UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Mihkel Tiks

Further Work on a Causally Consistent Reversible Debugger for MPI Applications

Bachelor's Thesis (9 ECTS)

Supervisor(s): Stefan Kuhn, PhD

Tartu 2023

Further Work on a Causally Consistent Reversible Debugger for MPI Applications

Abstract:

Reversible debuggers, which enable users to move backwards through the code they are debugging, can help users better find issues within their code. The feature of reversibility can be even more useful in the case of debugging distributed programs, since they may include intermittent problems, such as race conditions and deadlocks. The debugger created here is focused on compatibility with MPI (Message Parsing Interface), which is the current standard for parallel computation. Reversibility is implemented in the Linux environment using checkpoint and restore methods in userspace. This thesis gives an overview of checkpoint and restoration software and options for this use case, and further develops the previous work of Martens titled “Causally Consistent Reversible Debugger for MPI Applications”[1].

Keywords:

Reverse debugging, MPI, distributed debugging, checkpointing, parallel programming, reversibility

CERCS: P170 – Computer science, numerical analysis, systems, control

Edasiarendused põhjuslikku järjepidavust tagaval tagasipööratavalsilujal MPI programmidele

Lühikokkuvõte:

Tagasipööratavad silurid, mis lubavad kasutajal liikuda tagurpidi programmi töövoos, saavad olla kasulikud programmi seest vigade leidmisel. Tagasipööratavusest võib aga veel rohkem kasu olla paralleelarvutustusi teostavate programmide silumisel, kuna nendes võivad esineda näiteks trügimised (ingl *race condition*) või tupikud (ingl *deadlock*), mis sageli ei ilmne igal käivitusel. Loodud silur on tehtud MPI (ingl *Message Passing Interface*) programmide jaoks, mis on praegune paralleelarvutuse standard. Tagasipööratavus implementeeritakse Linuxi keskkonnas kasutades kontrollpunkti ja taastamise meetodeid kasutajaruumis. Käesolevas töös antakse ülevaade kontrollpunktide loomise ja nendest taastamise meetoditest, ning kirjeldatakse edasiarendusi Martensi eelnevalt valminud magistritöö “Causally Consistent Reversible Debugger for MPI Applications”[1] rakenduses.

Võtmesõnad:

Tagasipööratav silumine, MPI, hajus silumine, kontrollpunktide loomine, paralleelarvutus, tagasipööratavus

CERCS: P170 – Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1. Introduction

For software development, debugging remains an important part of ensuring the reliability and efficiency of software systems. Regular debugging techniques have long been the foundation of this process, allowing developers to identify and fix errors found in code. However, as software systems become increasingly complicated, the need for more advanced debugging tools becomes apparent.

A less common feature of debuggers is reversibility, which means having the ability to reverse into a previous state of the program. Debuggers with this capability are called reversible debuggers. Reversible debuggers also often include additional features allowing execution backwards through the program, using commands such as reverse-step or reverse-continue. Being able to reverse to a previous state in the program instead of re-execution can save a lot of time with long-running programs.

Data from 2012 shows that developers allocate a significant portion of their programming time to debugging, making up approximately half of their overall workload. This translates to an estimated cost of $312 billion annually by the software industry solely on debugging. The integration of reversible debuggers reduced the time spent debugging by 26%. This efficiency translates to an estimated at $81.1 billion saved per year, thereby presenting a compelling case for the development of such tools.[2]

Furthermore, more and more programs are being written to be executed concurrently. Writing programs to be executed in parallel requires accounting for many more nuances than in the case of serial programs. Some of the problems that might occur are for example race conditions or deadlocks, which often happen intermittently. Debuggers are also of great help in finding the source of such issues.

The debugger created here is focused on compatibility with MPI (Message Parsing Interface), which is the current standard for parallel computation. This thesis gives an overview of checkpoint and restoration software and options for this use case, and further develops the previous work of Martens titled “Causally Consistent Reversible Debugger for MPI Applications”[1]. Reversibility is implemented in the Linux environment using checkpoint and restore methods in userspace. Included is an overview of known algorithms for reverse

debugging commands, such as reverse-continue and reverse-step, and the implementation of these techniques for a debugger of MPI programs.

2. Theoretical overview

This chapter provides an overview of debuggers and related tools. It also includes an overview of reversible debuggers, the theory behind reverse debugging commands, MPI and checkpointing techniques.

2.1 Debuggers

Debuggers are often used to diagnose sources of errors when encountering faulty program states. Debuggers allow users to control the workflow of a program, enabling, for example, pausing the program at a certain point or continuing step-by-step. For programmers, this is useful, as the tools often also provide features to help with analysing the program and finding the root of existing errors.

2.1.1 Code execution control

Debuggers give developers control over how their code runs. They allow developers to move directly to a line of code that they are interested in or through their code step by step, which means they can look at one line of code at a time. This helps them find exactly where problems are happening so they can fix them easily.

One of the most common debugging commands is “continue”. The "continue" function is a command in debuggers that allows developers to resume the execution of their code after it has been paused by a breakpoint or another debugging command. Essentially, when a debugger encounters a breakpoint or pauses execution due to a debugging command, the developer may want to continue running the code from that point onward to see how it behaves further.

Another important debugging command is “step”. The "step" function, on the other hand, allows developers to move through their code one line at a time during debugging. Unlike the "continue" function, which skips over portions of code until the next breakpoint or until the end of execution, the "step" function provides more granular control by executing each line of code sequentially.

Besides step and continue, different debuggers implement different variations of these commands, such as step over, which executes the next line of code without entering function

calls. These numerous commands allow developers to navigate through code with ease, helping them find and fix problems quickly.

2.1.2 Breakpoints

Debuggers can also pause the code at specific spots, called breakpoints. This is useful when developers want to see what is happening at a particular moment in the program. Users typically instruct the debugger to insert a breakpoint at a line, which they wish to arrive at. After that, the user would issue the command “continue” and the program would continue execution until it reaches the breakpoint.

Breakpoints are known as instructions, which, when executed, will stop the execution of the program, and pass control back to the operating system or the debugger. Debuggers implement setting breakpoints by changing memory regions of the program that is being debugged. The original instruction at the line is swapped out with a trapping instruction. This will cause the program to stop execution and return control to the debugger. Then the debugger typically inserts the initial instruction back to the place it was swapped out of[3].

2.1.3 Inspecting memory

A very common method of debugging is inserting printing statements into the code to see if the program reaches that state or what values certain variables have at that point. However, this becomes very cumbersome if the location of the error or the cause is unknown. One of the most important utilities of debuggers is that they use tools to allow the user to inspect the memory of the target program. Using additional tools, it is possible to retrieve the information at some place in the process’ memory. This allows the user to view and analyse the state of variables that have been defined in the program. This way, there is no need to explicitly select or print the variables that the user wants to display, as they can do it interactively using the debugger.

2.2 ptrace

ptrace[4] plays an important role in the functionality of debuggers and their operations. It allows a process (the tracer) to observe and control the execution of another process (the tracee), enabling various debugging functionalities such as the previously mentioned code execution control, breakpoints, and memory inspection.

ptrace provides crucial tools with respect to code execution control. It enables the debugger to gain control over the execution of the target program. It defines functions like *PTRACE_CONT*, which instructs the program to continue execution, and *PTRACE_SINGLESTEP*, which instructs the program to execute only the next instruction. These calls are used directly by the debugger to carry out commands executed by the user.

Furthermore, ptrace is also the primary actor when interfacing with the memory and registers of the target program. ptrace provides commands such as *PTRACE_POKEDATA*, which has the parameters of data and address, and will write the given data to the address. Another important command is *PTRACE_PEEKDATA*, which will return the information at the supplied address. For example, when we are setting breakpoints, a system call *PTRACE_POKEDATA* is used to write the interruption instruction into the memory address of the initial instruction. Similarly, *PTRACE_PEEKDATA* will be used when retrieving the values of variables from the program's memory.

2.3 DWARF Debugging information

When we compile C and C++ programs, we have the option to tell the compiler to compile with debugging information included. There are many formats of this information, but one of the most common ones and the one used here is DWARF.

DWARF is a standardised format to store debugging information inside an executable file. This information is used by debuggers to connect the compiled code back to the original source code. This allows programmers to understand what the program is doing during debugging sessions. It achieves this by describing the executable program in a tree structure. This tree structure contains information about variables, functions, and their types. Debuggers can use this information to display variable values, call stacks, and step through the program line by line [5].

The information that is produced includes for example names, declaration lines and locations of variables and functions. For example, in figure 1, we can see the DWARF data for a function *passMessages*, which is declared in the file circle.c and the function declaration is at line 12.

This information is essential to every debugging tool and working without it would be very difficult if not impossible. Using only ptrace, we are able to modify the target's memory, but

we have no concrete information about the memory addresses of the variables or functions that we are interested in. Using DWARF debugging information, we have the last component we needed – lines of source code mapped to their corresponding memory addresses. Combining this with ptrace, we have everything we need to execute the aforementioned steps for setting a breakpoint and continuing execution to it.

