UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Ott-Kaarel Martens

Causally Consistent Reversible Debugger for MPI Applications

Master's Thesis (30 ECTS)

Supervisor(s): Eero Vainikko, PhD
Stefan Hermann Kuhn, PhD

Tartu 2022

Causally Consistent Reversible Debugger for MPI Applications

Abstract:
Writing programs for parallel computation is a process significantly more difficult than programming for sequential execution. Debugger tools are of use in multiple stages of software development including implementation, analysis and maintenance. Some sophisticated debuggers offer – in complement to generic debugging commands – reversible debugging commands, providing the ability to progress backwards in the program execution in some form. MPI (Message Passing Interface) is a widely used standard for developing parallel programs. In this thesis, the implementation of a causally debugger for MPI applications offering reversible debugging commands while being capable of maintaining causal consistency is presented. The debugger utilises a distributed independent checkpointing mechanism to record the execution of the MPI application and coordinated restore mechanism to support reversible debugging of the MPI application. To the best of the author's knowledge, this is the first debugger for MPI implementing this kind of checkpointing mechanism to enable reversible debugging. The produced tool demonstrates the viability of this checkpoint-restore mechanism to enable reversible debugging for parallel computation.

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

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

Põhjuslikku järjepidevust tagav tagasipööratav siluja MPI programmidele

Lühikokkuvõte: Paralleelarvutusrakenduste arendamise protsess on tunduvalt keerukam harilike jadamisi käitatavate programmide arendamisest. Silumistööriistad on kasulikud mitmes tarkvaraarenduse etapis, sealhulgas teostamisel, analüüsis ja hoolduses. Mõned kõrgetasemelised silumisprogrammid pakuvad täiendusena üldistele silumiskäskudele ka tagasipööratavaid silumiskäske, lubades seeläbi programmi käituses tagasisuunas liikuda. MPI (ing. k. *Message Passing Interface*) on laialt levinud standard paralleelarvutusprogrammide arendamiseks. Käesolevas töös esitletakse tagasipööratavate käskudega silumistööriista MPI programmidele, mis suudab tagada programmi põhjuslikku järjepidevust silumisprotsessi vältel. Antud silumistööriist rakendab kontrollpunktide hajusa loomise ja koordineeritud taastamise mehhanismi, et salvestada MPI programmi käitust ja pakkuda seeläbi tagasipöördumise funktsionaalsust. Autorile teadaolevalt on see esimene MPI silumistööriist, mis tagasipööratavaks silumiseks sellist mehhanismi rakendab. Valminud tööriist demonstreerib selle kontrollpunktide loomise ja -taastamise

mehhanismi sobivust tagasipööratavaks silumiseks paralleelarvutuses.

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

CERCS: P170 – Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Contents

1 Introduction

1.1 Motivation

As the field of software development evolves, computational processes are becoming increasingly more distributed in their implementation. This is especially true at the high end of software development, including the field of High Performance Computing, where the distributed (and predominantly parallel) computational model is highly prevalent. MPI - the Message Passing Interface - is a well-established parallel programming paradigm that provides message passing capabilities as a means of communication for distributed computational components.

With the continuous evolution of distributed and parallel computational paradigms, the need arises for the presence of auxiliary software that can be of aid in the development, analysis and maintenance of distributed computational software. This kind of software includes debugging tools which can provide powerful capabilities for the analysis of software applications, especially for detection of software bugs. The parallel programming model brings about inherent complexities and possibilities for bugs to arise, as compared to sequential programs, since the risk of communication-related problems (such as race conditions) increases as the number of computational components increases. This heightens the need for tools such as debuggers to aid the development process.

Most debugger tools are designed for debugging sequential (or multi-threaded, but not parallel) programs. Such tools may be of some utility for debugging parallel software, but often lack the functionality for capturing bugs originating from the parallel execution model. There do exist parallel debugging tools for debugging MPI applications, however the most prevalent of which are commercial applications with closed source, as described in chapter 2.3.3.

A class of powerful debugging tools offer reversible debugging features – the possibility of moving backwards in the sequence of program execution. As demonstrated by Lanese et al. in the implementation of CauDEr - a Causal-Consistent Reversible Debugger for Erlang [1], a debugger offering reversible debugging features for a parallel message-passing computational model needs to ensure causal consistency – a mechanism should be implemented to ensure that an action can only be undone (reverted), provided that its potential causal effects are undone as well. Whereas CauDEr is an implementation for the Erlang language, there appears to be a lack of a similar tool for MPI – an open-sourced reversible debugger for capable of preserving causal consistency.

1.2 Scope and Outline of the Thesis

The scope of this thesis is to produce an experimental implementation of a debugger for MPI applications offering reversible debugging commands while being capable of maintaining causal consistency. Such a tool may potentially be of high practical

utility for developing parallel programs with MPI. To that end, the relevant concepts and significant exhibits of previous implementations of relevant software are first reviewed. Subsequently, the implemented tool is described and assessed.

The rest of the thesis is organised as follows. The second chapter gives an overview of debugging paradigms on a conceptual level; a brief introduction to reversible computation and MPI; and an overview of utilities present in the Linux environment for implementing debugging tools. The third chapter describes previous practical implementations of relevant software, including existing implementations of relevant debuggers and mechanisms of implementing reversibility, particularly tools for checkpoint-restore, which appears to be a highly practical solution for implementing features of reversibility. The fourth chapter describes the implementation of the causally consistent reversible debugger for MPI applications, an experimental tool produced as the practical scope of this thesis. The fifth chapter presents a discussion on various aspects of the implemented tool, including future work and aspects of improvement.

2 Background

This chapter gives an overview of concepts relevant for the context of the implemented debugger, including a brief introduction to debugging paradigms, reversibility in computation, MPI - the Message Passing Interface and the utilities for implementing debugging tools in the Linux environment.

2.1 Debugging software

The primary goal when debugging a computer program is to have insight into the otherwise encapsulated details of the program being run. More precisely, it means being able to observe the state of the target process at a given point in time. The state of a process usually consists of the contents of internal data (such as variables), as well as the details of which instructions are executed, in which particular order. This kind of data is generally unexposed to external observation during regular execution of a process, primarily because it is not relevant in the execution environment. Usually, the data exposed during or as a result of computation is restricted to include only the desired output. Additional information about the flow of execution or encountered undesired states may be outputted, such as error codes with corresponding messages and user-defined messages that are written to dedicated secondary output streams by logging mechanisms for later analysis. However, this kind of secondary data is generally vague and lacks the precise details about the internals of a process. Moreover, the addition of this kind of data must usually be manually specified by the programmer, thus putting an expectation on the programmer to anticipate all the possible failures.

Debugging tools attempt to solve this black-box problem of processes with a general mechanism. This means that a debugger usually aims to be as unbiased as possible about the contents of the target process, thus enabling reusability of the debugging tool across a large variety of software targets. The reusability of a debugger is usually still bounded by some constraints, such as:

- OS-level: being constrained by the tools that a operating system provides to facilitate debugging, such as the Linux *Ptrace* utility;
- Execution environment-level: relying on the programming-language-level abstractions to enable debugging actions. Major aspects for such bounds include compiled versus interpreted languages, as well as language-level virtual-machine based execution versus independent executable execution;
- Concurrency: handling (or not handling) concurrency and multi-threading of the target in some specific manner.

However, the general aim when developing a debugging tool is to keep the domain of appliance as broad as possible, in order to maximise the potential usefulness of the

tool. A debugging tool can operate in one of a selected operational mode. Either, the tool can orchestrate the "real" execution of the target in a controlled environment, or the tool simulates the execution on some abstract level that is more shallow than the native execution of the program.

2.1.1 Controlled execution

In the traditional sense, a debugger is considered to be a tool that executes the target program under controlled conditions. The manner of execution for the target is similar to the normal runtime, presenting the target with an environment that is *close-to-reality*, although it may be also desirable to isolate the execution environment, to present the target with an environment with a well-defined state. The purpose of such isolation is to reduce the non-deterministic effects on the target originating from the runtime environment.

Sources of non-determinism

Generally, non-determinism in software originates from external sources that the program interacts with. For example, a major source of non-determinism in modern software is communication with other components over the network. This is because the behaviour of the devices that are communicated with can change over time for whatever reason. In addition, the network communication itself with its dynamic nature and susceptibility to physical faults can exhibit failures when transmitting messages. Another source of non-determinism originates from the program's interfacing with other aspects of the environment of the program, such as the operating system at runtime. It is common for software to make calls to the operating system for carrying out certain operations, such as reading and writing to files. Such calls often mean interfacing with storage devices that can also exhibit physical faults as well as communication faults. Even though the operating system abstracts the details of such devices from processes (writing a file for example to a hard disk or a usb device is an identical action from processes point of view), the operation can still fail, or yield different results during different times of execution. Usually, the running process does not have exclusive access to external sources (such as files on the OS filesystem) and additionally, the external components are not guaranteed to behave in an idempotent manner. To some degree, these types of debuggers that execute the target fall under the umbrella of dynamic analysis tools. However, dynamic analysis is a category much broader that also includes tools for software testing and profiling. Some dynamic analysis tools offer a broad range of functionalities beyond generic debugging. The Valgrind project [2] and its family of tool implementations is a good example of that. This tool can be used for debugging purposes (with the *memcheck* utility for example), as well as for profiling and detection of memory leaks.

A potential pitfall of these types of debuggers is the fact that the debugger environment itself may produce unwanted side-effects in the execution of the target process. According to Engblom, "Attaching a debugger can disturb the timing of a program, easily masking errors (so called Heisenbugs, where the act of observation changes the system to hide the bug)"[3]. Thus, execution-based debugging tools should be designed with care to minimise such unwanted side-effects.

2.1.2 Simulation

Another mode of operation for a debugging tool is simulating the execution of the target program instead of actual execution. An example for this kind of tool is the Harmony language project [4]. The Harmony language is specifically designed for "model-checking" a program, to identify sources of issues related to concurrency. The emphasis for this methodology is on the simulation aspect - the program is not executed in the way a regular program is, rather the external tools traverses the source code and models the execution in an abstract representation. This kind of approach has its benefits. First, the abstract representation of the runtime can be designed in the specific manner that allows observing some distinct qualities of the target (such as observation of race conditions with the Harmony language example). Secondly, the simulation can be executed in a much more resource-efficient manner compared to regular execution. For example, if our goal is to track side-effects or memory access, we might not need to know the result values of computationally expensive operations done in the program. Rather, we can afford to estimate or even ignore the result values of such operations, thus abstaining from excess computation. The main drawback of using simulation mechanisms for debugging purposes is the lack of detail of the simulation. It is difficult to model all possible software faults without the simulation approximating the granularity of the real runtime. Hence, most debugger tools (such as the ones described in chapter 3.2) are implemented through orchestration of the actual process.

2.2 Reversibility

Reversibility in computation refers to the quality of a computational process of being time-reversible, meaning that the dynamics of the process remain well-defined when the sequence of time-states is reversed. In other words, a computational process that can be "executed" backwards has the quality of being reversible. The idea of reversibility in computation originates from studies on the thermodynamic properties of computational processes by Robert Landauer in 1961 [5]. Landauer noted that irreversible computation must always exert a minimal amount of heat, as some information is lost. Therefore, logically reversible operations could in principle be carried out without any dissipation of heat. Since then, there has been considerable research and practical work done on multiple levels of computation to introduce the concept of reversibility into computational

processes. Notable cases include theoretical efforts, such as the design of reversible Turing machines [6] and conservative logic [7], as well as implementations on software level, like the formal approach to an undo operator [8], and the design and implementation of a time-reversible programming language Janus [9].

2.2.1 Reversibility of Computational Operations

In order to revert a computational operation, the effects resulting from carrying out the operation must be undone. This can be hard to achieve in both logical and practical sense. On the atomic level, not all computational operations are logically reversible. As illustrated on Figure 1, a logical NOT operator is reversible, as each possible output value Y has a single-valued input value A. This means we can deterministically deduce the input value of the operation from the output value. However, the logical XOR operator is not reversible, as we can deduce many possible combinations of input values X = (A, B) from the possible output values Y. This means that applying the XOR operation results in information loss. In reversible logic, a notable operator is the Toffoli gate (also known as the controlled-controlled-not or ccnot gate) proposed by Tomasso Toffoli. This gate has 3 input and output bits and functions as follows – if the first two input bits are set to 1, the third bit will be inverted, otherwise all 3 bits remain the same. [10]

The Toffoli gate is universal, meaning that for any Boolean function $f(x_1, x_2, ..., x_n)$, there is a circuit consisting of Toffoli gates that takes $x_1, x_2, ..., x_n$ and some extra bits set to 0 or 1 as input, and outputs $x_1, x_2, ..., x_n, f(x_1, x_2, ..., x_n)$, and some extra bits called referred to as garbage. This means that the Toffoli gate can be used to construct logic circuits that will perform any desired Boolean function computation, while maintaining the property of reversibility. The fundamental principle of the Toffoli gate is the foundation of enabling universal reversible computing – in addition to "useful output", we must also store some extra data related to the computational process itself, in order to be able to revert the operation. The first 2 bits of the Toffoli gate output values are irrelevant for the result, as they were known before executing the operation. These bits are only relevant for reverting the operation. Nevertheless, the cost of storing this extra information for non-reversible operations might be outweighed by the benefit of not having to execute the whole computational process from the start. [10]

2.2.2 Checkpointing

In addition to the mechanism of reverting computations for the goal of *moving to a previous state* in a computational process, a similar mechanism for achieving this is checkpointing. In this approach, intermediary states of the computational process are stored, so that the system can be reverted to these states when needed. As these intermediary states can hold a lot of information, this kind of checkpointing can be carried out at periodic intervals rather than after each individual operation. With periodic

NOT gate	
INPUT	OUTPUT
A	Y=¬A
0	1
1	0

XOR gate		
INPUT		OUTPUT
A	B	Y=A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Toffoli (CCNOT) gate					
INPUT			OUTPUT		
A	B	C	A'	B'	C'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Figure 1. Truth tables of NOT, XOR and Toffoli logic gates.

checkpointing, reverting the computational process to an arbitrary previous state t_m can be carried out by:

1. reverting the process to the latest checkpoint $C(t_n)$ where $t_n \leq t_m$;
2. continuing the execution forward from the checkpoint until the time-state t_m is reached.

