Reversibility in Message-Passing Systems

Ott-Kaarel Martens
Institute of Computer Science
University of Tartu

Eero Vainikko
Institute of Computer Science
University of Tartu
supervisor

Stefan Kuhn
Institute of Computer Science
University of Tartu
supervisor

Abstract—**Reversibility in computation refers to the quality of computational processes being time-reversible, meaning that that the dynamics of the process remain well-defined when the sequence of time-states is reversed.**

This report surveys the implementation of reversibility in software of distributed concurrent systems that rely on message passing as the method of communication. More specifically, two application domains are explored: employing reversibility to achieve fault-tolerance in concurrent systems, and implementing reversible debugging in message-passing systems, by looking at existing implementations, and exploring a possibility of implementing a causal-consistent reversible debugger for the Message Passing Interface.

Keywords—**reversibility, message-passing interface, debugging, fault-tolerance, parallel programming**

I. Introduction

The idea of reversibility in computation originates from studies on the thermodynamic properties of computational processes by Robert Landauer in 1961[1]. 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 machnines[2], and conservative logic[3], as well as implementations on software level, like formal approach to an undo operator[4], and the design and implementation of a time-reversible programming language Janus[5].

The possible motivations for introducing reversibility into a software system are manifold, and the list of applications is definitely expanding as the field develops. There are however domains of application where reversibility has already been shown to yield practical benefit. This is also true in concurrent systems, where some form of reversibility has been implemented on multiple occasions for a specific purpose.

The rest of the report is organized as follows. Chapter II gives a short overview of the Message Passing Interface, as the further topics explored in this report are evaluated against usage with MPI. The effort towards achieving *fault-tolerance* in concurrent software systems has had some different applications of reversibility, which will be looked at more closely in chapter III. Another application domain which has shown to benefit from reversibility is *reversible debugging*, which will be explored in chapter IV. In chapter V, the logical reversibility of MPI communication operations is evaluated. Finally, chapter VI explores the possibility of implementing a reversible debugger for operations defined by the Message-Passing Interface.

II. Message-Passing Interface

The Message-Passing Interface (MPI) is a specification that defines a set of library routines for achieving inter-process communication in parallel computing architectures via message-passing. This programming 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 each process[11]. The specification defines the syntax and semantics for the operations, without defining the implementation details. The MPI specification has implementations for languages such as C, C++, and others; and has been adopted widely for parallel computing workloads.

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.[11]

III. Fault-tolerance

Parallel systems with a large number of components are by their nature more susceptible to run-time failures than their serial counterparts. This is due to the fact that the probability of a single component failure increases rapidly as we add more components to the system. For example, if 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 [8]. Moreover, the communication between the components adds an additional point of failure.

A primary goal when ensuring fault-tolerance in a parallel system is the introduction of mechanisms that prevent a single component failure to cascade into a complete failure of the system. A failure needs to be isolated and mitigated, which can be done spatially (isolating the failed component from the rest of the system), temporally (rolling the system back to last failure-free state), or as a combination of both.

A common approach for failure mitigation in 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 a failure-free state. Checkpointing in a parallel system can be implemented in different configurations:

- *Level* - a checkpoint might be initiated on system-level (by the underlying os/runtime), by the communication library, or explicitly by the user.
- *Coordination* - checkpointing might be done in a coordinated manner (having all processes record their state at the same time), or, to some degree, in an uncoordinated/independent manner (since the cost of coordination is high).
- *Synchronisation* - checkpointing can be implemented as a blocking operation, meaning no process activity happens during the checkpoint, or asynchronously (for example by using the fork and copy-on-write mechanism) during the process activity. [8]

The goal of the checkpoints is their usage within a **rollback** operation. If an error occurs, rollback allows the system to restore a past state which was correct, and restart the computation from there. In a concurrent system, identifying a global past state is not a straight-forward operation however. Since concurrent systems usually lack a global notion of time and each unit in the system maintains their own local state, it is not obvious how to determine a single global state for the localities [10]. The approach usually followed in this scenario is *causal consistency* - since the components in the system interact, they have causal effects on each others behaviour. These causal links can be identified, so that the causal consistency can be maintained when some part of the system is rolled back to a previous state. Therefore *causal-consistent rollback* is defined as the minimal causal-consistent sequence of backward steps able to undo a given action[10]. Figure 1 depicts an example of communicating processes, where process a has a potentially causal effect on processes b and c, and therefore for executing a rollback on process a, all processes must be rolled back to maintain the globally coherent state.