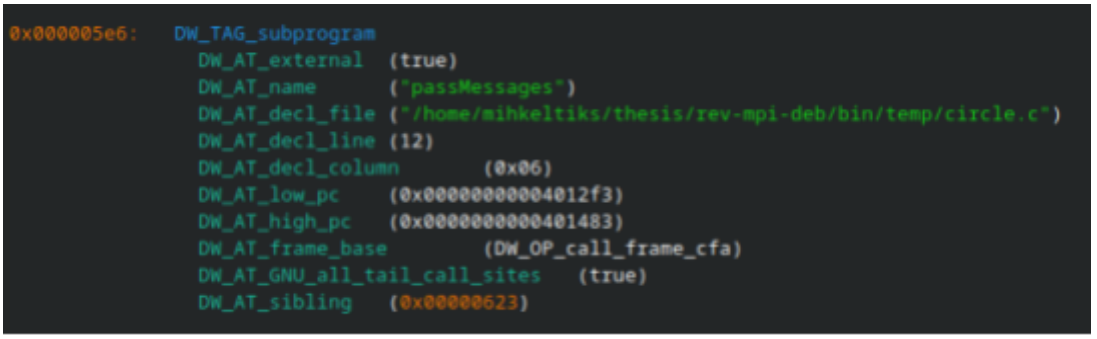

Figure 1. DWARF debugging information

2.4 Reversible debuggers

Reversible debuggers operate on a fundamentally different principle compared to traditional debuggers. While regular debuggers allow developers to move forward through their code, reversible debuggers introduce the ability to move both forward and backward in the execution flow. This bidirectional navigation allows developers to rewind program execution, inspect past states, and better locate the root cause of bugs.

In C, C++, FORTRAN and most other programming languages we are not able to actually undo an operation or thus execute in reverse. This is due to information being destroyed throughout the runtime of the program, for example when overwriting memory locations with new values. To implement reversibility with this restriction in mind, there are 2 common methods – recording and reconstruction[6].

2.4.1 Recording and reconstruction

In the case of recording, the program is executed once and all the necessary states are logged, so that they can be displayed later. In this case, reverting to a past state simply means displaying the state of the program at that position, which had been saved in the first execution. The notable drawbacks of this method are the necessity to generate large amounts of logs, which impact performance and memory requirements, and the fact that if the sought

after error does not occur on the recorded execution, then it will not be able to be analysed during debugging. This method is used in the Bachelor's Thesis of Alar Leemet[7], which reports memory usage increasing up to 30 times and execution time increasing up to 10 times for Python programs.

In the case of reconstruction, we save enough information about the process at some point in time, to be able to reconstruct it from that point later on. This way, the program is restored and actually executed again, instead of just simulating execution by displaying saved states. The procedure of saving information about the process will be referenced to as checkpointing and reconstruction will be referenced to as restoration. This is the method that is used with this debugger, and the methods that are depicted in the following chapters are also described with this method of reversibility in mind.

2.4.2 Reverse-continue and reverse-step algorithms

When executing the "continue" command, the typical expectation is for the program to run until encountering a breakpoint or reaching its termination point. Conversely, with "reverse-continue," the objective is to proceed in reverse until reaching either the last encountered breakpoint or returning to the program's initial state.

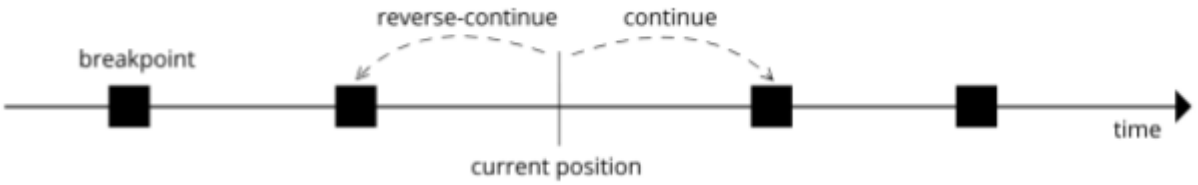

Figure 2. Executing forwards or backwards to the nearest breakpoint[8].

However, in the absence of logging, determining the program's current state becomes challenging. To address this, various tracking solutions and algorithms for reverse commands have been proposed, notably in the article "Efficient Algorithms for Bidirectional Debugging"[8]. One solution suggested involves integrating a counter mechanism into the compiled program. This mechanism increments at each program statement and compares against a predefined stop value. Upon reaching the stop value, control is transferred back to the debugger.

Utilising the "stop" and "counter" values, a reverse-step operation can be implemented by setting the stop value to one less than the counter value, restarting the program, and progressing until the counter is equal to the stop value. However, while this approach gives an idea of the program's current position, determining whether any of the current breakpoints have been encountered presents a challenge. This necessitates two executions: the first to identify which breakpoints will be encountered up to the current point, and the second to proceed to the last recorded breakpoint.

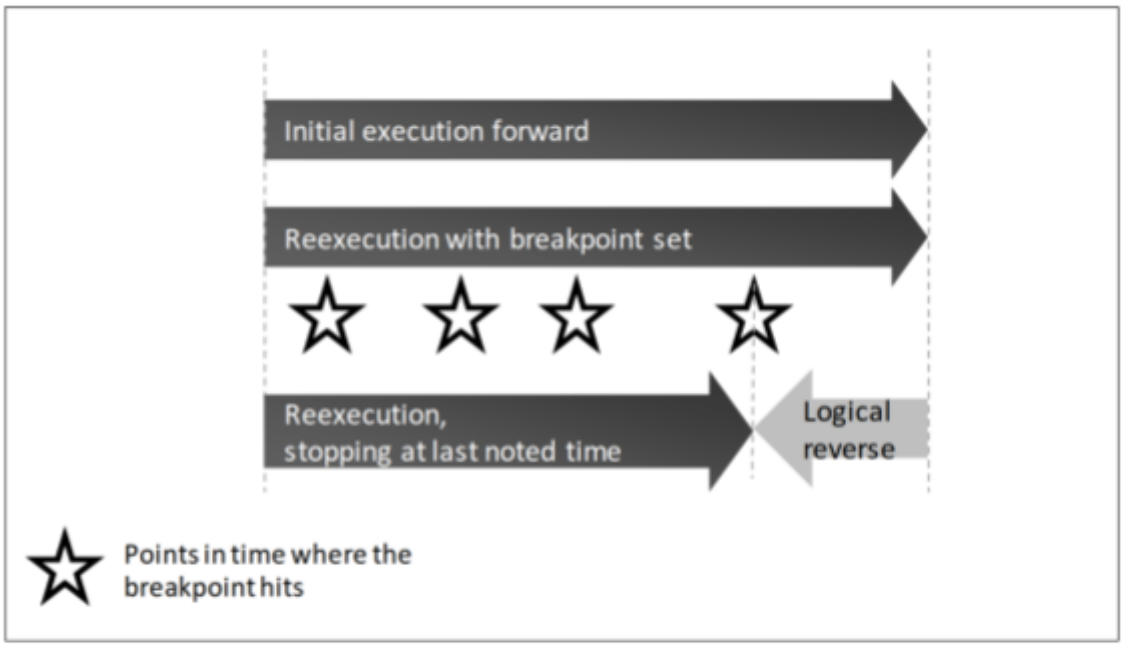

Figure 3. Reverse-continue algorithm illustration[6].

The proposed algorithm is as follows:

1. Restore the program to some past checkpoint.
2. Insert breakpoints to the locations where the user has set them.
3. Execute the program. As the program runs, it encounters the inserted breakpoints. When a breakpoint is hit, it should be reinserted, because it might occur again later on. During this replay execution, the algorithm records every breakpoint that is hit. This record essentially captures the sequence of breakpoints encountered during this forward execution.

4. Restore the program back to the chosen checkpoint. With the recorded breakpoint sequence in hand, the program executes forward. Once all the breakpoints of the first pass have been encountered, stop.

By following these steps, the algorithm effectively rewinds the program to the last breakpoint that was hit during the initial forward run. Since the program was actually not executed in reverse, it would be called a logical reverse.

2.5 Message Passing Interface

The Message Passing Interface (MPI) is a specification, which aims to standardise message-passing programs. Some of the more popular MPI implementations are OpenMPI and MPICH. These implementations most importantly provide libraries containing functions for process management (creating an environment and processes capable of communication) and message passing (sending and receiving data between processes in the created environment)[9].

In general, MPI enables programmers to distribute a program's workload across multiple processors or computers and facilitates communication inside the created environment. This approach significantly reduces execution time for problems that can be efficiently divided into independent subtasks.

Executing MPI programs starts with launching multiple processes. Each process runs a copy of the same program code but maintains its own separate memory space. Processes are distinguished by unique identifiers called ranks. These ranks are used for addressing specific processes when sending or receiving messages.

The foundation of MPI communication rests on two core operations:

- Point-to-point communication: Processes can directly send and receive messages with each other, specifying the source and destination ranks.
- Collective communication: All processes participate in operations like broadcasts (sending data from one process to all others) or reductions (combining data from all processes into a single value).