Checkpointing implementation levels

Checkpointing may be implemented on different levels of computational abstraction, such as the operating-system level and user-level. For system-level checkpointing, the user-space application may be oblivious to the existence of the checkpointing mechanism, which can be convenient from user perspective. System-level checkpointing also offers portability, as the checkpointing solution is agnostic to the specifics of the user-level processes it is being applied on. The downside of system-level checkpointing is the lack of granular configuration of checkpoints – as the target process's state is homogenous memory contents from the os-level perspective, then the whole state is included in the checkpoint by default, thus potentially redundant data is recorded, resulting in a unnecessarily large checkpoint sizes. The benefit of user-level checkpointing is the ability to granularly define the timing and scope of the checkpoints. The main downside however is the requirement to manually define the checkpointing configuration for each application. [11]

Distributed checkpointing

For parallel systems, checkpointing may be implemented globally (meaning the state of the whole distributed computational system is recorded at once) or in a distributed manner, by having the subcomponents (such as individual processes or process groups) record their state periodically, either with globally coordinated or independently timed intervals. Global checkpointing can be easier to implement, but results in checkpoints with a large memory footprint. Global checkpointing also requires the whole system to be rolled back at once, even if the failure happens on a subcomponent-level and a partial rollback might suffice as a mitigation. Distributed synchronised checkpointing requires care for synchronising the processes, and computation/communication may have to be halted globally in order to create a checkpoint. For coordinated distributed checkpointing, the footprint of checkpoints is small and there is no synchronisation overhead, but establishing a causally consistent checkpoint to roll back to (with methods such as the cascaded rollback) can be difficult. [12]

Although checkpointing is not the execution of the computational operations in reverse, it can serve a similar goal in practice of *moving to a previous state*. It is widely used for failure mitigation, as described in chapter 3.1, and the debugger implementation described in chapter 4 implements checkpointing to enable reverse debugging. Which implementation of reversibility is more suitable, depends on the exact use-case. Checkpointing can be suitable for long-running computational processes carried out on a relatively small dataset, as the cost of storing the checkpoints can outweigh the cost of executing a large number of operations in reverse. Additionally, in a practical sense, implementing a checkpointing-restore mechanism may be much easier than implementing the reverse execution of individual instructions.

2.2.3 Applying Reversibility

The possible motivations for introducing the concept of reversibility into a software system are manifold, and the list of applications is definitely expanding as this field develops. There are domains of application where reversibility has already been shown to yield practical benefit.

A prevalent domain in which reversible computation is a core concept is quantum computing, as reversible logic is a prerequisite for implementing quantum computation [13]. Another endeavour that has benefitted from the introduction of the concept of reversibility is enabling fault-tolerance in computational processes. Long-running computational processes are susceptible to failures of different kinds, such as the ones originating from faults in the computational hardware itself. Fault-tolerance is especially relevant for parallel computational systems with a large number of computational components, as they are by their nature more susceptible to run-time failures than their serial

counterparts. This is due to the fact that the probability of at least a single component failing increases rapidly as we add more components to the system. For example, if a single processor has the manufacturer's guarantee of a Mean Time Between Failures of 10 years, then a system with 10^4 processors will exhibit a failure every 9 minutes on average. A common approach for failure mitigation (especially for concurrent systems) is the checkpointing mechanism, combined with a rollback operation. Periodically, the system state is stored as a snapshot of the processes, and when a failure occurs, the system can be rolled back to the last recorded failure-free state. [11]

Reversibility in debugging

In the context of debugging, reversibility refers to allowing the user to traverse backwards in the execution of the target process to a previously encountered state. A common debugging functionality is to step forward in the execution flow – a debugger with support for reversibility also allows stepping backwards in some manner, thus enabling bidirectional debugging.

The use cases for such functionality are manifold. Generally, it provides more flexibility to the user who is exploring the (possibly abnormal) behaviour of the target program. More specifically, reversibility can be key for observing some types of faults, such as intermittent faults – faults that occur only on a fraction of executions, and originate from timing-related problems, such as race conditions. Traditional cyclic debugging (see Figure 2) is impractical for observing faults that appear only on some specific timing scenarios. A debugger with support for bidirectional debugging can be configured, for instance, to pause the execution when the failure happens, after which the user can traverse backwards in the execution to locate the root cause of the failure. [3]

Another debugging approach which is useful for debugging intermittent faults is the record-replay approach. This method relies on obtaining a recording of the execution that can be replayed deterministically. The recording can be implemented in a multitude of ways. The straightforward approach is to record all the instructions and replay them one-by-one. Another approach is to record all the sources of non-determinism in the program (such as the communication with the external environment) that can be replayed deterministically in the future replays. This kind of record-replay approach is implemented in the rr debugger, which is examined more closely in section 3.2.4. [14]

The risk of intermittent failures appearing is prevalent in applications with a degree of concurrency, such as multi-threaded or distributed programs. The communication between the distributed computational components can vary in latency, thus potentially yielding a different global order of events on each execution. This applies to programs which operate with the MPI paradigm that is addressed in this thesis, as communication between distributed components is the key concept in MPI. Hence applying debugging paradigms that enable replicability of intermittent failures has great potential in this context.

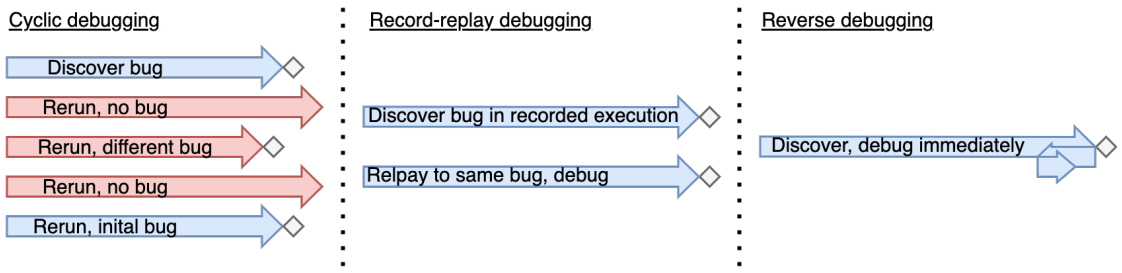

Figure 2. Different debugging strategies[3].

Achieving reverse debugging

Most software and respective runtime environments are implemented with paradigms that only enable forward execution of instructions. Even time-reversible programming languages like Janus do not enable to dynamically revert the order of execution in an arbitrary manner [9]. Hence, the debugger tool must be able to introduce this support of reversibility to a target in some external manner.

The debugger can employ some specific technique of reversibility to offer reverse/bidirectional debugging functionalities. For example, a debugger might employ a **recording**-based solution – that is to record the effects of each executed machine instruction, to produce an event log that is detailed enough for enabling the reconstruction of past states of the program execution. The reverse debugging commands such as reverse-step make use of recorded history to undo the effects of past actions to revert the program into a previous state. The gdb debugger described in section 3.2.1 implements this mechanism. The perceived benefit of this mechanism is great flexibility, as the user can step backwards one instruction at a time. The downside is the inevitable slowdown of the forward execution of the target, as each executed instruction n of the target requires executing m instructions to determine and record the effects of instruction n. Moreover, the recording must be implemented for each instruction offered by the processor's instruction set (which can vary across processor families), if implemented at instruction-level. Supporting all instruction sets is a considerable effort, and failing to do so results in incomplete functionality. The efficiency of such per-instruction reversibility implementation can be improved by making use of logical reversibility of the instructions. For instructions that are reversible by nature, the debugger could deduct the needed actions to undo a particular instruction from the output state of the instruction (i.e. the current state). This would eliminate the need to store the effects of reversible operations as they are executed forwards, thus yielding a smaller recorded data volume and potentially faster execution speed. For irreversible operations, there is still the need to store some data about the input of the instruction (analogously to the first 2 output bits of the Toffoli gate described

in section 2.2.1).

An alternative approach a debugger might utilise to enable reverse debugging is to employ a **checkpointing**-based mechanism to periodically record the state of the target process. For example, the debugger might record the state of the target periodically, and offer the functionality to restore the state of the target process to these recorded checkpoints. Such checkpoints may either be live - preserved as a live copy of the original process (obtained for example via the fork system call in Linux) or stored to the disk as files. The creation of checkpoints may be triggered automatically (such as after every n instructions, after each time-interval t, or upon specific scenarios, such as a system call made by the target process). This approach has benefits over the per-instruction approach, such as the relatively low computational overhead on forward execution. A potential drawback is large resource footprint of the checkpoints, if no optimisations are made or if checkpointing is done very frequently. Typically, the creation of live checkpoints is fast, but maintaining a large set of checkpoints simultaneously consumes system resources. Checkpoints that are saved on disk may take longer to construct and deserialise, but do not consume computational resources other than disk space when idle. Another potential disadvantage of checkpointing-based solutions is the loss of granular operations, as the user only revert n operations at minimum. In practice, this kind of granularity of operations might be sufficient, depending on the exact use cases. Moreover, the single-step-reverse operation can still be achieved by reverting n instructions and then executing forward for n-1 operations. This kind of approach of reversible debugging based on checkpointing is implemented by the debugger described in chapter 4 of this thesis.

Causally consistent reversible debugging

Causal consistency is a consistency model in the domain of distributed computing. It arises from the observation that distributed computing systems usually lack a globally unique notion of time. Rather, the agents participating in the distributed communication network each follow their unique notion of time. This observation means that it is difficult if not impossible to define or observe a total ordering for events occurring in separate components of the system. For such systems, we can however occasionally identify a "happened before" relation between two events. Events happening on the single process can be ordered logically in a total order. Additionally, a message sending event must happen before the receiving of this particular message, meaning a "happened before" relation can be established between sending and receiving events. However, two events happening on different processes with no interaction cannot be ordered in this manner. Events that cannot be causally ordered are considered to be concurrent. Hence, only a partial ordering of events can be established globally. Such partial ordering can be represented as vector clocks. In a distributed system with n processes which rely on message-passing as means for communication, for each process p_i, a vector

$V_C = (c_{P_0}, ..., c_{P_{n-1}})$ is maintained to represent the known order of events by this process. The vector clocks are synchronised upon a messaging event between two processes. Figure 3 shows an example of using vector clocks to maintain the history of known events. The partial ordering potentially has multiple possible extensions to a total ordering, which by their nature are arbitrary and may differ from the "real" ordering. [15]

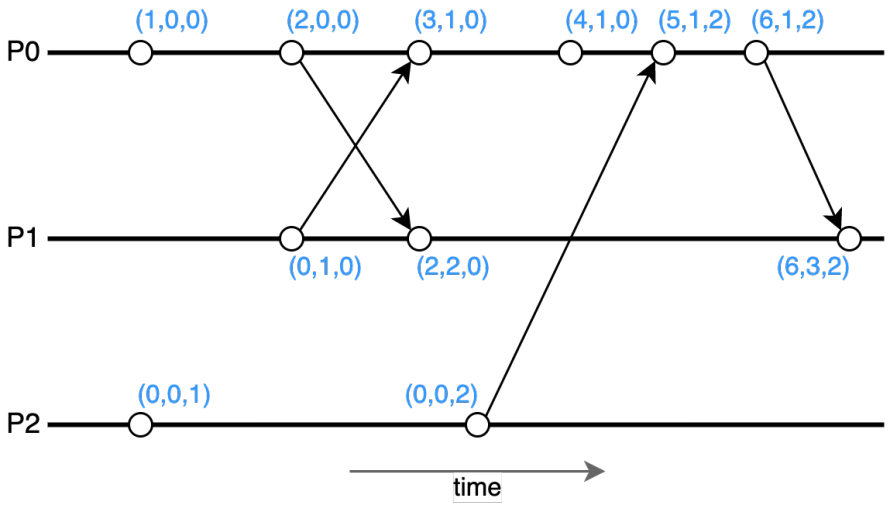

Figure 3. Representing a partial ordering of events with vector clocks.

