>     MPI_Put(&value, 1, MPI_INT, 1, 80, 1, MPI_INT, window);
>     MPI_Barrier(MPI_COMM_WORLD);
>   }
>
>   if(my_rank == 2)
>   {
>     MPI_Barrier(MPI_COMM_WORLD);
```

With your code fixed, you can try again compiling and running your code with the RMA-Analyzer, until the issue is fixed. In this case, introducing the synchronization fixes the issue.

RESTRICTIONS AND LIMITATIONS

- Since the dynamic analysis alone becomes prohibitive at scale, the RMA-Analyzer does not work on large code bases for now
  - A static analysis to complement the dynamic analysis is a work in progress;
- Since the RMA-Analyzer relies on an LLVM pass to perform its instrumentation of the user code, it is only compatible with compilers that can use an external LLVM pass (e.g. Intel OneAPI, Clang);
- The MPI-RMA standard is partially supported for now:
  - `MPI_Win_Flush*` in-epoch synchronization routines are not supported;
  - `MPI_Win_Lock/Unlock` synchronization model is not supported;
  - `MPI_Win_Post/Start/Complete/Wait` synchronization model is not supported;
  - Atomic operations (e.g. `MPI_Accumulate`, ...) are not supported.
- The RMA-Analyzer will be integrated as a plugin in the PARCOACH framework in a future version.