Figure 4 showcases a sample MPI program using *mpi4py*, a Python binding for MPI[10]. This example demonstrates the core concept of rank-based communication for data exchange between processes. Two processes (ranks 0 and 1) collaborate:

1. Rank 0 retrieves its rank and sends data to rank 1 using the rank information.
2. Rank 1 retrieves its rank and receives data from rank 0.

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

Figure 4. Message passing example with mpi4py[10].

2.6 Checkpointing and restoring processes

In the context of reversible computing, checkpointing a process refers to capturing a snapshot of its state at a specific point in time. This snapshot includes enough information to allow for the process to be restarted, or restored, from that point later. Restoration can occur either after the process terminates unexpectedly or even during its normal execution.

For distributed applications like those built with MPI (Message Passing Interface), there are two main approaches to checkpointing:

1. Global Checkpointing: This involves saving the state of all launched MPI processes simultaneously. This ensures a consistent global state across all processes when restoring. However, it can be complex to coordinate and may introduce performance overhead due to the need of synchronising all processes.
2. Independent Checkpointing: Checkpoints are taken of individual processes independently. This approach is simpler to implement but requires additional

mechanisms to ensure consistency when restoring, especially if processes were communicating or exchanging data at the time of the checkpoint.

One of the key challenges in implementing reversible debugging for MPI programs with distributed checkpointing is causal consistency. In concurrent programs, certain actions by one process can have consequences for other processes. Providing causal consistency means ensuring that when rewinding the execution (restoring a checkpoint), the consequences of these causal actions are also undone to maintain a valid program state[11].

For instance, in a scenario where process A sends data to process B, and B successfully receives it, we can say that the "send" operation from process A causally precedes and results in the "receive" operation on process B. When we want to restore process A to a state before the send operation, causal consistency dictates that we must also revert the state of process B to undo the "receive" operation. Otherwise, we would end up with an inconsistent state where the "receive" has no corresponding "send" that caused it.

2.7 Checkpointing software

To make checkpointing different applications easier, there are software options that can be used. Instead of having to implement checkpointing mechanisms and information collection, it is possible to incorporate software that facilitates these operations. Checkpointing solutions can exist either on the kernel, user or application level.

2.7.1 Checkpointing implementation levels

Kernel level checkpointing implies extended access to the kernel's resources and more extensive privileges. This allows the checkpointing to be transparent to the application, meaning no modifications to the application have to be made. However, one of the drawbacks of this is that it is not very portable. This means that you could not use the checkpointing implementation on another Linux distribution or kernel version that doesn't include the required features. Berkeley Lab Checkpoint/Restart for Linux (BLCR) is one example of a project that provides kernel level checkpointing and is tailored specifically for parallel applications using MPI. However, work on the project seems to have stopped, since the last listed update is from January 2013[12]. This means that the last supported kernel versions are also from that time and not widely used. There are some alternative kernel level solutions,

however, most of these suffer from the same drawback of a narrow selection of platforms or having stopped development.

Application level checkpointing implies that the software itself includes a way to save its state periodically or on command, so that it can be recovered later. Earlier versions of OpenMPI had some fault tolerance methods and research into improving them. This included features such as checkpoint and restart, message logging and network fault tolerance. However, the development work for this has come to a stop, with all the mentioned features being deprecated[13].

User level checkpointing works by using libraries to gather information about the process without having direct access to the kernel's information. For the purpose of this thesis, checkpointing MPI applications, there are a few suitable software projects that are still being actively developed. Due to the lack of good options for the kernel level checkpointing and application level checkpointing development having stopped, this method was chosen.

2.7.2 Checkpoint/Restore In Userspace

Checkpoint/Restore In Userspace (CRIU)[14] is Linux software that can checkpoint and restore containers and individual applications. During a checkpoint, data will be saved to disk, and this can be used later to restore the process to the saved state. CRIU is primarily used for facilitating process or container migration[15].

CRIU utilises ptrace to stop the work of the target process using *PTRACE_SEIZE*. It then collects available information about the process from the filesystem, such as memory maps and file descriptor information. After this, it inserts code in the target's and makes it execute it. The inserted code collects additional information about the memory of the process. This process is also done for all the child processes of the target process, enabling these to be restored too. During restoration, CRIU transforms itself into the target process. It does this by re-creating as many processes as were checkpointed and modifying the processes to match the information that was saved[16].

CRIU also allows for retaining process identifiers during restoration. Assigning process identifiers in the Linux kernel involves using the file */proc/sys/kernel/ns_last_pid*, from which the last assigned identifier is read, incremented by one for the next program, and then the file counter is incremented by one as well. Upon restoration from a checkpoint, CRIU

locks this file, writes one less than the desired identifier to it, restores the process, and then writes back the original number. However, this functionality of CRIU is also one of its most notable drawbacks, since editing the mentioned file requires administrator privileges and generally debugging tools are not expected to do so. Nevertheless, explicit root access is not necessary if the user sets the capability of *CAP_CHECKPOINT_RESTORE*, which is available after kernel version 5.9 and allows CRIU to edit the file */proc/sys/kernel/ns_last_pid*[17].

While CRIU is primarily known for its role in facilitating live migration of containers, there have been instances where it has been employed for checkpointing MPI. In these instances, it has been found that CRIU can be used to checkpoint MPI using native execution[18]. It has also been shown that MPI can be checkpointed within containers. Tests showed that checkpointing can span multiple nodes, but with a limited success rate[19].

2.7.3 Distributed MultiThreaded Checkpointing

Distributed MultiThreaded Checkpointing (DMTCP)[20] is another userspace tool for checkpointing distributed applications for MPI, OpenMP, Python and more.

DMTCP uses a subcomponent MultiThreaded Checkpointing (MTCP), which is used for checkpointing individual processes. For each created process, MTCP saves the wanted info from the process side and DMTCP is then responsible for all the necessary components that come with distributed applications, such as sockets. It uses a coordinated checkpointing method, where all processes are stopped for the checkpoint and network data in transmission is also saved, so that it is available during restoration.

Differently from CRIU, DMTCP uses a coordinator to launch the target process and follows all the created child processes. It does this by using Linux functionalities to inject DMTCP libraries into the runtime of the children processes. This way they execute the necessary DMTCP code before starting their own execution. The inserted code creates a connection between the coordinator and registers itself. This coordinator can later be used to checkpoint all the processes related to it by sending a checkpoint signal through the created connection. The process works similarly to CRIU, where memory regions, open files and their offsets, network data and other necessary information is saved.

DMTCP does not have the requirement of administrator rights and also supports distributed checkpointing spanning multiple nodes. This makes it much more suitable for use on computing clusters, since applications are often run on multiple nodes and generally administrator privileges are not granted to the users.

3. Previous work

In this chapter is given an overview of the results of the Martens' thesis. It introduces the architecture and features of the debugger, as well as some important implications they have.

3.1 Debugger architecture

Because concurrent programs involve several processes running independently, effectively debugging them requires attaching a separate debugger to each process. This ensures each process can be monitored and controlled individually.