When developing a debugger for a distributed system and introducing the features of reversible debugging, the concept of causal consistency becomes relevant. In the message-passing paradigm, the sub-processes of a distributed computing process are usually conceptualised as agents that have the ability to communicate to each other and may be occasionally synchronised with other processes, but are otherwise independent in their execution. For forward execution, the causal consistency cannot be violated, as a message receiving event cannot happen before the respective message is sent (unless the communication implementation is seriously faulty). However for backwards execution, the reverting of a message send event without reverting the corresponding message receive events would violate causal consistency on the system scale. Violating causal consistency in this manner would result in a globally inconsistent state in the system. This incorrect state is misinformative, as it depicts a scenario which cannot occur during regular execution, and additionally will result in potential failure if the system is executed forwards from that point in time.

To prevent such violation of causal consistency, a debugger offering distributed reverse debugging must ensure that when an action is undone, the potential causal effects of the action need to be undone as well. To this end, the debugger should implement some synchronisation mechanism to maintain a causally consistent global state. This implies

either the existence of a global synchronising agent, or the collaboration of the distributed nodes themselves to maintain this desired state. Such mechanisms are implemented in the CauDEr debugger described in chapter 3.2.2 and in the implementation of the debugger described in chapter 4.

2.3 MPI – the Message Passing Interface

The Message-Passing Interface (MPI) is a specification that defines a set of library routines for achieving inter-process communication. As the name suggests, the communication operations are defined with the message-passing paradigm. MPI is widely used in parallel computing models, and is considered the *de-facto* standard for distributed-memory parallel computing model communication [16]. Although introduced decades ago, the MPI specification has evolved through many iterations and remains highly relevant in the era of *Exascale computing* [17].

The terminology used in this thesis for MPI systems is as follows. An *MPI program* is an arbitrary computer program designed to be run with an MPI runtime. An example MPI program can be found in figure 5. An *MPI runtime* is a tool that takes as its input a compiled MPI program, along with the amount of processes to be spawned n, and spawns an *MPI job* - a distributed computational process consisting of n *MPI processes*. Each MPI process is a software process instance executing the specified MPI program. The MPI processes can communicate among each other via message-passing to carry out a distributed computational process. The MPI job is considered finished if all of the MPI processes in this job have finished their execution.

The MPI model implies that memory is not shared between processes, and data is moved from the address space of one process to that of another process through cooperative operations on the associated processes. The main constructs defined by MPI are for two-party and multiparty communication, but the specification has been extended over the four major releases to support other parallel programming utilities such as dynamic process creation, parallel I/O and remote memory access.[18]

As the MPI specification does not define the implementation details of the subroutines, then it is the responsibility of the MPI runtime to implement the communication. Different runtimes implement varying communication methods to carry out the communication, such as communication over network sockets (allowing great flexibility in terms of process distribution), process-to-process communication (on the same machine, via some mechanism offered by the current architecture) or even shared memory (provided that processes have access to shared memory). This flexibility in communication implementation allows adapting the communication mechanism to the particularities of the used system, especially considering the hardware. For example, a multi-core processor with access to shared memory may use this feature to implement communication that exceeds the efficiency of networked communication. The programmer developing MPI programs need not be concerned with the mechanism of the communication, and can

```
MPI_Send(void *buff,
         int count,
         MPI_Datatype type,
         int dest,
         int tag,
         int comm)

MPI_Recv(void *buff,
         int count,
         MPI_Datatype type,
         int source,
         int tag,
         int comm,
         MPI_Status *status)
```

Figure 4. Signature of MPI Send and Receive operations.

rather design the communication in an abstract sense.

The formal MPI specification defines the syntax and semantics for the MPI subroutines, without defining the implementation details. A selection of programming languages such as C, C++ and FORTRAN have implementations of the MPI specification. Performant low-level languages are generally used, as the primary goal of parallel computing is maximising computational efficiency. Bindings for higher level languages exist as well though, for example Python language implementation *mpi4py* [19]. In this thesis, the C language implementation is attended to primarily.

In the following subchapters, an overview of MPI paradigms and mechanics is given, to the degree that it is relevant to the debugger described in chapter 4.

2.3.1 Primary MPI Operations

Figure 4 shows the signature of the two base MPI subroutines *MPI_Send* and *MPI_Receive* in the C language syntax. In order to transmit a message from process A to process B, the processes must call *MPI_Send* and *MPI_Receive* respectively. The MPI communication methods have both blocking and nonblocking implementations. Figure 4 displays the signatures for the blocking variants.

The message transmission occurs over a communication channel specified as the comm argument. By default, all initialised MPI processes belong to the *MPI_COMM_WORLD* communication channel that is created on MPI initialisation, but it is possible to create custom communication channels such as process sub-groups to establish a multi-level communication hierarchy.

In order to send or receive a message, the rank of the respective receiving or sending party must be specified. The rank is an integer value uniquely identifying a process

participating in the communication channel. The rank of a process can be queried by calling *MPI_Comm_rank*. It is possible to receive messages from any source by specifying *MPI_ANY_SOURCE* as the source. Specifying the involved processes is not needed for multiparty communication methods that involve the whole process group (such as *MPI_Bcast* or *MPI_Gather*). [18]

The MPI specification also describes more complex communication methods, including one-to-many (e.g. *MPI_Bcast*, *MPI_Scatter*), many-to-one (e.g. *MPI_Gather*, *MPI_Reduce*) and all-to-all (e.g. *MPI_Alltoall*, *MPI_Allreduce*) communication. For synchronising processes, *MPI_Barrier* can be used that blocks on all processes until each process has reached this routine. [18]

2.3.2 Control Flow of MPI Processes

When an MPI program is executed with an MPI runtime, the number of processes to be spawned n is specified. Upon this, n processes are spawned that each start executing the provided MPI program. Figure 5 displays a sample MPI program in the C language syntax that is intended to be run with 2 processes. The first and last calls *MPI_Init* and *MPI_Finalize* block on all processes until each participating process has made the call. *MPI_Comm_rank* is used to retrieve the unique identifier for each process , that is to be used in subsequent communication. If started with 2 processes, ids 0 and 1 are respectively assigned to the spawned processes. *MPI_Send* is a blocking call to send the value of `send_value` to the process id identified by `other_process_rank`. Subsequently, a call to *MPI_Recv* is made by each process to receive one message designated to this process. Both Send and Receive are executed on the default `MPI_COMM_WORLD` communicator. The Receive command specifies `MPI_STATUS_IGNORE` as the status parameter, but may specify a status object to capture the origin of the message, when accepting messages with `source=MPI_ANY_SOURCE`. [18]

2.3.3 Debugging MPI Programs

The shared source code MPI programming model, referred to as "many voices in one head" easily lends itself to the users making errors when developing these programs. In addition, even with logically correct implementation, bugs originating from communication-timing-related conditions can arise. Hence, the need for debugging tools is prevalent. However, debugging a distributed computational process is not as straightforward as debugging serial programs. As put by Frank Nielsen: "Programming parallel algorithms is far more delicate than programming sequential algorithms. And so is debugging parallel programs too"[20]. In the following, different debugging strategies are analysed for their viability to debug MPI programs.

```c
#include <mpi.h>

int process_rank;
int other_process_rank;

int send_value;
int receive_value;

int main(int argc, char **argv) {
    MPI_Init(NULL, NULL);

    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

    if (process_rank == 0) {
        other_process_rank = 1;
        send_value = 123;
    } else {
        other_process_rank = 0;
        send_value = 456;
    }

    MPI_Send(&send_value, 1, MPI_INT, other_process_rank, 0,
        MPI_COMM_WORLD);

    MPI_Recv(&receive_value, 1, MPI_INT, other_process_rank, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    MPI_Finalize();
}
```

Figure 5. Example MPI program.

Serial debugging

The most primitive approach to debugging an MPI program would be to try and run the program as a serial process and attach a generic serial debugger to it. Although MPI programs can be executed as regular serial programs, the program will result in MPI failure, if the program assumes more than one MPI process is spawned. Hence, any purposeful MPI program cannot be executed and debugged in this manner.

Serial debugger per process

Another approach to debug MPI programs is to wrap the target program's execution with a debugger program, so that when the MPI job is executed with n processes specified, the n processes are spawned with n instances of the debugger by having each MPI process be ran with a dedicated debugger. This strategy works, and offers a fair amount of functionality relative to the complexity of the setup. With this strategy, each process in the MPI job can be debugged individually, while maintaining the intended MPI communications between the processes.

This approach also has drawbacks. First, the debuggers spawned in this manner are oblivious of the existence of other MPI processes and their respective debuggers. This means that it is difficult to maintain a globally coherent view of the overall state of the MPI job, and impossible to achieve synchronisation of the debugged processes. This becomes a prevalent issue when introducing reversible debugging features, as it is not possible to maintain causal consistency on reversible debugging (described in chapter 2.2.3). Second, the user experience using this strategy is sub-par, as n individual debuggers have to be managed by the user. It is not obvious to the user which debugger corresponds to which process (as the process identifiers are conventionally hidden unless explicitly queried). Finally, running a conventional serial debugger instance per each MPI process results in large usage of computational resource, as most debuggers are not intended to be used in this manner and are not lightweight enough in their implementation. Hence, debugging an MPI job with this strategy is not feasible if a substantial amount processes is involved.

Global debugger

The most fitting solution to debugging MPI programs is to make use of a debugging tool designed for distributed process groups, including MPI jobs. Such a global tool can either spawn the MPI job itself, or dynamically connect to all of the MPI processes spawned in the MPI job. The debugger binds itself to all the n individual MPI processes. By having cognisance of all the MPI processes at once, and also having the ability to excerpt control over each process, this kind of debugger architecture can be highly useful. This sort of debugger architecture maps well to the distributed nature of MPI jobs. As

the transmission of messages by MPI operations occur with the participation of multiple processes – the sending and the receiving party – then the global view allows us to observe such events in their unity. It also allows developing higher-order debugging functionalities, such as “following a message”, dynamically constructing inter-process communication graphs, or causally consistent reversible debugging. Most wide-spread debuggers are however implemented to support debugging serial (possibly multi-threaded, but not distributed and parallel) programs. *OpenMPI* explicitly supports two debuggers capable of parallel debugging – the *TotalView* debugger [21] described in chapter 3.2.5 and *DTT* [22], both of which are commercial projects with closed source code. [23]

A debugger might also utilise an architecture that is a hybrid of the described approaches. The debugger implemented in this thesis and described in chapter 4 generally follows the global model, but also makes use of distributed lightweight wrapper processes for each process.

2.4 Debugging Utilities in the Linux Environment

In this section, an overview of a selection of debugging tools and utilities available in the Linux ecosystem is given. The focus on Linux is warranted by its rich offering in related tools, and the possibility of porting the Linux environment to other platforms via containerisation solutions. The described tools are selected by their relevance to the practical debugger implementation described in chapter 4, and by considering their general offering and prevalence of use (such as by the debuggers described in chapter 3.2).

2.4.1 Ptrace

Ptrace is a system call available in UNIX-based systems. As described in the Linux manual pages, “the ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee")” [24]. In the context of debuggers, the debugger tool assumes the role of the tracer for observing and controlling the execution of the program being debugged - the tracee.

Ptrace is a flexible tool and is often used as the primary mechanism through which the debugger orchestrates the execution of the target process. In the following functionalities of ptrace that are most relevant for debugging purposes are described and assessed.

Establishing the tracing relationship

A debugging tool must in some manner establish the tracing process by designating the target program as the tracee. This can be done in one of the two ways. Either, the debugger is attached to a running process, or the debugger starts the target process itself. In both cases, the tracee first must call ptrace with `PTRACE_TRACEME` request, in order

to signal the permittance of tracing. Following this, the tracer (a debugging tool) must call ptrace with PTRACE_ATTACH request with the process id of the tracee in order to assume control over the tracee. Naturally, there are rules and limitations to which kind of processes can be traced. In general, a process to which the tracer is permitted to send signals to, can be traced.

Pausing the execution

Pausing the target executable either when certain conditions are met (such as execution arriving at a certain line in the source code representation or during a system call) is a commonly offered functionality from state-of-the-art debugging tools. This functionality can be implemented by ptrace.

General pauses – The baseline behaviour of ptrace is to pause (or rather stop) the execution of the tracee on certain conditions. These conditions include the receiving of a signal, a call to enter or exit a system call, and other various scenarios. This mechanic provides the debugging tool a way to observe and possibly intercept the communication of the tracee with the external environment. When such conditions do not need special handling from the debugger, these pause events can be simply ignored and the tracee can continue it's normal execution.

Arbitrary pauses – A powerful functionality for a debugger tool is to allow the user to initiate a pause either immediately, or by specifying the conditions for triggering a pause. For immediate pausing, the debugger tool can either issue a PTRACE_INTERRUPT ptrace request, or send an appropriate process signal to the tracee. For pausing on the tracee at a certain point of execution, one approach is to instrument the tracee process by inserting an instruction that signifies an interrupt signal at a certain location in the instruction section of the process memory. This kind of instrumentation can be done by utilising the other ptrace utilities described in the following sections. As the tracee continues execution and arrives at the inserted interrupt signal instruction, it will raise a trap to the operating system (because the program is being traced), thus pausing the execution.

After any of the pausing conditions are met, the tracee process will transition to a stopped state. The state of the process is maintained, in terms of register values and memory contents, but the process is unable to continue the execution on its own. The debugging tool must be able to detect the transition of the tracee into the stopped state. This can be achieved by the linux *wait* system calls. The tracer can wait for the target process using this utility, to detect the pausing of the tracee along with the signal that the tracee was stopped with. Following a pause, the tracer can either kill the tracee process by sending an appropriate signal, or continue the execution of the tracee, by issuing a PTRACE_CONT request through the operating system.

Reading and writing process memory

Usually, an external process is prohibited to write to another process's memory, and under most circumstances cannot read the memory either. However, for debugging purposes (such as implementing breakpoint debugging described in chapter 4.2.1 such access might be needed. The ptrace utility exposes `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` requests which can be used to respectively read from and write to the tracee's memory. The tracer must specify the (virtual) memory address from which the read/write operation should be executed, along with either the size of the retrieved data or the contents of the data to be written, respectively. After this, the ptrace system calls handles the moving of data from one process to another. Naturally, the data manipulated in this manner is in the raw format of binary buffers. In order to use these methods in a meaningful way, the debugger must be aware of the contents of the tracee's memory. For reading or writing a value of the tracee's variable, the debugger would need to know aspects such as where is the value located in tracee's memory, how large is the binary representation of the variable, and how to convert the raw binary value into a meaningful representation. A common way of solving this problem for debuggers is using the debugging information and instructions that should be bundled within the tracee's binary. This data is included in compiled binaries optionally, and usually represented in DWARF format, which will be looked at more closely in chapter 2.4.3.

A limitation of the `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` functionalities is the size of the data that can be read or written. On the base level, the system call typically supports reading or writing a single machine word at a time. This means that manipulating a large volume of data with this mechanism can be cumbersome and also prone to failures, as the system call needs to be executed consecutively $size(data)/word\_size$ times, even for operations on consecutive memory regions. A more reasonable alternative for this purpose can be the *proc* filesystem described in chapter 2.4.2.

Reading and writing register values

In addition to manipulating the target's memory, ptrace exposes `PTRACE_PEEKUSER` and `PTRACE_POKEUSER` requests that provide the ability to read and write the contents of the tracee's registers. Accessing various registers can be useful in a multitude of ways. The following list describes registers that can be utilised by a debugger tool. Although differing in their implementation details and names, these registers are present in most processor architectures.

Instruction pointer register – The instruction pointer register holds the memory address of the instruction that is currently being executed by the process. The value of the instruction pointer can be used for locating the current state of the tracee within its source code context. Also, this value can be overwritten to divert the execution flow of the tracee. When changing the instruction pointer value, it should be kept in mind that

the execution of instructions in a non-default order can have potential side-effects on memory contents and other register values. Thus, the debugger tool must ensure that its instrumentation does not yield unwanted side-effects in the state of the tracee.

Stack pointer and Base pointer registers – The stack pointer (also referred to as the stack register) and base pointer are used to maintain a reference to the call stack of the process. Accessing the call stack can be utilised in a multitude of ways for a debugging tool. These two registers can be used in conjunction to unwind the call stack, thus obtaining the stack trace, which is necessary to locate the sequence of subroutine calls that the process has made in order to reach the present instruction. Moreover, a process often stores information relevant to the subroutine calls on the call stack, such as references to function parameter values and some variables scoped to the subroutine call. A debugger can thus present this information as a detailed representation about the current state of the target process.

Registers for hardware-level memory watchpoints – A useful utility a debugger tool can offer is to allow setting watchpoints on certain memory addresses, to notify the user when a certain memory location was accessed by the target. Such memory watchpoints can either be implemented on a software or hardware level. The hardware-level watchpoints are considered to be much more performant than software watchpoints [25]. However, hardware-level requires the existence of purpose-built registers of the cpu, which expose this functionality in a low-level and thus performant manner. Support for such registers varies between architectures, and is generally quite limited. The x86 processor architecture for example exposes 4 such registers called Debug Registers, meaning a maximum of 4 memory watchpoints can be set at a given time for a process [26]. In [27], an efficient scheme for implementing software-level memory watchpoints is proposed. This however requires implementing dynamic instrumentation of the target binary, which introduces a layer of complexity and a possible source of unwanted side-effects. Hence, it is rather a mechanism for a specific tool, than for a general purpose debugger.

Hardware performance counters – Some functionalities a debugger might offer require knowledge of the exact count of instructions executed by the target at a given time. For example, the rr debugger (described in more detail in chapter 3.2.4) requires this kind of hardware support for enabling deterministic recording and replaying [28]. In general, this instruction count can be used as a logical discrete time-measure for the target program execution. Some modern cpu architectures offer this kind of mechanism via registers referred to as *hardware performance counters*. As the name suggests, the main purpose of this register is to allow monitoring the amount of instructions executed per time period to estimate the performance of the system on hardware level. If present in the system, this register can be utilised successfully for debugging information purposes, as the rr debugger demonstrates. [14]

2.4.2 Proc Filesystem

The proc filesystem (*procfs*) is a unique filesystem in Linux and other Unix-like operating systems that presents information about processes running on the system as a filesystem. In Linux, procfs was introduced in version v0.97.3 [29]. For each process with process id `pid`, there exists a directory `/proc/pid` that hosts a collection of files that describe aspects about the running process. Some files also enable interfacing with the process by writing to the file. Procfs exposes information and functionality to the user-space from kernel-space that would have to otherwise be retrieved via dedicated system calls. Such functionality can be useful for debugging tools as well, as described in the following.

Reading and writing process memory

A debugging tool can benefit from access to the target process's memory for multiple goals. From being able to read the process's memory, the debugger can access internal data of the target, such as the values of variables. Having write access to the target process's memory enables an array of debugging functionalities. One very common use case is the ability to set breakpoints in the target process by inserting interrupt instructions at specific locations in the target's instruction area. Another use case is overwriting values of internal data structures with different values, for example to restore a checkpoint recorded earlier in the target process's execution. The following files are of use for accessing the memory of a process via procfs:

`/proc/pid/maps` – The `maps` file contains a memory map of the process. It describes how the memory of the process is divided into blocks, such as blocks for the stack and heap. Each block is also described with a corresponding set of linux access permissions (read/write/execute). For example, code sections of the memory are executable, whereas sections storing data are not.

`/proc/pid/mem` – The `mem` file hosts a binary image of the target process's memory, by mapping the virtual address space of the target to locations in the file. Reading this file is identical to reading the memory of the target process directly. Moreover, this file also allows writes, and the write action corresponds to writing into the target process's memory. This kind of access to the memory of the target is much less constrained than utilities offered by ptrace (peek/poke data), which enable accessing memory one machine word per system call. Since this functionality is very powerful, it requires privileged access, and the required access corresponds to having ptrace capabilities to the target.

By utilising the combination of these mentioned files, a debugger can implement multiple useful debugging features, as demonstrated by the practical implementation of the debugger described in chapter 4.

2.4.3 DWARF Debugging Standard

DWARF is a standardised format for representing debugging-related information of a program. Its main goal is to provide metadata about the program's internal structure with reference to the original source code constructs (such as variable names, function signatures, etc.) and thus enable "decoding" the internal structure of compiled or otherwise prepared executable files. The DWARF information can be included in ELF (Executable and Linkable Format) files, usually configurable by some parameter of compiler tools. The format however is independent of object files. The DWARF standard has seen 5 major releases over the course of multiple decades, the latest major version 5 being released in 2017 [30]. DWARF is designed to be architecture independent and applicable to any processor and operating system. A wide variety of programming languages are supported, with emphasis on procedural languages, such as C and C++, but also including more modern languages such as Go and Rust. [31]

Structure of DWARF Data

The DWARF format represents debugging data in a proprietary data structure called DIE (Debugging Information Entity). Each DIE has a tag representing the type of the entity (e.g. `DW_TAG_Variable` for variables, `DW_TAG_subprogram` for subroutines or functions), and attributes - key value pairs for representing aspects of the entities, such as `DW_AT_Name` present for `DW_TAG_subprogram` entities describing the name of the given subroutine or function as declared by the programmer in the original source code. A DIE unit can have other units as its children, and the units are organised into a tree structure that must be parsed by the reader. [31]

Mapping DWARF information to process memory

Relevant for debugging purposes, entities such as variable and function declarations include an attribute `DW_AT_location` that specifies the instructions for locating the given entity in the process memory. Since programs exhibit a large variety of different locations for storing data (such as processor registers, the (process call-) stack, dynamically allocated memory regions, static memory regions and so on), the location descriptions must be sophisticated enough to represent all such cases, in addition to addressing the specific byte-address inside the broader location context. For such purposes, the DWARF standard describes a mechanism by which the location instructions can be decoded. The trade-off for the versatility of the location instructions is the relative complexity of parsing the instructions.

In addition to locating the desired entities in the processes memory, the binary representation obtained this way is not meaningful in debugging context without a conversion to the data-type intended by the programmer. For this purpose, DWARF

includes information about data types, such as the byte-size and encoding of a type. This information can be used to convert the binary representation of an entity to its intended (possibly human-readable) format for meaningful investigation. [31]

3 Previous Work

This chapter describes previous implementations of software that is relevant to the implemented debugger. First, an assessment of checkpointing tools and frameworks in the Linux environment is given, as checkpointing is a suitable mechanism for implementing reversible debugging features. Subsequently, a selection of debugging tools are reviewed and analysed, which are relevant to the practical scope of this thesis.

3.1 Checkpointing Tools and Frameworks in Linux Environment

As was discussed above, a debugger offering reverse debugging actions may be based on a checkpoint-restore mechanism to deliver said actions. To this end, it is worth investigating existing developments in the problem space of (distributed) checkpoint-restore mechanisms. It is worth mentioning in advance that the problem space these tools aim to solve is wider and slightly different from the goal of reverse debugging functionality, being mostly aimed at stateful process migration, fault-tolerance and general checkpointing procedures. However, the mechanisms these tools utilise could be used to implement checkpointing for reverse debugging purposes as well. In this chapter, such existing checkpoint-restore solutions are discussed.

3.1.1 BLCR - Berkeley Lab Checkpoint Restart

Berkeley Lab Checkpoint Restart [32] is a now-discontinued project to implement robust kernel-level checkpoint/restart functionality for Linux. Major use cases proclaimed by the project's authors included a wide variety of scenarios such as job scheduling, process migration and backup procedures. BLCR was developed with the intention to support parallel applications such as MPI.

Being mainly a kernel level implementation has both advantages and disadvantages, although the authors advocated for the kernel-level approach as more favourable than the user-level. From the design standpoint, the insufficiencies of the user-level implementation put forward by the authors include the undesirability of replicating kernel-level data in the user-space and the insufficiency of existing APIs for interfacing with the kernel to produce a fully idempotent checkpoint-restore behaviour. A notable positive aspect of the kernel-level implementation is that the user application can remain oblivious to the existence of a checkpointing solution and does not need special setup to support checkpointing. Another useful feature originating from the kernel-level implementation is the ability of BLCR to maintain a consistent Linux environment for the application across restarts, meaning process ids remain constant across restores, which is crucial for a category of processes that rely on process ids and their parent-child relations as integral data points. The kernel-level implementation however also has notable downsides, that are apparent in retrospection. Most notably, the kernel-level approach demands explicit

adaptation of the implementation by different operating system vendors, and continued support in the OS release- and maintenance cycle. BLCR was developed as an "out-of-tree" kernel module which was not incorporated into the primary distribution of Linux. At the time of writing this thesis, the latest Linux kernel version to support BLCR is 3.7.1, released in 2012 [33].

3.1.2 CRIU - Checkpoint/Restore in Userspace

CRIU [34] is an open-source software project to implement checkpoint-restore functionality in Linux userspace. This tool can be used to take a snapshot of a running process and save the snapshot as files. The snapshot can in turn be used to start a process with the initial state described in the snapshot. Since its original conception as a project of OpenVZ, this project has been made use of in many other mainstream virtualisation and containerisation implementations such as Docker, Podman and is reportedly being implemented to be supported by Kubernetes to conduct live container migration. [35]

One of the advantages of being implemented in user-space is that CRIU does not require modifications to the OS kernel. This means that the solution is favourable if portability and ease of setup is required. It also means that the project is more likely to be supported by a larger variety of operating system versions, since user-space apis are often more uniform whereas the internal implementation can differ between vendors and releases.

In [36], an approach for accommodating CRIU with usage of OpenMPI is discussed for process migration purposes. It is als shown that with some modifications, this tool can be used to support the migration of OpenMPI jobs. The authors also evaluate that among the existing prevalent checkpoint-restart implementations (including the other implementations described in this chapter), CRIU is the most suitable implementation for this goal.

3.1.3 DMTCP - Distributed MultiThreaded Checkpointing

DMTCP [37] is a user-level checkpointing implementation that allows the creation and restoration of checkpoints for distributed applications. It is an open source project that has been proven to support checkpointing a wide variety of applications including MATLAB, Python and implementations of MPI. The quality of having an user-level implementation implies high portability of the solution. DMTCP has a very low temporal overhead of creating checkpoints, making it efficient for even large-scale jobs that are prevalent in High-Performance Computing.