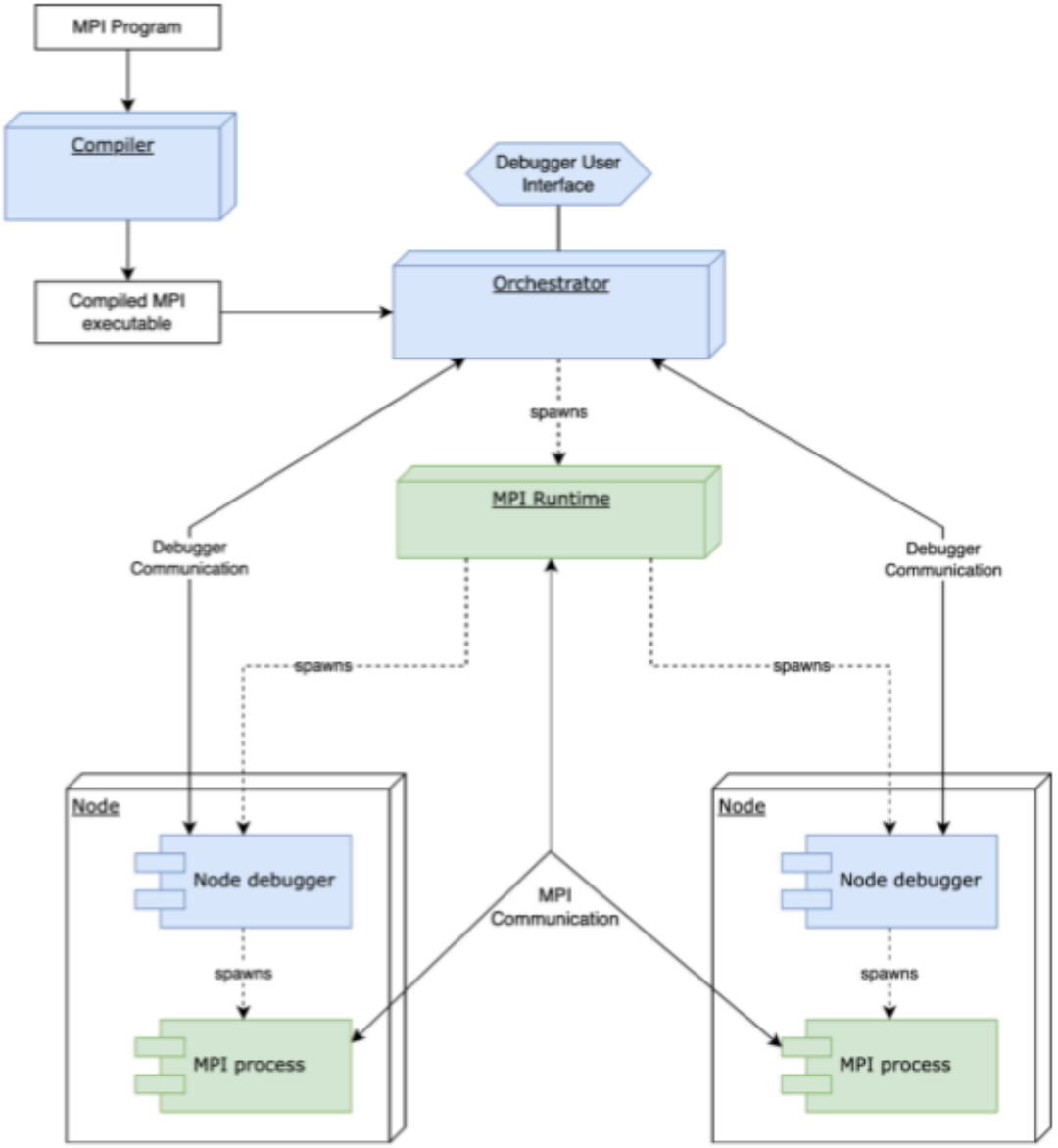

Figure 5. Martens' illustration of the debugger's architecture[1].

To manage these numerous debuggers and maintain control over the debugging process, Martens introduced an orchestrator component. This orchestrator acts as a central controller for user commands and communication to and from the debuggers.

When initiating the debugging session, the user specifies the number of processes (tasks) they wish to debug. When the debugging session begins, the orchestrator is the first component to launch. Based on the user's input, the orchestrator creates and launches the corresponding number of debugger instances. It does in the same manner that multiple MPI processes are launched. Each debugger instance is then responsible for launching a single instance of the program that the user wants to debug, and attaching itself to the launched program. These individual debuggers maintain communication with the central orchestrator using a TCP connection. This connection allows for sending and receiving instructions. When a user wants to instruct a debugger to perform an action (like pausing execution or setting a breakpoint), they do not interact directly with the individual debuggers. Instead, the user communicates their desired action to the orchestrator.

3.1 Basic commands

Martens' application included both a graphical user interface and a command line interface. The debugger was to be started from the command line and then used in combination with the GUI in the debugging process.

The CLI provided a set of commands for navigating program execution within specific MPI processes. Users could control execution flow using commands like "continue" and "step" targeted towards specific ranks (processes) within the MPI program. The CLI also allowed users to set breakpoints at desired locations in the code, enabling them to pause execution and inspect program state at those points. Additionally, the CLI offered functionalities for examining variable values within the running processes, providing insights into program behaviour.

The GUI served as a visual representation of the program's execution history. It displayed all the MPI operations (like sends and receives) that each process had executed, along with the relationships between them. This visual representation in the GUI was particularly valuable for users to locate specific locations in the program's execution flow. By analysing the displayed communication patterns, users could identify the point to which they wanted to rewind the program. The GUI further offered functionalities for initiating the restoration

process at the chosen point. Users could select the desired checkpoint provided by the GUI and trigger the restoration of the process.

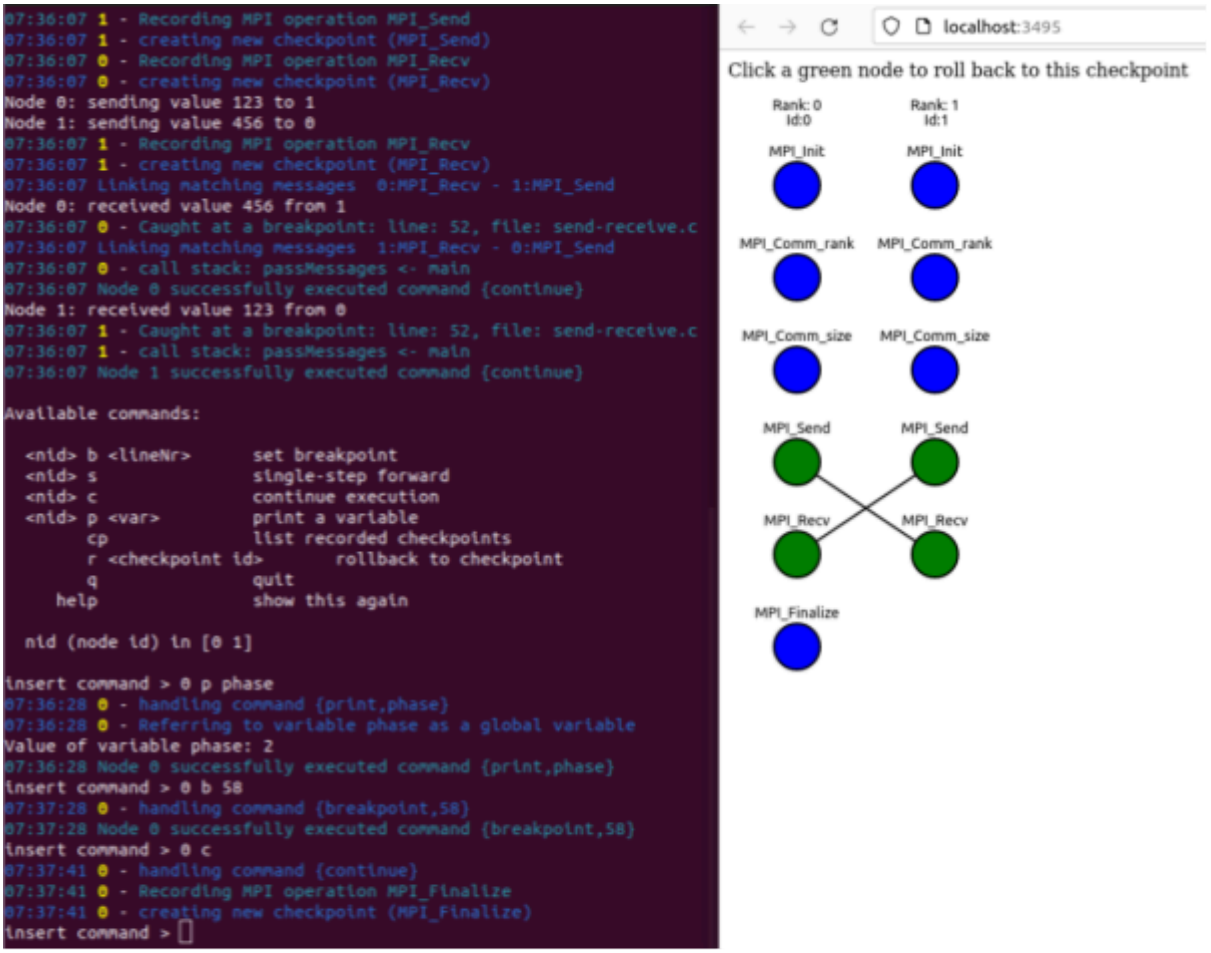

Figure 6. Side by side view of the CLI and GUI [1].

3.2 Checkpoints and the user interface

Martens gives in his thesis an overview of the existing software solutions available for checkpointing and restoring processes. While these solutions offer comprehensive functionality, he opted to develop a custom approach with specific limitations. His implementation focused solely on saving and restoring register values and related memory regions. This approach, while functional, neglects other crucial aspects of a process state, such as opened files, network connections (sockets), and their associated buffers. This lack of comprehensive state capture could lead to inconsistencies or errors upon restoration.

Martens' custom checkpointing approach used the distributed strategy, creating checkpoints of individual processes at every MPI operation. This approach allows for restoration of individual processes to a specific point in time, corresponding to the chosen MPI operation. Martens used the GUI in this restoration process (as shown in Figure 7). In the example, processes are initialised, followed by rank 0 sending a message to rank 1 (*MPI_Send*) and rank 1 receiving it (*MPI_Recv*). The communication flow then reverses, with rank 1 sending a message and rank 0 receiving.

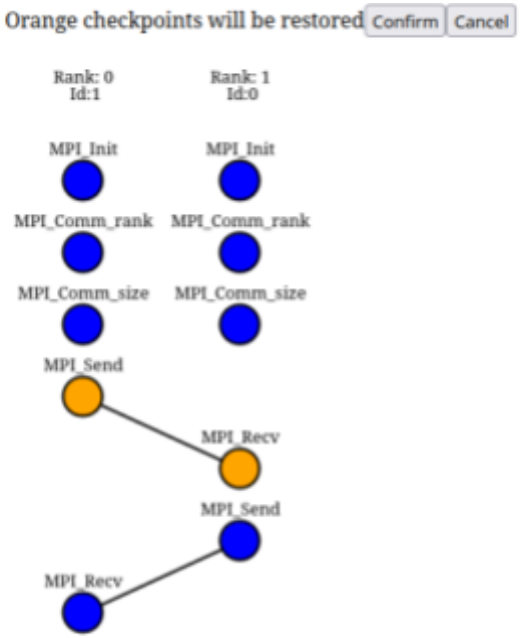

Figure 7. GUI view after having selected the MPI_Send operation to rollback to.