The checkpointing approach DMTCP utilises is *coordinated checkpointing* – during a checkpoint, the state of the whole application is suspended. This means the checkpoint operation yields a global checkpoint, although the checkpointing process is carried out in a distributed manner on each node. The subcomponent executing the checkpointing

operation is MTCP, a single-node multithreaded checkpointing utility. [38]

A notable quality of DMTCP is its ability to account for operating-system level artefacts when creating checkpoints. DMTCP accounts for things such as open sockets and files and parent-child process relations, which are all serialised into the checkpoint image files and restored when the process itself is restored. This ensures that the original process environment is preserved as much as possible when restoring checkpoints and no side-effects occur.

MANA – MPI-Agnostic Network-Agnostic Transparent Checkpointing

Whereas DMTCP was introduced over ten years ago, a more recent and very interesting development of this family of tools is MANA – the MPI-Agnostic Network-Agnostic Transparent Checkpointing project [39]. This initiative presents a novel solution for checkpointing tailored specifically for MPI. MANA introduces the concept of *split-process* - an MPI process node is internally separated into two separate processes – one for the user MPI application and the other for MPI runtime libraries – that share the same address space. This split-process technique appears to be novel in the broader context of process orchestration. In the MANA checkpointing approach, only the user-application-half of each node is checkpointed, whereas the process area hosting the MPI runtime is excluded from checkpoints. This setup has multiple advantages. First, this approach allows dynamically replacing the underlying runtime of MPI with a different implementation during checkpoint/restarts, which opens up a manifold of new use cases such as migrating MPI jobs between clusters with different architectures, network configurations and MPI implementations. This aspect of portability also implies the little maintenance overhead for supporting different implementations and new releases of MPI. Second, it significantly reduces the complexity of checkpointing both procedurally (needing less orchestration to account for mimicking a homogenous MPI environment across restarts) and resource-wise (as checkpoints are much more light-weight since only user application needs to be checkpointed). The checkpointing approach in this solution provided inspiration for the checkpointing implementation implemented in the debugger described in chapter 4.

3.2 Assessment of Relevant Existing Debuggers

In this chapter, an overview is given of a selection of debuggers. The debuggers are selected by their relevance to the topic of this thesis, by factors such as the support for distributed and reversible debugging, support for ensuring deterministic debugging, the portability of the implementation and the general popularity of the tools.

The *GDB* debugger is essentially the standard of modern debuggers and the backbone of many higher-level tools. The *CauDEr* project is an existing implementation of a causally consistent reversible debugger. *URDB* is an interesting approach for utilising

checkpointing on an existing debugger to achieve reversible debugging. The *rr* project is a sophisticated tool for enabling deterministic debugging for non-deterministic software with a record-replay mechanism. Finally, the *TotalView* debugger is a highly specialised debugger developed for the specific needs of high-performance computing.

3.2.1 GDB - GNU Project Debugger

GDB, the GNU Project Debugger [40], is an open-source debugger project developed in the GNU ecosystem. Initially released in 1986, it has seen continuous development for over 36 years, with new releases still regularly appearing. GDB is compatible to run on Unix-like operating systems as well as on Windows. The list of supported programming languages ranges from C and C++ to more modern languages like Go and Rust. The paradigms by which GDB operates expect the target to be implemented with a compiled programming language; there is no support for interpreted languages directly. GDB is integrated with a wide range of developer tools, such as IDEs. The extensive history and emphasis on open source has contributed to the wide usage of GDB and established it as a standard- or reference implementation of a classical debugger. It supports all common operations, such as setting breakpoints, pausing the execution, examining the stack and other memory regions of the target, etc. GDB also has sophisticated features, including remote debugging, memory watchpoints and support reverse debugging, all of which are examined below.

Remote debugging – A useful mechanic that GDB implements is the ability to debug targets that are running on a machine different from the runtime machine of the debugger. This can be achieved in two ways, either by having the remote target implement a "stub" for communication with the GDB protocol, or executing a *gdbserver* - a back-end for the GDB with remote communication capability - on the target machine, and connecting a GDB debugger to it from the host machine. Remote debugging enables flexibility for the user, as well as debugging targets on different platforms than the host machine (such as a different processor architecture). [40]

Memory watchpoints – GDB supports setting memory watchpoints that enable monitoring access of a memory location of the target. GDB implements the mechanic for both hardware and software watchpoints. As described in the previous section, implementing watchpoints on hardware level is efficient, but lacking in processor architecture support and in the amount of concurrently available watchpoints. Thus, GDB will default to using hardware watchpoints, but will fall back to using software watchpoints, which are always supported, but result in a drastic slowdown of the target's execution. Memory watchpoints can be set by referencing a variable name to monitor changes made to the variable directly, or by referencing a particular memory location, which also enables detecting memory corruption bugs.

Reversible debugging – Since version 7.0, GDB includes the support for reversible debugging [41]. This implementation enables reverse execution through step-based

commands (e.g. reverse-step that undoes the effects of the most recently executed instruction) and also continued execution in reverse via the reverse-continue command. The implementation is based on tracing – to enable reverse debugging, the recording of instructions must be enabled first with the record command, after which, the debugger will proceed to record the effects of every instruction in the target. Although the tracing-based solution is very slow, lacks the support for multi-threaded programs and has limited processor architecture support, this implementation is notable as at the time of its release it provided the first open-source implementation of reverse debugging. [3]

Distributed debugging – There is no explicit support by GDB to debug a distributed program, but this kind of debugging can be achieved by executing instances of *gdbserver* on each machine node, and executing GDB with multi-mode that supports concurrently connecting to multiple processes. However, the resulting setup (and user experience) is the same as running n separate instances the GDB debugger, so no apparent benefit is gained that facilitates the distributed debugging experience.

3.2.2 CauDEr - Causal-Consistent Reversible Debugger for Erlang

CauDEr is an implementation of a causally consistent reversible debugger for the Erlang language, developed by Lanese et. al. [1].

Erlang

The Erlang language is a fundamentally concurrent programming language based on the *actor model* - an *actor* is a lightweight sub-process spawned in the program (an actor may spawn other actors as its children), that can communicate with other actors via a message-passing mechanism. The execution of actors is scheduled to happen in a concurrent manner – each sub-process is similar to a lightweight thread or process. Erlang programs are executed in a specialised virtual machine that orchestrates the lifecycle of a process and its sub-processes, including handling the delivery of passed messages between the sub-processes. An Erlang program may be executed on a single machine or on a cluster of distributed machines, while the user process itself remains oblivious of the particularities of the runtime environment. The fundamental principles of Erlang make it similar to the MPI paradigm. Both have an emphasis on a (potentially distributed) multi-agent mechanism that relies on message-passing as the method of communication. Erlang is encapsulated in its implementation in the Erlang run-time VM, whereas MPI is open to different implementations. Since the fundamental principles are similar, the principles for implementing debugging tools can be carried over respectively as well. Hence, the CauDEr project has concepts that can be applied to the MPI, as demonstrated by the implementation described in chapter 4.

Causal-consistent reversible semantics of Erlang

The theoretical foundation of CauDEr relies on the formulation of formal semantics of the message-passing operations in the Erlang language. Conceptually, an Erlang process - a system - is modelled as a pool of processes that can interact among themselves via message-passing. A running system is composed of $\Gamma; \Pi$, Γ denoting a global mailbox - a multi-set of pairs in the form $(target\_process\_pid, message)$, and Π denoting a pool of processes. Each process is modelled as having a unique identifier pid, an environment and expression to evaluate in the environment, and a designated mailbox. Upon a process sending a message, the message is added to the global mailbox, from which the process identified by $target\_process\_pid$ can read the message. Upon this notation, operations denoting all possible events are defined, such as sending and receiving a message or spawning of actors. Upon the regular actions, semantics for rollback operations are defined that formally describe the "reverse" of regular actions. To enable this, the formal model is expanded by assigning a history state to each process for recording past events, and each message is designated a timestamp λ to identify the ordering of message history. [1]

The definition of these formal semantics allows formal verification that the rollback operations behave in a logically correct manner. Moreover, the reverse operations are defined in a way that preserves causal consistency. In the words of the authors: "a backward step is causal consistent if an action cannot be undone until all the actions that depend on it have already been undone"[1]. Hence, the rollback operations are defined in this manner that guarantees the undoing of dependent actions to undo a given action, thus ensuring causal consistency.

Causally consistent reversible debugging

CauDEr supports a wide set of debugging actions that enable the execution of instructions in the forward or backwards direction. As an Erlang process is a collection of essentially autonomous sub–processes, the debugger allows the user to control the execution of each spawned sub–process. While executing in the forward direction, an agent acts as it regularly would, stalling on message receiving events until the actor sending the message has been instructed to progress to the send instruction; and otherwise executing in an unconstrained manner. For backwards execution, certain actions that would violate causal consistency (such as reverting a send operation on a process if the corresponding receive operation isn't reverted on the receiving process) are prohibited. The debugger has different operational modes to either undo all the causally connected actions automatically or require the user to undo each causally connected action manually.

This mechanism guarantees that the global state of the Erlang process is always in a causally consistent state - hence representing a valid possible state in the execution of the process (meaning there are no violations of causality). Without such global coordination

(allowing for unconstrained backwards execution of the agents), this property would not hold, allowing for logically incoherent states to occur.

3.2.3 URDB

A unique reversible debugging implementation is described in the project URDB – A Universal Reversible Debugger Based on Decomposing Debugging Histories [42]. This approach makes use of the DMTCP checkpointing implementation, which is shown to be general enough that it can be utilised to checkpoint a debugger program's execution itself to provide a degree of reversible debugging capability. The solution works – as is described by the project name – by decomposing the debugging history of the debugger. The debugger process is wrapped with URDB that relays commands to the debugger, while monitoring the user-inputted commands. A complimentary checkpointing command is presented to the user by URDB that checkpoints the debugger process along with the debugged target process. The resulting checkpoints can in turn be used to revert the debugging experience into an earlier time-state via reverse- commands offered by URDB, which are carried out in a restore-and-re-execute manner. The significant benefit of this approach is its generality that allows easy adaptation of different existing debuggers and software execution environments. Additionally, the overhead of URDB upon normal execution (of the debugging software) is low when checkpoints are made infrequently. A notable potential caveat is the loss of granularity of checkpointing arising from the need to record checkpoints manually by the end-user, and the associated potentially suboptimal user experience. In general, this approach seems to be of high merit for its simplicity and adaptability.

3.2.4 rr

The rr (short for Record and Replay) project [43] is a debugging tool for Linux designed to record and deterministically replay the execution of a process. Originally designed by Mozilla, its primary goal from the development phase was to provide a debugging tool for complex concurrent software projects that can be prone to timing-related bugs such as race conditions, that are difficult to detect and observe with generic debugging tools because of their nature to appear intermittently. This design goal puts the emphasis on deterministic replay of the program execution.

Interaction with the rr tool occurs in two primary stages - record and replay. During the recording phase, the process execution is recorded in a manner to capture the timing of all possible sources of non-determinism in the process. The result produced by the recording stage consists of written files, which in turn can be replayed. The replaying phase takes as input a recorded execution and re-executes the process in the exact manner that it was recorded as, meaning the internal events and state transitions in the process happen exactly as they did during the initial recording. During the replay phase, rr

acts as a *gdbserver* and presents the user with a gdb debugger ui accepting regular gdb debugging commands as input for orchestrating the replay.

As for on what level the recording happens, the authors state: "[we] identify a boundary around state and computation, record all sources of nondeterminism within the boundary and all inputs crossing into the boundary, and re-execute the computation within the boundary by replaying the nondeterminism and inputs".[14] These sources of nondeterminism include signals delivered to the process, results from system calls and so on, for which the results are recorded in the produced recording. To measure and identify the timing of these non-deterministic events, rr uses hardware performance counters that measure the count of instructions executed in the process until the particular event. This counter is used as a global notion of time in the process. For multi-threaded applications, executing one thread at a time guarantees a global ordering of the events. [14]

In addition to standard debugging commands, rr also supports reverse debugging commands that are made possible via a lightweight fork-based process checkpointing solution. The technical report of rr does not describe the specifics of the checkpointing solution but the authors state that it is performant - "It's completely reasonable to reverse-continue from the end of a Firefox run all the way back to the beginning"[44].

Whereas rr is not a standard debugger in the traditional sense (compared to for example the gdb debugger), the functionalities it offers have proven to be a practical tool used daily by developers of complex applications, such as Firefox, Chromium and LibreOffice.[14]

3.2.5 Commercial Projects

There exist also commercial implementations for reversible and or distributed debugging. Being closed source projects, conducting technical analysis of these tools here is not possible. Notable initiatives are however mentioned.

TotalView [21] is a debugging and dynamic analysis tool designed specifically for High-Performance Computing applications. It supports multiple distributed computation environments including MPI, OpenMP and CUDA. Totalview debugger includes a reverse debugging capability module referred to as ReplayEngine. The reversible functionality is achieved by loading a code instrumentation library to the target process that manages recording the execution and allows for replay functionality. [45]

UndoDB [46] is a "time travel debugger" for C, C++ and Java languages. It supports debugging software for Linux and Android platforms and has reversible debugging capabilities. Analogously to TotalView, it uses on-the-fly instrumentation of the target program to capture the sources of non-determinism and allow for replay functionality. [47]

4 Implementation of Causally Consistent Reversible Debugger for MPI

This chapter describes the debugger implemented as the practical part of this thesis. In the first sub-chapter, a detailed analysis is given of the architectural design, including reasoning for design aspects and their resulting benefits and drawbacks. In the second sub-chapter, an overview of the capabilities offered by the debugger is given.

4.1 General Architecture

The implemented debugger is designed for debugging distributed computational programs developed for execution within the MPI parallel runtime. Instead of debugging a single target process, n instances of the distributed MPI computation are debugged simultaneously.

The parallel execution model is an integral design constraint for the debugger – the architecture of the debugger is distributed in a way that is aligned to the distributed computational process itself. This enables favourable software design principles to be followed, such as the separation of concerns of the subcomponents.

The debugger is composed of hierarchically aligned components – a centralised *orchestrator* and n *debugger nodes* that wrap the individual MPI processes spawned by the MPI runtime. Figure 6 illustrates the general architecture of the debugger.

A design goal for the debugger was to not require additional software components to be installed on the user system, other than the evidently necessary tools for compiling and executing MPI programs (An MPI compiler and an MPI runtime providing *mpicc* and *mpirun* commands respectively). A convenience from the debugger standpoint would be the presence of MPI runtime libraries compiled with included debugging information. However since this setup is improbable to exist in the default configuration of MPI users, the debugger does not assume such preconditions and uses other application-level mechanisms to interface with MPI.

The debugger is designed with explicit support for MPI applications written in C language, but uses mechanisms that allow generalisation to support other languages (including C++, Go, etc). At present time, the debugger supports the x86 processor architecture, but is designed in a manner that allows extending support for other processor instruction set architectures.

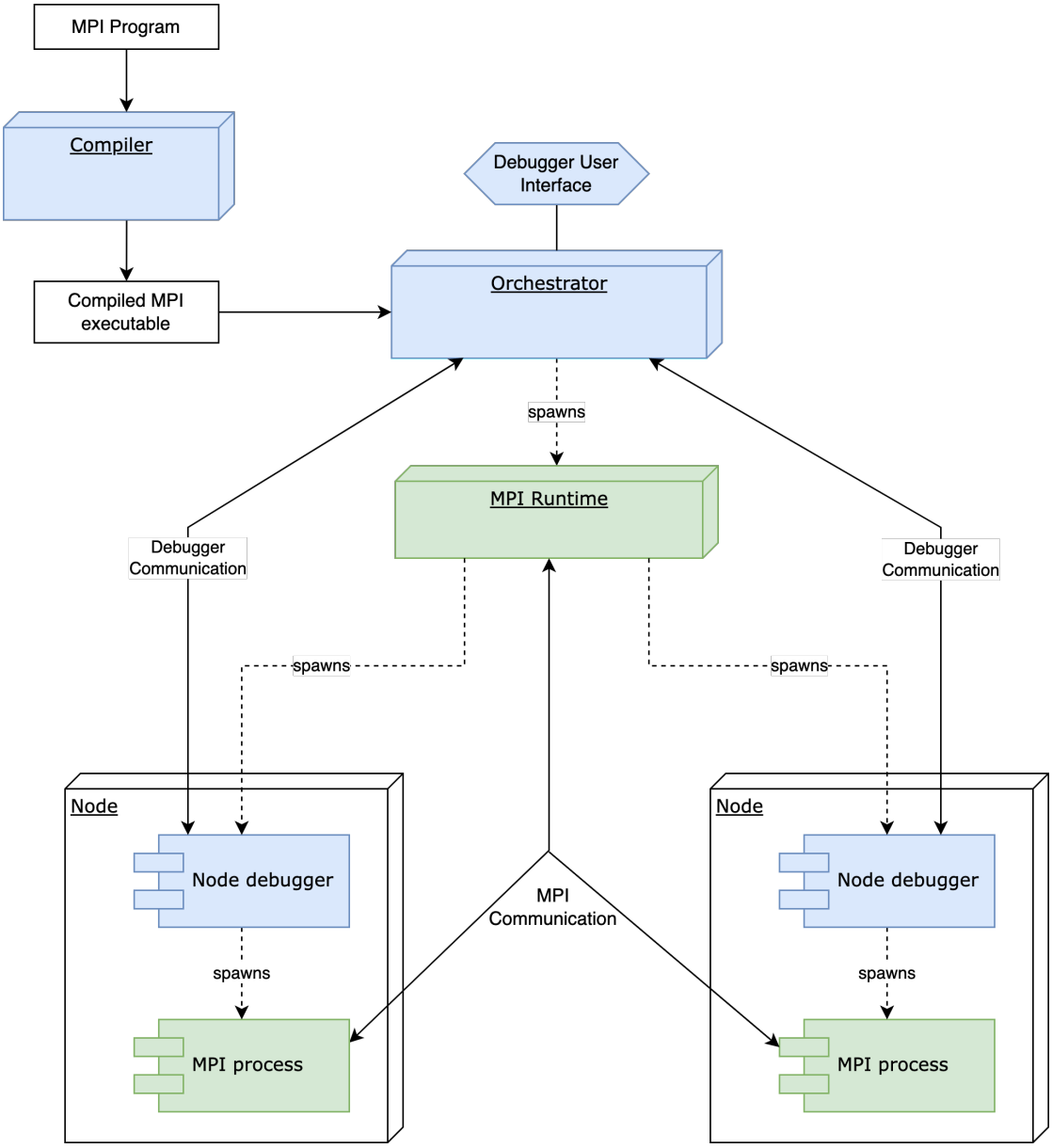

Figure 6. Debugger architecture component diagram. Green – MPI components, blue – debugger components.

Implementation language

The debugger is implemented in the Go programming language [48]. Go is a relatively low-level language with a C-like nature but has many favourable aspects, such as memory-safety and -management, built-in concurrency and a more user-friendly programming style with influences from functional programming languages. Since a debugger node is attached to each MPI process, the debugger component should aspirationally have a low overhead on a node, otherwise spawning a large number of nodes might incur a high performance overhead from the debugger components. The low-level nature of Go permits this as demonstrated by the implementation.

Relevant to this project in particular, the Go standard library includes convenient functionality for interfacing with Linux system calls as well as parsing DWARF data from compiled executables.

4.1.1 Bootstrapping sequence

Compilation phase

The debugger includes a supplementary compilation utility that is used to produce a suitable binary for execution with the debugger. There are two primary reasons for including a dedicated compilation utility. The first reason is ensuring the presence of debugging data in the compiled binary (DWARF data described in section 2.4.3). By having control of the compilation phase, the debugger can ensure to include in the compilation process the relevant compiler parameters that instruct the compiler to include the debugging-related information in the resulting binary. The second reason is wrapping the MPI runtime library with an intermediary layer during the compilation phase, which enables the debugger to easily intercept calls to MPI. This intermediary layer is included in the DWARF data, which enables the debugger to set automatic breakpoints that capture calls to MPI subroutines, which is integral to the reverse-debugging functionality described below. This way, the need for MPI runtime libraries with included debugging information is eliminated, however there are alternative scenarios for solving this problem, such as dynamically (pre-)loading an instrumented MPI library into the process.

Execution phase

The debugger utilises the native process creation mechanism of the MPI runtime (mpirun) for its bootstrapping sequence. The bootstrapping process happens in the following stages:

1. The user executes the debugger, specifying a compiled binary as the source application to be debugged, along with n – the number of MPI nodes to be spawned. This user input is analogous to starting an MPI job directly.

2. The debugger (more precisely the *orchestrator* component) creates an MPI job with the specified amount of nodes, and instead of specifying the MPI program as the target, it spawns n instances of the node debugger components that receive a handle to target MPI program.

3. Upon startup, the debugger nodes follow a bootstrapping phase that includes:

   (a) establishing a remote connection with the orchestrator component;
   (b) parsing the DWARF debugging data from the target binary;
   (c) starting the target binary and setting up the tracing relationship via the Ptrace system call;
   (d) inserting automatic breakpoints for calls to MPI subroutines.

After the completion of the bootstrapping sequence, the debugger assumes the process of accepting and executing user-inputted commands.

4.1.2 Communication

Since the debugger is designed in a distributed manner, the communication between the components utilises network communication, more specifically Remote Procedure Calls (RPC). The Go language standard library includes a built in `net/rpc` module for RPC functionality [49] that allows providing access to the exported functionality across a network connection. This communication implementation is very easy to integrate into the existing object-oriented nature of Go data structures, and abstracts away the complexity of managing the communication manually.

Both the orchestrator and the node debugger components act as a client-server hybrid for communicating with each other. For operating the server and client functionality simultaneously in a single component, *goroutines* are utilised, which is a convenient-to-use abstraction offered by the Go language for managing concurrency.

4.1.3 Orchestrator

The primary function of the orchestrator component is, as the name suggests, orchestrating the execution of the node debuggers. It exposes a rpc server to which the nodes connect to for reporting various events including registration on startup, recording checkpoints, logging and de-registration on exiting. Most of the commands are directly relayed to the debugger nodes without top-level orchestration, with the exception of the rollback operation. As described below, the rollback operation needs to be orchestrated globally to maintain a causally consistent state across the MPI job. The orchestrator holds in its state a checkpoint log for recording all the checkpoints reported by the debugger nodes. This checkpoint log is recorded as a dependency matrix – upon receiving a new checkpoint, the orchestrator will determine the dependencies between send- and receive-messages

that are referenced as causal links between the timelines of the communicating processes. This dependency matrix is used to find a set of checkpoints required to be rolled back for a requested rollback operation. This rollback process is described in more detail in the section 4.2.2.

4.1.4 Debugger node

The debugger node component encapsulates all the functionality for the debugging of a single process. As it is bootstrapped to be the parent process of the designated MPI process with Ptrace capabilities over the latter, it is able to intercept, observe and modify the behaviour of the MPI process.

The debugger node holds in its data store various data required for the debugging process, most importantly:

- the DWARF information about the target binary;
- a handle (essentially a Linux process id) to the running MPI process;
- a record of breakpoints currently inserted to target;
- a record of checkpoints recorded by this debugger node;
- information about the running binary's call stack.

The debugger node is designed to be executable both in a distributed setup and in a standalone manner, acting as an atomic debugger for a single processes.

4.1.5 User Interface

The debugger incorporates both a command-line interface and a graphical user interface for receiving commands and presenting the state of the MPI job that is currently being debugged. Figure 7 displays a screenshot of the debugger's interfaces during a debugging session.

The command-line interface is primarily for presenting the user with an event log from the debugger, such as the occurrence of events like triggering a breakpoint on a particular node, or printing the resulting values of retrieved variables. It accepts textual commands in a manner that is standard across most debuggers (such as GDB).

The graphical user interface is incorporated for presenting a graphical representation of the recorded MPI operations, and their causal relationships. Moreover, the graphical view allows initiating rollback actions for reverting the state of the MPI process nodes to recorded MPI operations. The rollback operation is carried out in multiple stages. First, the user selects the action that they wish to revert back to, by clicking the corresponding node in the recorded execution graph. Subsequently, the debugger will compute the list of additional checkpoints that need to be rolled back in order to maintain causal consistency on the whole MPI job scale. The user can then review these checkpoints and either commit or abort the rollback operation.

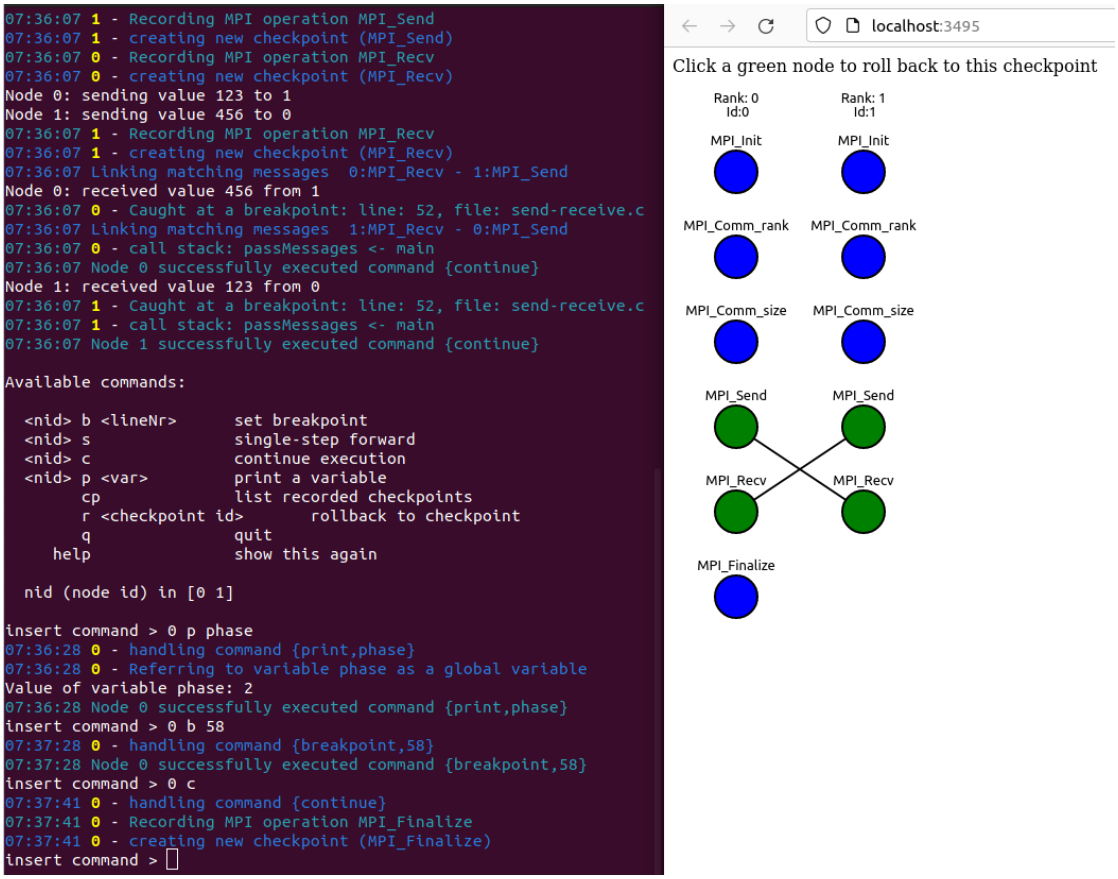

Figure 7. A screenshot of the command line interface (left) and the graphical user interface (right) during a debugging session.