The causal relationship between processes becomes particularly relevant in Martens' implementation of distributed checkpointing, where individual process checkpoints are captured frequently. Restoring a single process to an arbitrary checkpoint without considering causal dependencies could easily lead to an invalid program state that would never occur during normal execution.

Figure 7 visually demonstrates the concept of causal consistency in MPI checkpointing. When the user has selected the "send" operation of rank 0 for restoration, the corresponding "receive" operation of rank 1 is also highlighted. This signifies the need to restore both processes together to maintain causal consistency and avoid program inconsistencies.

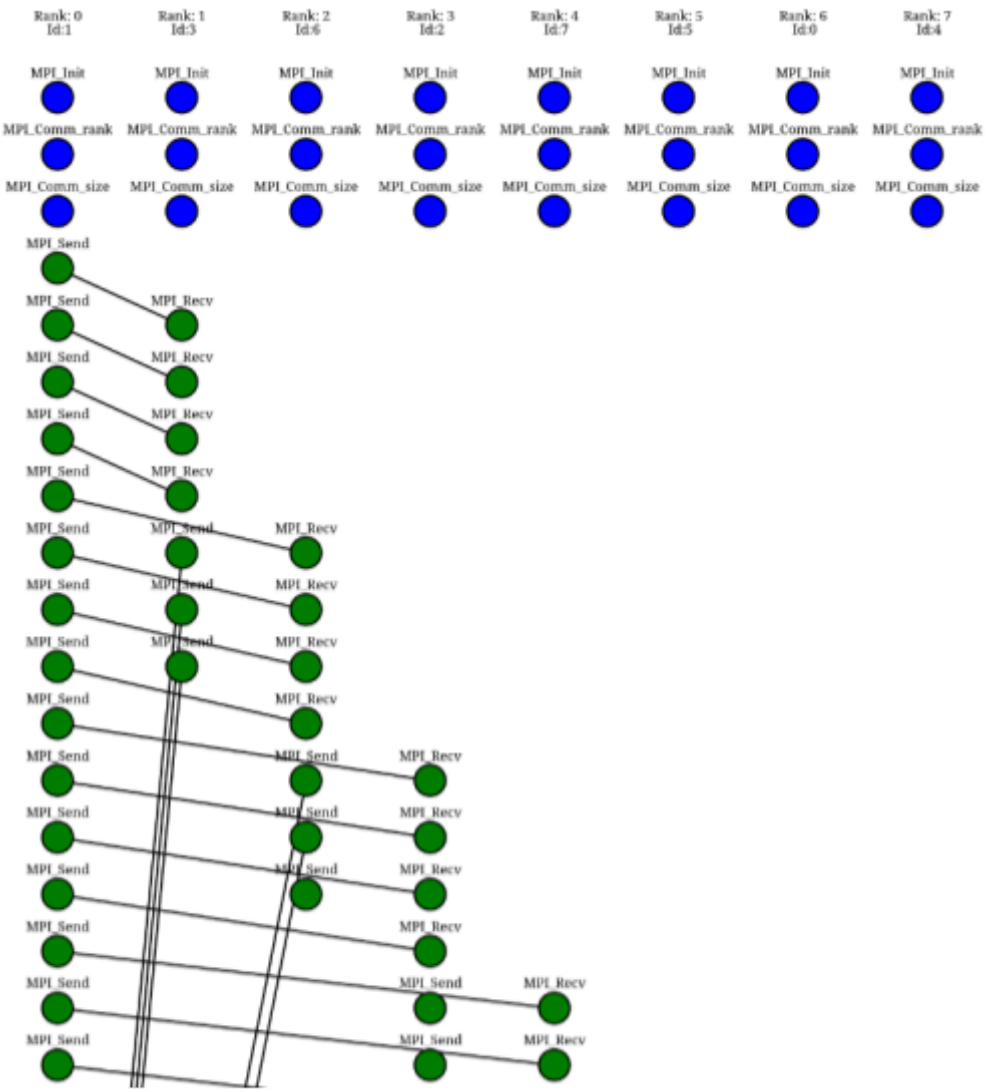

Figure 8. Visual representation that is difficult to understand.

The current utilisation of the distributed checkpointing method presents several significant drawbacks. Firstly, it places the burden on programmers to implement algorithms capable of determining all processes that must be rolled back to maintain causal consistency. This can be a non-trivial task, especially for complex MPI programs. Moreover, MPI programs often rely heavily on collective communication operations (like broadcasts or reductions) and frequent message passing. This inherent parallelism makes it difficult to achieve truly independent rollbacks by restoring a single process. In most cases, rollbacks will likely involve multiple processes, essentially becoming semi-global operations. Furthermore, checkpointing at every MPI operation can introduce significant performance overhead. The process of capturing and

storing the state of each process so often can be resource-intensive, potentially impacting the program's overall execution speed.

The GUI designed for visualising checkpoints proves valuable for understanding the communication flow and restoring smaller MPI programs. However, its effectiveness diminishes as program size increases.

Consider the example program: with just 7 tasks, the GUI can become cluttered with information, making it difficult to navigate and interpret the visualised checkpoints. For larger problems with numerous processes, the GUI's primary purpose of showcasing the communication topology and structure becomes less relevant. The sheer volume of information displayed could overwhelm the user and hinder comprehension.

Therefore, it might be worth it to consider the GUI as an optional component. While it offers valuable insights for debugging smaller programs, for large-scale MPI applications, alternative visualisation techniques or command-line interfaces might be more appropriate. This approach allows users to focus on specific aspects of the program's execution and checkpoints without being overloaded with excessive visual data.

3.3 Processor architecture

In the completed work, Martens notes that the debugger supports the x86 processor architecture at the current state. Additionally, he also mentions that in this architecture, there are 4 debugging registers available[1: 26]. However, on further inspection, it was found that these registers are instead available in the x64 architecture, and the completed program was compilable and executable in it. However, this is a useful error regarding the prospects of using the program since the x86 instruction set is outdated compared to x64, and the latter architecture is far more popular.

3.4 Compiler

One of the outputs of Martens' work was a compiler that ensured the existence of DWARF data and inserted breakpoints next to MPI functions [1: 40]. To interpret the debugging data from the compiler, a syntax analyser (parser) was also written, which divides them into sections and tries to extract certain fields of information from the sections. The compiler initially included a call to the MPI compiler. This command is made available by the installed

MPI implementation in the system. However, it is also necessary to have a compiler capable of compiling C, C++, and FORTRAN programs for installing any MPI implementation. One of the most common compilers for this purpose is found in the GCC (GNU Compiler Collection) software package. However, it was not mentioned in the thesis or in the program's GitHub documentation which MPI implementation, its version, and also GCC version the program should work with or has been tested with. Therefore, during the initial testing, although the compiler worked as intended, an error message appeared when the debugger was launched, indicating insufficient information. Upon investigating the DWARF data produced by the compiler, it was found that the data the parser was looking for was not present there. The issue was that the version of DWARF data produced depends on the compilers and their versions. The author found that the syntax analyser worked with MPICH version 3.4.3 and GCC version 9.2.0. This is due to the fact that with that combination of versions, the right version of DWARF data was generated for the parser to be able to interpret it. The compiler was instructed to generate debugging information according to the correct version of the standard.

4. Results

This chapter discusses relevant findings about the previously completed work of Ott-Kaarel Martens titled "Causally Consistent Debugger for MPI Applications" [1], and then presents the changes and additions to it. Throughout this thesis, the large language model ChatGPT[21] was used to provide assistance with phrasing and structure of paragraphs.

4.1 General changes to the debugger

The debugger initially included commands for basic navigation through the program such as continue, step and breakpoint. Additional options were added to make the experience better for the user.

The commands used for navigation and setting breakpoints had to be used with a target process. The user had to specify the index of the target process, on which the command should be executed. An additional feature was added that makes it possible to execute the same command on all processes at once. This way, instead of having to issue a continue command individually on every process, the user could do it in one command.

At first, the step command entered functions that were not present in the source code. For example, when issuing a step command before a standard print function, the code would step into it. An additional step function was added, which would pass by all the lines in the print function, and would end up on the next line of the source code. This is useful, since many of the external functions can have thousands of steps and traversing these might not be useful for the user.

The compiler was also adapted to include the counter mechanism previously described. The adaptation was made by parsing the source code that the user provided and inserting function calls before relevant statements. Before the user’s code, the function that is called and 2 global variables, counter and target, were also inserted. The function increments the counter variable and compares it to the target. If the counter is equal to the target, the program passes control back to the debugger.

4.2 Checkpointing solution

CRIU was chosen as the checkpointing software for implementing checkpoint and restore in this project. It showed to be faster and more intuitive to use than DMTCP. Furthermore, CRIU is being developed more actively and has a much easier installation procedure.