As a future development, it would make sense to unify the two user interfaces into a single interface that would incorporate the functionality of both current interfaces.

4.2 Capabilities

4.2.1 Baseline Debugging Functionalities

The debugger offers a selection of commands for the user to interface with the running target. These commands include:

- *continue* – sends a Ptrace command `PTRACE_CONT`. Used to resume the execution of a stopped target. Passes the control flow to the target, until an event stops the debugger again (either because of hitting a breakpoint or finishing the execution)

- *set breakpoint* – takes as input l – a line number in the program source file and inserts a breakpoint instruction at the corresponding line. The breakpoint instruction is an interrupt instruction that is written into the program text area, the area where instructions are read from on execution. The target will stop when it encounters the interrupt instruction on execution, passing the control flow to the debugger. Since the original instruction is replaced upon inserting a breakpoint, the debugger will record the original instruction prior to inserting the breakpoint, and will replace the interrupt instruction with the original instruction when the target hits the breakpoint.

- *single-step* – issues a Ptrace command `PTRACE_SINGLESTEP` to the target, analogously to the continue command. The target will resume execution and will execute exactly one instruction before passing stopping and passing the control flow back to the debugger

- *print* – takes as input s – an identifier for a variable in the program and retrieves the current value of the variable, if a corresponding variable is found. The variable matching is analogous to how values are referenced in the program itself – first, the call stack of the program is inspected to locate variables declared in the functions in the call stack, or arguments of these functions matching the specified identifier. If no matching variable is found, the global variable space is inspected. If a matching variable declaration is found, its DWARF data is used to locate the memory address, byte–size and original data type of the variable. Then, the binary value of the variable is retrieved from the memory of the target and converted to the original data type before presenting it to the user.

  It is important to note that at present time, a limited amount of data types and location instruction decoding patterns are supported by the debugger, as implementing support for all possible cases of data types and location instructions is a considerable amount of work that did not fit into the scope of this thesis.

- *exit* – will cause the debugger to shut down, causing the stopping of the MPI job as well.

- *list checkpoints* - prints a textual representation of all checkpoints recorded by the debugger nodes up until that point of execution in the target processes. When the graphical interface is used, this command is not directly relevant.

- *rollback to checkpoint* - takes as input a checkpoint id that the user desires to roll back. The user can roll back to the automatically created checkpoints that are recorded on calls to MPI functions (such as sending and receiving a message). When the user issues a rollback command, the orchestrator will determine the set of checkpoints across all nodes that need to be rolled back in addition to the

supplied checkpoint in order to preserve causal consistency in the overall MPI job (described in more detail in next section). The user is presented with this computed list of supplementary checkpoints for confirming the operation. If confirmed, a rollback operation is made on the matching node of each determined checkpoint, reverting the computational process to the recorded state. The rollback operation can also be executed via the graphical user interface.

4.2.2 Support for Reversibility

In order to support reversible debugging, the user is presented with the rollback command described in the previous section. This section describes how the checkpointing and rollback solutions are implemented.

The debugger uses automatically inserted breakpoints for calls to MPI functions to record checkpoints in the execution of the MPI process nodes. Whenever an MPI node enters a call to some MPI function (such as sending or receiving a message), the debugger node will record the current state of the MPI node into a checkpoint.

The checkpointing procedure is *independent*, meaning there is no global coordination in the checkpoint creation process. A significant advantage of independent checkpointing is the lack of overhead originating from global synchronisation – the other nodes can resume their normal execution when one node is performing a checkpoint. Another advantage is the small footprint of the checkpoints, as one checkpoint records the state of one node. Moreover, the independent checkpoints allow rolling back only the desired subset of the distributed computation (of course, multiple checkpoints can be rolled back to achieve a global rollback). This allows for more granular operations by the user.

Creation of checkpoints

Creation of checkpoints For constructing the checkpoints, two checkpointing solutions were evaluated and experimented with – file-based checkpoints that are serialised to disk and *live* checkpoints obtained via forking the target process. Both strategies were viable, but since the latter strategy has the proclivity to exhaust more system resources (as each running process requires resources to be sustained, and the file-based checkpointing utilises only disk space), the file-based checkpointing solution was incorporated to the currently final version of the debugger.

The file-based checkpoint is constructed by serialising the state of the target process into a file that is written to disk. The checkpointing process is as follows:

- the processor register contents of the target process are captured. Since this data structure is quite lightweight, it is stored in the memory of the debugger node;
- the proc filesystem file `/proc/pid/maps` is utilised to locate the areas of the target process memory which will be checkpointed.

- The proc filesystem file `/proc/pid/mem` is used to retrieve the binary contents of the mentioned memory areas. The contents are written to the checkpoint file with the corresponding address ranges.

The checkpoint is created when the target process is in a stopped state (waiting for continuation signal from the debugger process), which ensures the state of the process does not change during the creation of the checkpoint.

Restoring the checkpoints

Upon issuing a checkpoint restore operation to a debugger node, the restoration process is carried out, for which the most significant steps are the following:

- the register values stored with the checkpoint are written into process registers;
- the checkpoint file is read, and each memory region in the file is written at the designated address range stored with the region, restoring the recorder state of value-space of the target.

This checkpointing approach might be described as partial checkpoint/restore. In the classical checkpoint/restore, the restore operation includes the creation of a new process from the recorded checkpoint, however in this implementation, the original process is preserved throughout the restore operations. This aspect is in fact very favourable in the context of the debugger because of many aspects. Primarily, it enables maintaining a consistent Linux environment throughout the process lifecycle, meaning the process ids of the nodes and their parent-child relationships are able to remain constant, which is important for the internal process health monitoring of the MPI runtime. Additionally, analogously to the MANA checkpointing approach in [39], this checkpointing approach dispenses with recording data about MPI internals, thus reducing the footprint of the checkpoints and enabling interoperability across different MPI runtime implementations. It is necessary to mention that the checkpoint restore solution at the time of writing this thesis is not exhaustive, as it fails to capture all attributes of a process that should ideally be restored, including some process memory areas, opened files, etc. It is sufficient however to demonstrate its viability for this use case, and is able to restore the value-space of the MPI programs included as examples in the debugger source code (see appendix I).

4.2.3 Support for Causal Consistency

In order to maintain causal consistency as described in section 2.2.3, the orchestrator implements a synchronisation mechanism on rollback operations to maintain a globally consistent state in the MPI process. This mechanism is analogous to the Loosely Synchronised Method for Fault-Tolerance as described in [50]. If a checkpoint is to be

rolled back on a particular node, the dependent checkpoints that additionally need to be rolled back are evaluated. This process computes a set of pairs $S = \{(n, c)|n1 \neq n2\}$ where $n \in N$ is a node id and $c \in C_n$ is a checkpoint recorded on node n to which the corresponding node needs to be rolled back to.The pseudo-algorithm for this process is as follows:

Algorithm 1: Determining the list of related checkpoints for rollback

Result: S - list of checkpoints to be rolled back

1 $S \longleftarrow \{(n_p, c_q)\}$; *// c_q is the original checkpoint desired to be rolled back on node node n_p*

2 **for** *each element (n, c) $\in S$* **do**

3 **for** *each checkpoint c_i recorded on node n where $c_i \leq c$ (checkpoint c_i was recorded later than c or $c = c_i$)* **do**

4 *m $\longleftarrow$ $corresponding\ matching\ event$ (n', c'_i) from the checkpoint log (for send-events, find the corresponding receive-event, and vice-versa).*
if *m* **then**

5 **if** $\exists(n', c_j) \in S|c'_i < c_j$ *(c_j is a checkpoint on node n' that is recorded later than c_i)* **then**

6 *Remove (n', c_j) from S;*

7 *Add (n', c'_i) to S;*

8 *Repeat from line 2 until no new entries are added to S on a single iteration.*

After executing this algorithm, S contains a set of checkpoints that need to be rolled back to maintain a global causally consistent state in the MPI job. Figure 8 illustrates a scenario where the dependent checkpoints are determined from the initial checkpoint desired to be rolled back.

To the best of the author's knowledge, this is the first reversible debugger for MPI implementing this mechanism for maintaining causal consistency on reverse debugging.

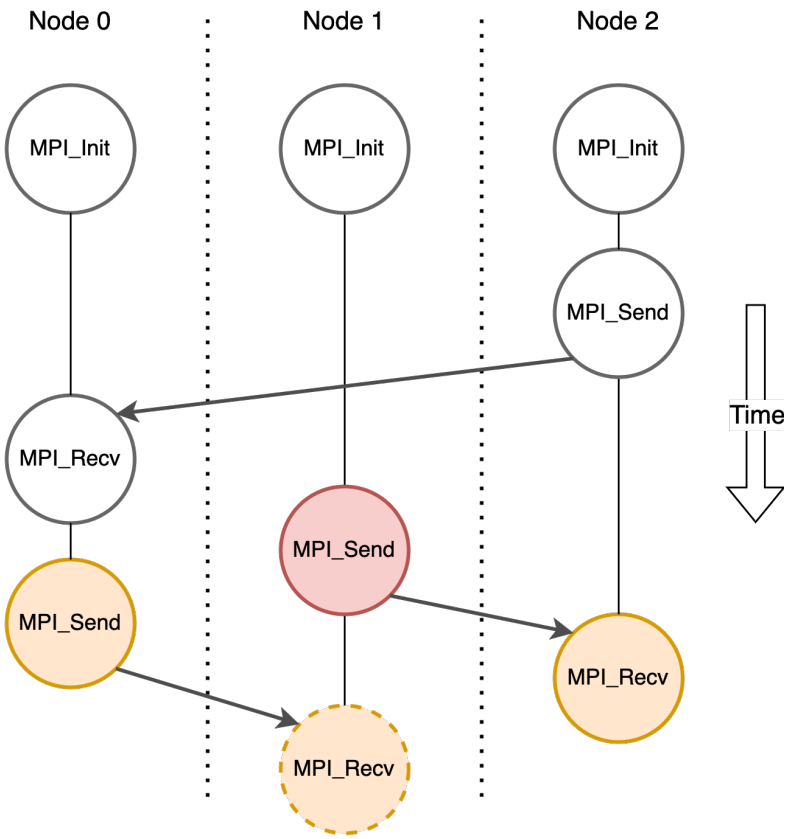

Figure 8. Determining the dependent checkpoints to be rolled back. Red - the original checkpoint to be rolled back. Orange - dependent checkpoints to be rolled back as well.

5 Discussion

This chapter aims to give a qualitative discussion on notable aspects of the debugger that either manifest in the current version of the tool, or are not present in the current version but are reasonable additions that should be included. As the scope of implementing such a tool in a fully usable manner for practical everyday use is very large, all of this work did not fit into the scope of this thesis. This chapter aims to give a descriptive base of the additional scope that should aspirationally be incorporated into the debugger in future efforts.

As described, maintaining causal consistency is integral to the functionality of the presented debugger. A question might be posed whether the ability to maintain causal consistency is always a necessary property for a parallel debugger. The conclusion would be that it depends on the particularities of the tool – for the debugger presented in this thesis, this property is crucial, because the debugger uses non-stubbed re-execution of the target after restore operations, meaning when a message sending event is replayed multiple times, the message is actually in fact transmitted multiple times. If this was not the case and the subsequently executed run-times were replayed without actually transmitting messages more than once (a *replay* mechanism in contrast to *re-execution*), maintaining causal consistency would not be necessarily needed, at least not for preserving the integrity of the communication process. However in addition to supporting the re-execution approach, a causally consistent workflow throughout the debugging session appears intuitive from the user point of view, as the debugged state is an actually viable state of events in the target at all times, based on the causal consistency model.

The approach taken by the debugger is to handle tracing of MPI operations on the user-application layer without interfacing with the MPI runtime directly. The apparent benefit of this is the portability across different MPI implementations, as the presence of a standardised MPI library is the only assumption the debugger makes. However this approach can have shortcomings – for example, it lacks a mechanism for handling cases where the user-designed MPI communication is inherently inconsistent and redundant messages are produced. Rectifying such cases on reverse actions might require the functionality of flushing the redundant messages currently in transmission within the MPI runtime. For this functionality, it would be necessary to interface with the MPI runtime directly. This aspect relates to the design decision of where the boundary of the debugged application is placed. With the presented debugger, the boundary is directly between the user application and the MPI runtime library, whereas for some aspects it might be useful to widen this boundary to include the whole MPI runtime process as well. This change would however have significant consequences to other aspects of the debugger as well.

The checkpoint-restore mechanism incorporated in the debugger differs from this mechanism's generic implementations in a sense that the checkpoint restoration is

applied on live processes (in contrast to bootstrapping new process instances after restore operations), preserving the lifecycle of a process throughout the restore operations. This is highly useful for preserving the lifecycle of the MPI job, and reduces the complexity of the implementation, as fewer aspects of a process's lifecycle need to be handled during restore operations, such as process bootstrapping. This approach, albeit in a prototypal stage, appears to suit well for the purpose of offering reversible debugging features.

5.1 Future Work

As the implementation of this debugger is in a stage that might be characterised as an exploratory prototype of the described functionalities, there are numerous aspects that should be improved in order to produce a tool that is sophisticated and robust enough for practical everyday use. In the following, various aspects of improvement are described that should be implemented in order to achieve this.

Support for more MPI commands – Currently, the debugger supports sample MPI operations for one-to-one message passing (`MPI_Send` and `MPI_Recv`). Support should be added for more commands, including one-to-all (e.g. `MPI_Bcast`) all-to-one (e.g. `MPI_Reduce`) and all-to-all communications, non-blocking communications, etc.

Support for retrieving values for variables for all datatypes – Currently, only a fraction of datatypes are supported with lack of support for complex data types such as structs and arrays.

Support a wider range of processor architectures – In order to generalise the usability of the tool, more processor architectures should be supported in addition to the x86 architecture. The main component requiring an abstraction layer for this purpose is interfacing with the processor registers, for which the usage tends to vary across architectures.

Improved user experience – Currently, the debugger exposes both a command-line interface and a graphical user interface, both of which are needed to conduct a debugging session. If the graphical user interface were to be improved, the debugger could dispense with the CLI and provide a more frictionless user experience. Moreover, the debugger could interface with other software development tools such as *IDE*s (Integrated Development Environments) so that the tool could easily be incorporated into the software development workflow.

Fully exhaustive checkpoint/restore – At the time of writing this thesis, the debugger has limited support for restoring the full memory space of the target, and lacks support for restoring other components such as open files and communication sockets, created/destroyed threads or child processes, opened pipes, etc. In addition, the checkpoint-restore does not address the MPI messages currently in transmission in socket buffers. This may cause either redundant or missing messages to appear after rollback operations.

A multitude of these problems could be solved by incorporating an existing sophisticated checkpointing implementation to handle the checkpointing of a single node. Solutions described in chapter 3.1 might be suitable for this, especially *CRIU* or the *MTCP* component of *DMTCP*.

Support for additional reversible debugging commands – Currently, the reversible functionality exposed by the debugger is the functionality to restore the recorded checkpoints. On top of this functionality, reverse-step and possibly reverse-continue commands could be implemented. This could be achieved by following the checkpoint/re-execute mechanism to produce the desired result based on automatic re-execution from the checkpoint to the desired location.

6 Conclusion

The main scope of this was to implement a debugger for MPI applications capable of performing reversible debugging operations while preserving causal consistency. It was discussed in the thesis, why maintaining causal consistency is an important property for a distributed debugger offering reversible debugging functionality.

The theoretical part of this thesis was dedicated to reviewing the fundamental concepts and mechanisms for building debugger tools, notable existing debugger implementations and solutions for implementing reversibility via checkpoint-restore mechanisms.

The practical part described the implementation of a causally consistent reversible debugger for MPI applications. The debugger features a distributed architecture that maps to the structure of the parallel MPI runtime and incorporates independent checkpointing and coordinated restore mechanisms to offer reversible debugging functionality. To the best of the author's knowledge, this is the first reversible debugger for MPI implementing this mechanism to provide causally consistent reversible debugging.

The experimental implementation demonstrates that this approach is suitable for implementing a reversible debugging tool for parallel computational models. To that end, it is reasonable to develop the described debugger into a state of maturity where it can be used as a practical tool in parallel software development processes, as a future effort.

References

[1] Ivan Lanese et al. “CauDEr: A Causal-Consistent Reversible Debugger for Erlang”. In: Jan. 2018, pp. 247–263. DOI: 10.1007/978-3-319-90686-7_16.

[2] Nicholas Nethercote and Julian Seward. “Valgrind A Program Supervision Framework”. In: vol. 42. June 2007, p. 89. DOI: 10.1145/1250734.1250746.

[3] Jakob Engblom. “A review of reverse debugging”. In: *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. 2012, pp. 1–6.

[4] Robbert van Renesse. “On Concurrent Programming in Harmony”. In: *Harmony Programming Language* (). URL: https://harmony.cs.cornell.edu/docs/textbook/ (visited on 20/04/2022).

[5] R. Landauer. “Irreversibility and Heat Generation in the Computing Process”. In: *IBM Journal of Research and Development* 5 (1961), pp. 183–191. DOI: 10.1147/rd.53.0183.

[6] Charles H. Bennett. “Logical Reversibility of Computation”. In: *IBM Journal of Research and Development* 17 (1973), pp. 525–532.

[7] Edward Fredkin and Tommaso Toffoli. “Conservative logic”. In: *International Journal of Theoretical Physics* 21 (2002), pp. 219–253.

[8] George B. Leeman. “A Formal Approach to Undo Operations in Programming Languages”. In: *ACM Trans. Program. Lang. Syst.* 8.1 (Jan. 1986), pp. 50–87. DOI: 10.1145/5001.5005.

[9] Tetsuo Yokoyama and Robert Glück. “A reversible programming language and its invertible self-interpreter”. In: *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. PEPM ’07. Nice, France: Association for Computing Machinery, Jan. 2007, pp. 144–153. DOI: 10.1145/1244381.1244404.

[10] Tommaso Toffoli. “Reversible computing”. In: *Automata, Languages and Programming*. Ed. by Jaco de Bakker and Jan van Leeuwen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 632–644.

[11] Thomas Herault and Yves Robert. *Fault-Tolerance Techniques for High-Performance Computing*. 1st. Springer Publishing Company, Incorporated, 2015.

[12] B. Bhargava and Shu-Renn Lian. “Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach”. In: *Proceedings [1988] Seventh Symposium on Reliable Distributed Systems*. 1988, pp. 3–12. DOI: 10.1109/RELDIS.1988.25775.

[13] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. USA: Cambridge University Press, 2011.

[14] Robert O'Callahan et al. "Engineering Record And Replay For Deployability: Extended Technical Report". In: *CoRR* abs/1705.05937 (2017). arXiv: 1705.05937.

[15] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems, 3rd Edition*. CreateSpace, 2017.

[16] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[17] William Gropp. "MPI at Exascale: Challenges for Data Structures and Algorithms". In: *16th European PVM/MPI Users' Group Meeting* (Sept. 2009). DOI: 10.1007/978-3-642-03770-2_3.

[18] *MPI: A Message-Passing Interface Standard. Version 4.0*. Message Passing Interface Forum. 2021.

[19] Lisandro Dalcin and Yao-Lung L. Fang. "mpi4py: Status Update After 12 Years of Development". In: *Computing in Science Engineering* 23.4 (2021), pp. 47–54. DOI: 10.1109/MCSE.2021.3083216.

[20] Frank Nielsen. "Introduction to MPI: The Message Passing Interface". In: *Introduction to HPC with MPI for Data Science*. Feb. 2016, pp. 21–62. DOI: 10.1007/978-3-319-21903-5_2.

[21] *TotalView Debugger*. Perforce Software. URL: https://totalview.io/ (visited on 12/02/2022).

[22] *ARM DDT Debugger*. URL: https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt (visited on 11/03/2022).

[23] OpenMPI. *Debugging applications in parallel*. URL: https://www.open-mpi.org/faq/?category=debugging (visited on 11/03/2022).

[24] Michael Kerrisk. *ptrace(2) — Linux manual page*. URL: https://man7.org/linux/man-pages/man2/ptrace.2.html (visited on 09/01/2022).

[25] Jon Ashburn. "Hw-breakpoint: shared debugging registers". In: *LWN* (2009). URL: https://lwn.net/Articles/353050/ (visited on 15/07/2022).

[26] P. Krishnan. "Hardware Breakpoint (or watchpoint) usage in Linux Kernel". In: *IBM Linux Technology Centre* (2010). URL: https://www.kernel.org/doc/ols/2009/ols2009-pages-149-158.pdf (visited on 12/07/2022).

[27] Qin Zhao et al. “How to Do a Million Watchpoints: Efficient Debugging Using Dynamic Instrumentation”. In: *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*. CC’08/ETAPS’08. Budapest, Hungary: Springer-Verlag, 2008, pp. 147–162.

[28] Robert O’Callahan et al. “Engineering Record and Replay for Deployability”. In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC ’17. Santa Clara, CA, USA: USENIX Association, 2017, pp. 377–389.

[29] Linus Torvalds. *Linux-0.97.3*. Tech. rep. Linux kernel historic tree, 1992. URL: https://git.kernel.org/pub/scm/linux/kernel/git/history/history.git/commit/?h=0.97.3.

[30] DWARF Standards Committee. “DWARF Version 5 Standard Released”. In: *The DWARF Debugging Standard* (2017).

[31] DWARF Debugging Information Format Committee. *DWARF Debugging Information Format, Version 5*. 2017.

[32] Jason Duell. “The design and implementation of Berkeley Lab’s linux checkpoint/restart”. In: *Lawrence Berkeley National Laboratory* (2005).

[33] *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 18/04/2022).

[34] *CRIU - Checkpoint/Restore In Userspace*. URL: https://criu.org (visited on 18/04/2022).

[35] Adrian Reber. “Checkpoint and restore in Kubernetes”. 2021.

[36] Adrian Reber and Peter Vaterlein. “Checkpoint/Restore in User-Space with Open MPI”. In: *Symposium on Information and Communication Systems* (2014).

[37] Jason Ansel, Kapil Arya and Gene Cooperman. “DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop”. In: *23rd IEEE International Parallel and Distributed Processing Symposium* (Feb. 2007). DOI: 10.1109/IPDPS.2009.5161063.

[38] Michael Rieker, Jason Ansel and Gene Cooperman. “Transparent User-Level Checkpointing for the Native Posix Thread Library for Linux.” In: Jan. 2006, pp. 492–498.

[39] Rohan Garg, Gregory Price and Gene Cooperman. “MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing”. In: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’19. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 49–60. DOI: 10.1145/3307681.3325962.

[40] GDB. *GDB: The GNU Project Debugger*. URL: https://www.sourceware.org/gdb/ (visited on 05/01/2022).

[41] GDB. *GDB and Reverse Debugging*. URL: https://www.sourceware.org/gdb/news/reversible.html (visited on 05/01/2022).

[42] Ana-Maria Visan et al. “URDB: A Universal Reversible Debugger Based on Decomposing Debugging Histories”. In: *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. PLOS ’11. Cascais, Portugal: Association for Computing Machinery, 2011. DOI: 10.1145/2039239.2039251.

[43] Robert O’Callahan et al. “Lightweight User-Space Record And Replay”. In: *CoRR* abs/1610.02144 (2016). arXiv: 1610.02144.

[44] Robert O’Callahan. “rr 4.0 Released With Reverse Execution”. In: (2015). URL: https://robert.ocallahan.org/2015/10/rr-40-released-with-reverse-execution.html.

[45] Chris Gottbrath. “Reverse Debugging with the TotalView Debugger”. In: *Proceedings of the Cray User Group Conference* (2008).

[46] UndoDB. *UndoDB - Time Travel Debugging for C/C++ and Java*. Tech. rep. URL: https://undo.io/ (visited on 09/04/2022).

[47] UndoDB. *What is Reverse Debugging and Why Do We Need It?* Tech. rep. URL: https://undo.io/resources/reverse-debugging-whitepaper/ (visited on 09/04/2022).

[48] *The Go programming language*. URL: https://go.dev/ (visited on 20/06/2022).

[49] *rpc package - Go packages*. URL: https://pkg.go.dev/net/rpc (visited on 20/06/2022).

[50] Dwight Joe, David Glasco and Michael Flynn. *Fault Tolerance: Methods Of Rollback Recovery*. Feb. 1970.

Appendix

I. Source Code

The source code of the implemented debugger is available at the git repository located at: `https://github.com/ottmartens/cc-rev-db`. The repository includes instructions on how to set up and run the debugger, as well as example MPI programs for debugging.

II. Recordings of Debugging Sessions

The debugger source code includes example MPI programs for which demonstrational debugging sessions are recorded. The recordings are available at the following link: `https://drive.google.com/drive/folders/1juo1BQPlmrf0pe1GRoEgo-EVFv9I8X7G?usp=sharing`

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Ott-Kaarel Martens**,

1. herewith 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,

   Causally Consistent Reversible Debugger for MPI Applications,

   supervised by Eero Vainikko and Stefan Hermann Kuhn.

2. I grant the University of Tartu a permit to make the work specified in p. 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 3.0, 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 p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Ott-Kaarel Martens
08/08/2022