However, CRIU does not allow preserving one feature of Martens' created program, which is the reversibility of individual MPI processes. This limitation occurs due to the connection between the central MPI runtime and the processes that it starts. When creating checkpoints, UNIX socket states are also saved, but this can not be done from the perspective of only one side. Thus, when creating a checkpoint, we need to checkpoint the root MPI process, which in turn will save the states of all the spawned processes and both sides of these connections. Therefore, the program's architecture needs to be adjusted so that checkpoints are created from all processes simultaneously, instead of one process at a time. Since when launching the initial MPI process, we are starting debuggers, which start the binaries that we are debugging, we end up also checkpointing the debuggers.

Martens also highlights that both at the operating system kernel level and as a useful part of his own reversibility implementation, it's possible to maintain a consistent environment upon restoration, which is an important aspect for distributed processing tasks. CRIU allows restoring process identifiers and maintaining a consistent environment, given the limitations described earlier.

4.3 Integrating CRIU to work with the debugger

Despite its widespread adoption for process migration, there seems to be an absence of precedents for its utilisation in the manner demonstrated here – as a core component of a debugger. Consequently, integrating the software into this specific application posed considerable challenges.

The first obstacle was that CRIU uses the system call ptrace for creating checkpoints, which is also used by the debugger. ptrace establishes a relationship between the tracer and the tracee, with only one of each allowed at maximum. The CRIU documentation states that it's not possible to save programs attached to a debugger[22]. Thus, it's not possible to save the state of the program that is being debugged immediately because CRIU attempts to

checkpoint the program being debugged, but it cannot attach to it because the debugger has taken the tracer position. Therefore, when creating checkpoints, the debugger needs to release its position as the tracer, so that CRIU can create a checkpoint of it and later reattach itself to the program that is being debugged.

Additionally, there were issues with using CRIU within a Go program. To make using CRIU more convenient within Go programs, the Go software package go-criu[23] was created, allowing CRIU commands to be used as function calls. While the go-criu library successfully facilitates using CRIU functionalities within Go programs by offering function calls for checkpointing, restoring programs internally presents a challenge.

The issue stems from how CRIU distinguishes between programs launched from the command line and other processes. Command-line programs typically require a teletypewriter (TTY) for interaction. However, when a Go program checkpoints a process, the process itself is closed before restoration. This closure also terminates the associated TTY, creating a problem during restoration.

To address the missing TTY issue, a workaround consisting of the following steps is necessary:

1. Create a new teletypewriter. Instead of relying solely on go-criu function calls, the Go program needs to create a new TTY independently.
2. Execute restoration command. The program then executes the same command it would use for command-line restoration, but within the newly created TTY.
3. CRIU starts and restores. Upon executing the restoration command within the new TTY, CRIU launches, associates itself with the newly created TTY, and transforms itself into the program being restored. This allows the restored program to continue execution without a missing TTY.

Due to the architecture of the debugger including a TCP connection between the node debuggers and the orchestrator to facilitate communication, those would have to be saved also. CRIU is able to detect and keep alive TCP connections if instructed to do so. It does this by setting certain rules in the firewall that drop all the packages trying to close the connection in question. This functionality, in combination with another that is used in this project, to keep the process running instead of closing it after the checkpoint, have some when used in

combination. The first being that the firewall rules created upon checkpoint are reverted if we instruct CRIU to keep the program running, so that upon closing the program and later trying to restore it, the TCP connection is gone, and the restore fails due to not finding the rules it constructed while creating the checkpoint. A workaround for this was tested, which meant recreating and reinserting the firewall rules that CRIU created just before restoring the process from a previous point. Upon restoration, however, packet level errors occur when trying to restore from any checkpoint before the last one that was created. This is likely due to inconsistencies created by changing one side of the TCP connection. Due to this, the orchestrator will not continue the communication with the node debuggers.

Since the issue is that the previous state of the TCP connection was not able to be restored, we would have to construct a method to disconnect and reconnect upon checkpoints. However, after closing the connection from the node debugger side to the orchestrator side, there would be no way to send new instructions to the node debuggers. This means it would also be impossible to instruct them to connect back to the orchestrator. Thus, to bypass this connection issue, the checkpoint and restore method had to be implemented with a certain trick. The trick depended mostly on the robustness and quickness of CRIU. When initiating a checkpoint, we send a message to all the debuggers to disconnect from the orchestrator, disconnect ptrace, sleep for a while, and finally reconnect to the orchestrator. At the same time, we start the checkpoint process with CRIU from the orchestrator side. This way, when the debuggers reach the sleep instruction, the checkpoint from CRIU will start and the process will be saved in that state of executing the sleep instruction. Upon being restored, the processes will finish the sleep instruction and immediately try to connect to the orchestrator. Through trial and error, it was found that a sufficient time for the sleep instruction came out to be 1.2 seconds, which was successful for up to 128 tasks.

4.4 Implementation of reverse debugging commands

It was intended to implement the opposites of the common commands continue and step - reverse-continue and reverse-step. These commands were to be implemented to be targeted at individual processes and also globally.

4.4.1 Reverse-continue implementation

At first, the reverse-continue algorithm was not implemented using the algorithms described in the theoretical overview. This was in an effort to eliminate the necessity for compiling the program again and to use the resources available without the added counter.

The crucial information needed to implement a reverse-continue command is only which point in the program we are at currently, or more precisely, how to reach it. This is due to the nature of the algorithm being that we need to observe the breakpoint hits on the execution up to the current location of the program. There is however no variable that the program keeps in its memory or no way to deduce this from debugging information. For example, in the example given in figure 9, if the program was currently at the print instruction of the function *randomfunction*, you could use ptrace and debugging information to know that you are at line 2, but there would be no way to know whether it would be inside the function invocation at line 6 or at line 7.

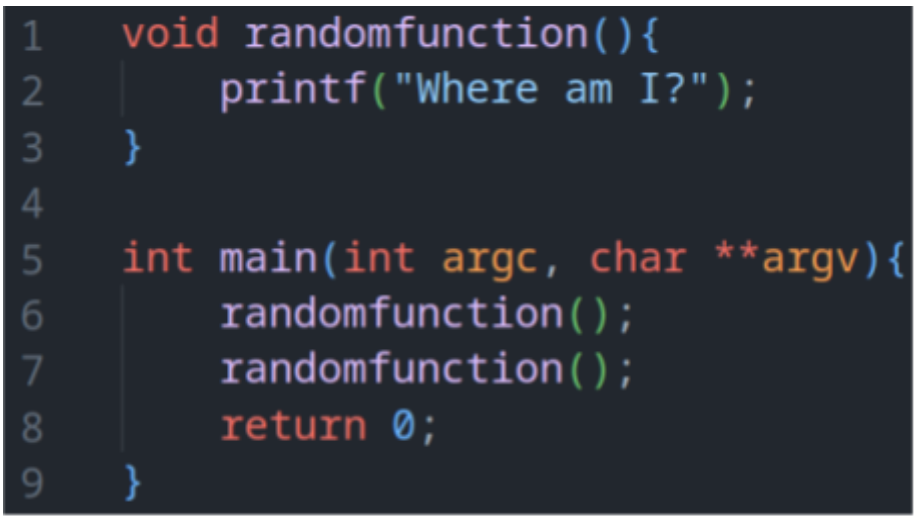

Figure 9. Code example with consecutive calls to the same function.

The initial solution was created with the goal to avoid the necessity for recompilation and extra code injection. The idea of using traps was considered. Traps would be inserted at statements to track execution flow. However, this method implies a significant performance impact and was never pursued.

Another approach explored and implemented in this project involved recording the commands issued by the user and associating them with checkpoints. By restoring the program to a previous checkpoint and replaying the commands executed since the given

checkpoint, the debugger would theoretically reach the desired state. However, analysing and combining these commands for efficient execution proved challenging. Nevertheless, this proved to be another viable solution for this task, for which the author found no precedent.

The approach of using user-executed commands to realise the reverse-continue command was sufficient, but the counter method was adopted due to the reverse-step being infeasible without an internal counter. The complete algorithm for reverse-continue is as follows:

1. Save the breakpoints currently set by the user and the value of the variable counter.
2. Select a previous checkpoint and revert to it.
3. Replace the breakpoints that are set at that checkpoint with the ones that were saved in step 1. Also, adjust the internal variable target to the value of the counter variable that was saved.
4. Continue the execution and record any encountered breakpoints. Additionally, after encountering each breakpoint, set the breakpoint again because it might also appear later on. Continue again until the counter variable's value within the program matches the adjusted target value.
5. If the debugger doesn’t encounter any breakpoints, return to step 2 and select a checkpoint further back in the execution. In the case that there are no breakpoints encountered when starting from the beginning of the program, then stop after step 6.
6. Restore from the selected checkpoint
7. Continue the execution and record any encountered breakpoints. Reset encountered breakpoints. Execution halts when the recorded sequence of breakpoints exactly matches the sequence observed during the previous execution.

Since we are using a global checkpointing method, all processes are rolled back to some step. If we want to execute reverse-continue on all processes simultaneously, we follow the algorithm for all processes. If the command is targeted at a single process, then only that one will execute the algorithm described. The other processes will save the counter value and, after checkpoint, continue to that point by modifying the target.

4.4.2 Reverse-step implementation

If the reverse-continue function could be implemented using just the commands executed by the user, for the reverse-step this is realistically not possible. Consider the simple case given in figure 10.

If the execution is stopped at line 8, it is impossible to determine whether reverse-step would end up at line 5 or at line 4. This might be conceivable if the debugger included some way of evaluating these statements during the run or by using the aforementioned trap based debugging, but this would be significantly inferior in complexity and performance to the counter based approach. Furthermore, this is a very simple example and the case gets much worse if we consider more complicated statements and recursion, for example.

```
1  int main(int argc, char **argv){
2      int a = 1;
3
4      if (a == 0) {
5          a = 1;
6      }
7
8      a = 2;
```

Figure 10. Code example for inconclusive reverse-step.

Using the counter based approach, the implementation became much simpler. When the program is initially compiled, the target variable that we insert into the program is initialised to a very high value that it will never naturally reach. For implementing reverse commands, this value is used to mark stopping points within the program. To execute a reverse-step command, we can use the following algorithm:

1. The current value of the counter variable is saved.
2. The program state is then reverted to a previous checkpoint.
3. Using ptrace, the value of the target variable is modified. This new value is set to 1 less than the previously saved value of the variable counter.
4. As the program re-executes from the checkpoint, it will now stop at the modified target variable, one instruction short of the previous state. This effectively simulates a "reverse-step."
5. The target variable is reset back to its original high value to avoid interfering with future program execution.

Similarly to reverse-continue, we have to account for the execution of reverse-step for a single process and all processes simultaneously. If we want to execute reverse-step on all

processes, we follow the algorithm for all processes. If the command is targeted at a single process, then only that one will execute the algorithm described. The other processes will follow the algorithm also, but instead of setting the new target value to 1 less than the counter, it is set to be the same as the counter. This way, only the selected process will be reversed by 1 step, the others will end up at the same state they were before.

4.5 Graphical user interface adaptation to the changes

Martens' project included a graphical user interface that displayed all the MPI operations that the MPI processes had executed. The graphical user interface was adapted for the user to get a better overview of the checkpoints that have been made.

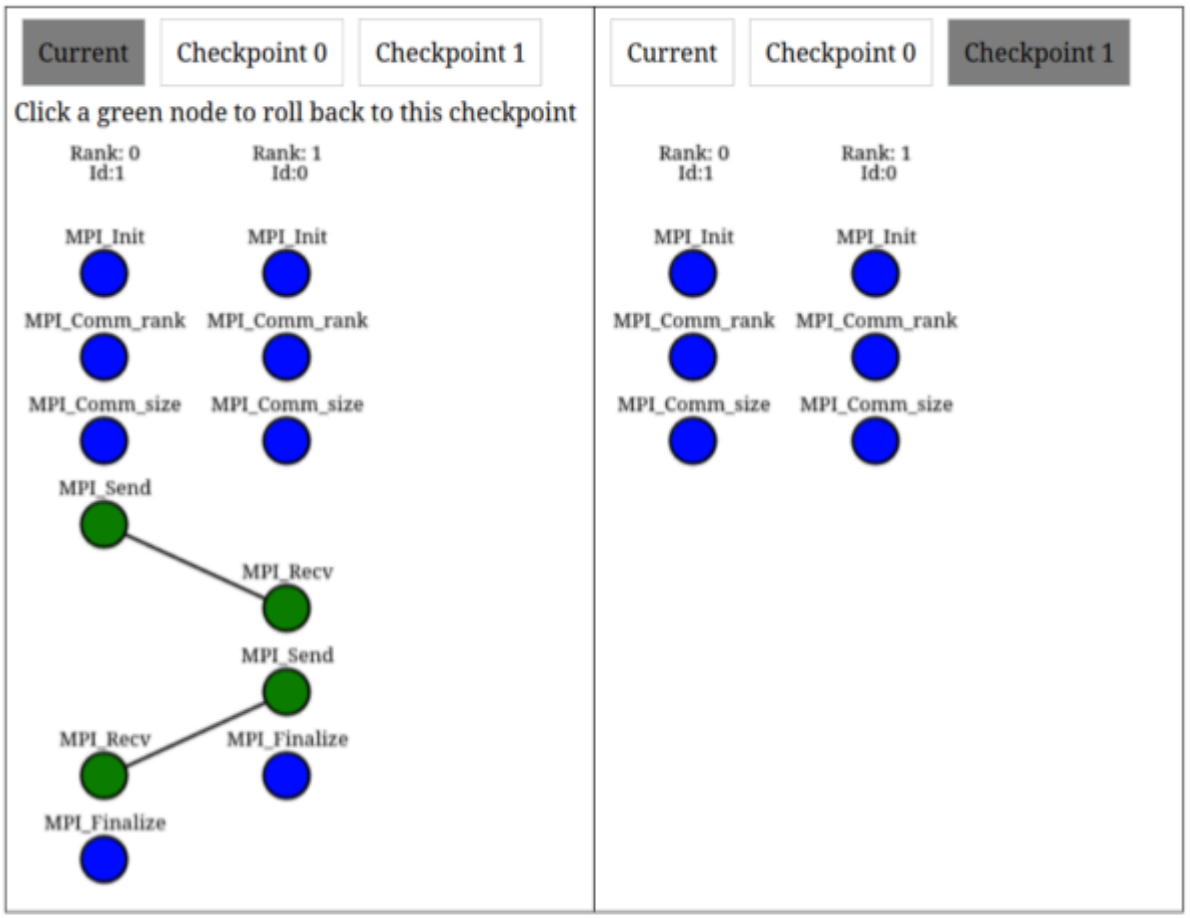

Figure 11. Side by side view of the adapted GUI. Left side displays the tab for the current state. Right side displays the state for a checkpoint previously issued by the user.

The GUI used data about MPI operations that had been executed at that point in time. Using this information, it created a mapping and displayed the relations of these operations. An

addition was made to this, which made it possible to display the MPI operation mappings at the global checkpoints that have been made, in addition to the current state.

For instance, in figure 11 are displayed 2 instances of the GUI with different states. On the left side, the display shows the current message passing history and MPI initialization. On the right side, checkpoint 1, which was issued by the user, has been selected to be displayed. Checkpoint 1 was made after the initialization of MPI has been made, but no message passing has been done. This way, the user can have a clear overview of the global checkpoints that have been issued and an easier way of navigating them. Upon issuing a global restore, the tab displaying the current state is updated. This allows the global checkpointing method to be coupled with the individual rollback method.

4.6 CRIU performance analysis

The performance of the checkpointing tool used for a reversible debugger is a very important factor. The checkpointing times and, more importantly, restoration times directly impact the time required for commands executed in reverse, and the total time spent on debugging. If the procedure for restoration would take 10 seconds, for example, then carrying out a reverse-continue would require at least 20 seconds in addition to the time spent on actually executing the program.

The performance of CRIU was examined for 3 aspects: time required for making a checkpoint, time required for restoring the checkpoint, and the size of the created checkpoint. Tests included three sample MPI programs, which can be found at LINK, executed with process sizes of 2, 4, 8, 16 and 32. For each sample, the given aspects were measured at the beginning, middle and end of the program execution and the results are visible on figure 12.

Both the checkpoint and restore durations include reconnection between the orchestrator and the debuggers. The checkpoint time averages are represented by the blue bars in the graph. The average checkpoint time increases from 1.206 seconds for 2 processes to 1.23 seconds for 16 processes and 1.36 seconds for 32 processes. Most of this time, 1.2 seconds, is made up by the sleep period in-between disconnect and reconnect.

The restoration time averages are represented by the orange bars in the graph. The average restoration time increases from 40 milliseconds for 2 processes to 160 milliseconds for 16 processes and 550 milliseconds for 32 processes. The fast times can be explained by CRIU

being very fast and the programs being restored at the very end of the sleep function. This way they immediately reconnect to the orchestrator and work can be resumed. A span of 40 to 550 milliseconds for restoration is a good result, since this only infers an overhead of roughly a second for up to 32 processes and almost none for 2 processes. Checkpoint and restore times also had minimal differences when comparing the different programs they were measured on.

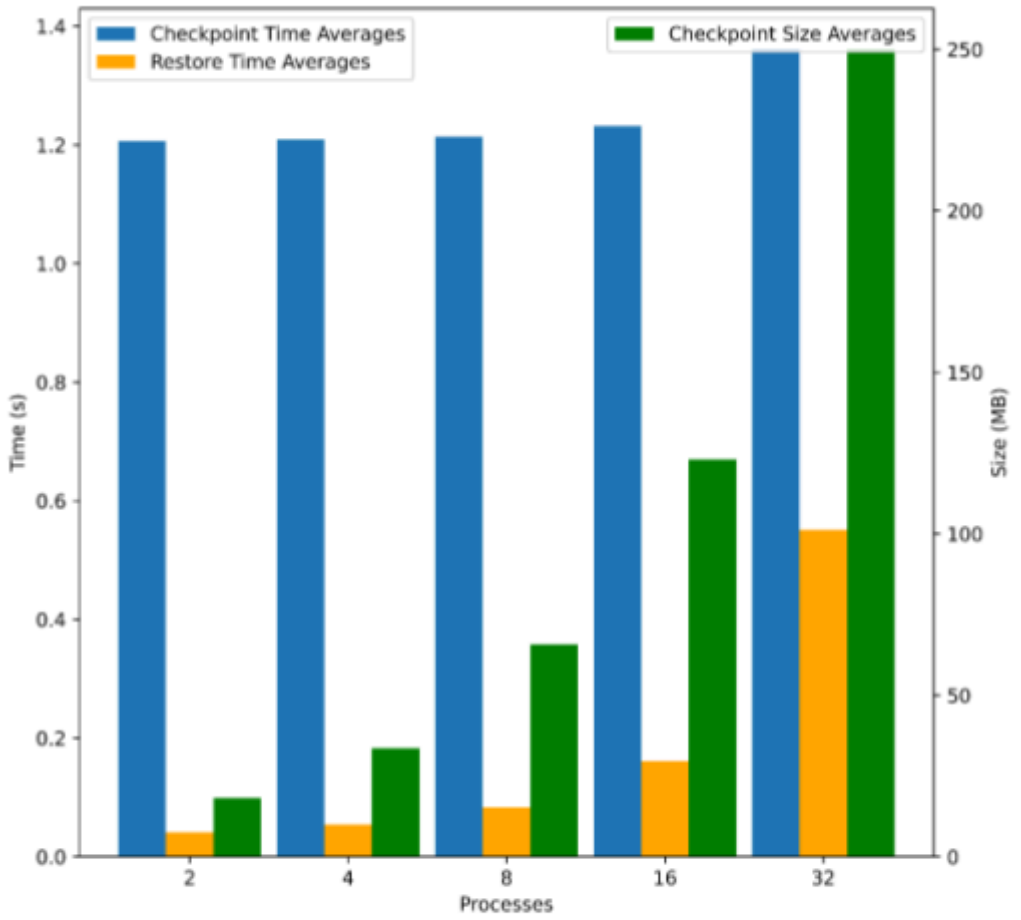

Figure 12. Graph displaying the checkpoint time averages, restore time averages and checkpoint size averages in relation to process counts.

While the checkpoint and restore durations show a remarkable increase only with 32 processes, the checkpoint sizes grow much more consistently. The checkpoint sizes correspond to the green bars in the figure and grow from an average of 15.5MB for 2 processes, 33.6MB for 4 processes to 250MB for 32 processes. As expected, the checkpoint sizes grow linearly, on average doubling for every test in which the process count was also doubled. However, the program that is being executed and the point in the program execution

that the checkpoint was made at impacted the size of the checkpoint directly. Checkpoints were significantly smaller at the start of the program and for programs allocating less memory throughout the execution. Checkpoint sizes at the beginning of the program imply that the footprint of checkpointing the debuggers is roughly 2MB per process.

5. Conclusion

The purpose of this thesis was to further develop the work done by Ott-Kaarel Martens in his thesis titled “Causally Consistent Reversible Debugger for MPI Applications”[1]. The main goals were to incorporate software providing a fully exhaustive checkpoint and restore, and to implement commands that move backwards in the program execution.

In the thesis is given a brief overview of checkpointing methods and their drawbacks with some software examples. Checkpoint and restore was incorporated using CRIU and the steps to take that were described. In addition, it includes an analysis of CRIU as a tool for checkpointing MPI applications and the suitability of the tool for the given purpose. CRIU showed to be a suitable tool with minor drawbacks for integration into debuggers.

The thesis also includes an overview of debugging. Descriptions are given of common debugging commands and their reverse execution counterparts – reverse-continue and reverse-step. Furthermore, included are descriptions of the algorithms needed to carry out these commands and modifications for use with parallel programs.

5.1 Further work

Even though the current state of the software is sufficient to be used in personal computers, it might not be suitable for shared environments such as computing clusters. To make the debugger more flexible and open up possibilities for use in other environments, the following ideas could be analysed:

- DMTCP has been used on computational clusters and does not require any additional permissions. This software should also be tested to see whether it is possible to be used in this case. It should also have the added benefit of not needing to checkpoint the debuggers attached to the executables, and might also provide the ability to make checkpoints of individual processes.
- Separating the GUI and CLI to both be standalone tools. Even though the CLI can currently be used completely on its own, both of these have their strengths, and improving the GUI would be beneficial for beginners learning MPI.
- Analysing the issue with CRIU and TCP connections. Finding a solution allowing the TCP connection to be fixed would relieve the necessity for reconnection on checkpoints and restores. This would reduce the duration of checkpointing by at least a second and restoring by a fraction of a second.
- Adding automatic incremental checkpointing. Creating checkpoints automatically with some interval would relieve the user of having to create them. It would also make commands executed in reverse faster, since there would be less code needed to re-execute after the restores.
- Improving the compiler. The compiler could be improved by allowing linking and compilation of programs consisting of multiple files of source code. Incorporation of the counter mechanism could also be improved and tested, since code injection is not easy to do without any errors.

References

[1] Martens, O. (2022). “Causally Reversible Consistent Debugger for MPI Applications”. *University of Tartu, Institute of Computer Science*. *Master’s Thesis*. URL: https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=75282 (visited on 16/04/2024).

[2] Britton, T., Jeng, L., Carver, G., Cheak, P., & Katzenellenbogen, T. (2013). Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, *229*.

[3] Rosenberg, J. B. (1996). *How debuggers work: algorithms, data structures, and architecture*. John Wiley & Sons, Inc.

[4] Michael Kerrisk. ptrace(2) — *Linux manual page*. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html (visited on 16/04/2024).

[5] Michael J. Eager. “Introduction to the DWARF Debugging Format”. URL:https://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf (visited on 16/04/2024).

[6] Engblom, J. (2012). A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference* (pp. 1-6). IEEE.

[7] Leemet, A. (2018) “Kõiketeadev silur arenduskeskkonnale Thonny”. *University of Tartu, Institute of Computer Science. Bachelor’s Thesis*. URL: https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=61853 (visited on 16/04/2024)

[8] Boothe, B. (2000). Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (pp. 299-310)

[9] *MPI: A Message-Passing Interface Standard*. 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf (visited on 05/05/2024)

[10] *mpi4py*. URL: https://mpi4py.readthedocs.io/en/stable/ (visited on 16/04/2024)

[11] Giachino, E., Lanese, I., & Mezzina, C. A. (2014). Causal-consistent reversible debugging. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 370-384). Berlin, Heidelberg: Springer Berlin Heidelberg.

[12] *Berkeley Lab Checkpoint/Restart (BLCR) for LINUX*. Berkeley Lab. URL: https://crd.lbl.gov/divisions/amcr/computer-science-amcr/class/research/past-projects/BLCR/(visited on 05/05/2024).

[13] *Fault tolerance for parallel MPI jobs*. The Open MPI project. URL: https://www.open-mpi.org/faq/?category=ft (visited on 05/05/2024).

[14] *CRIU - Checkpoint/Restore In Userspace*. URL: https://criu.org (visited on 5/05/2024).

[15] Widjajarto, A., Jacob, D. W., & Lubis, M. (2021). Live migration using checkpoint and restore in userspace (CRIU): Usage analysis of network, memory and CPU. *Bulletin of Electrical Engineering and Informatics*, *10*(2), 837-847

[16] CRIU. *Checkpoint/Restore*. URL: https://criu.org/Checkpoint/Restore (visited on 5/05/2024).

[17] CRIU. *Pid restore*. URL: https://criu.org/Pid_restore (visited on 5/05/2024).

[18] Reber, A. & Vaterlein, P. (2014). Checkpoint/restore in user-space with Open MPI. In *Proceedings of the BW-CAR Symposium on Information and Communication Systems (SInCom 2014), Germany, Villingen-Schwenningen* (pp. 50-54).

[19] Berg, G., Brattlöf, M., & Blanche, A. D. (2019). Evaluating distributed MPI checkpoint and restore using docker containers and CRIU. In *International Conference on Electrical, Communication, Electronics, Instrumentation and Computing.*

[20] Ansel, J., Arya, K., & Cooperman, G. (2009). DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE international symposium on parallel & distributed processing* (pp. 1-12). IEEE.

[21] OpenAI. (2022). ChatGPT 3.5 (April 20 version) [large language model]. https://chat.openai.com/

[22] *CRIU. What cannot be checkpointed*. URL: https://criu.org/What_cannot_be_checkpointed (visited on 5/05/2024).

[23] go-criu repository on GitHub. URL: https://github.com/checkpoint-restore/go-criu (visited on 5/05/2024).

Appendix

I. Source Code

The source code for the debugger with the changes described in this thesis is available at the following GitHub repository: *https://github.com/mihkeltiks/rev-mpi-deb*.

II. Licence

Non-exclusive licence to reproduce the thesis and make the thesis public

I, Mihkel Tiks

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis Further Work on a Causally Consistent Reversible Debugger for MPI Applications, supervised by Stefan Kuhn, PhD.
2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in points 1 and 2.
4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Mihkel Tiks

15/05/2024