The rollback mechanism can be considered as an implementation of reversibility, as the process relies on stepping backwards in the flow of program execution. Because of the state-saving via checkpoints, the program need not be reversible in the strict sense to perform rollbacks.

Fault-tolerance in MPI

For isolating a single component on failure in MPI applications, relevant work has been done by the User Level Failure Mitigation (ULFM) working group [9]. The working group has developed a proposal, which is a set of extensions to the standard MPI specification for mitigating failures as they happen. More specifically, the proposal introduces:

- new error codes, such as:
  - MPIX_ERR_PROC_FAILED - *when a process failure prevents the completion of an MPI operation*
  - MPIX_ERR_COMM_REVOKED - *when one of the ranks in the application has invoked the MPI_Comm_revoke operation on the communicator*
- new interfaces for mitigating failures, such as:
  - MPIX_Comm_revoke - *interrupts all pending communications on a communicator object*
  - MPIX_Comm_shrink - *cretes a new communicator where dead processes in comm were removed*
  - MPIX_Comm_failure_get_acked - *obtains the group of currently acknowledged failed processes*

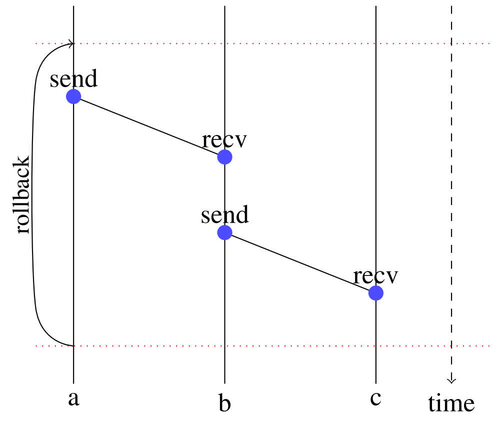

Fig. 1: A causally consistent rollback implies rolling back all causally connected processes

The ULFM proposal does not provide a direct solution for failure mitigation, but rather the interface for the user to define their own implementation of fault-tolerance. For example, the process failure isolation can be combined with user-level checkpointing, so that the failed processes are restored independently. For certain computation patterns there is no need for complex recovery schemes, such as the manager-worker model, where the restore mechanism is as simple as spawning a new worker and dispatching the failed task to a worker for the second time.

In addition to the checkpoint/rollback mechanism, there are other alternatives for ensuring fault-tolerance in an MPI application. For example, replication of tasks is a suitable model for some workloads, as well as algorithm-specific solutions that often show better performance than the resource-costly checkpointing approach[8].

IV. Reversible debugging

Debugging is a practice external to the production run-time of an application, but can still be considered an integral part to the lifecycle of a program. Traditional debuggers support operations such as breakpoints, incremental stepping through the program and accessing the process stacks. Reverse debuggers build on top of that functionality by enabling stepping backwards in the execution of the program.

Many operations that a generic programming language supports are not *backwards-deterministic* (it is not possible to deduct the inputs of an operation from the outputs of an operation). Furthermore, implementing the true reverting of operations without saving previous states of the program execution would require support for this from the underlying run-time environment of the program. As common program languages/run-times do not support stepping backwards in the execution, a reversible debugger needs to supply this functionality in some external way. A common approach for this is *process recording* - as the program is executing in the forward direction, the execution is recorded. This recording can in turn be replayed backwards to essentially mimic backwards execution.

There exist multiple implementations of reversible debuggers. One such implementation is *GDB* - the GNU Project debugger. GDB supports a variety of programming languages and runs on most UNIX-like systems. The support for reversible debugging was added in version 7 in 2009[6]. The reversible debugging is implemented on top of process record/replay functionality - when recording is started, each step of the target process is recorded, and reverse-stepping in the debugger acts by "replaying" the recorded actions in reverse.

A similar recording approach is employed by *rr project* originally developed by Mozilla. rr stands for record-replay, as the tool enables the user to record any Linux process as it is executing, after which the recording can be deterministically replayed using the gdb debugger. The recording is implemented by monitoring the interactions between a Linux process group and the kernel, as well as recording non-deterministic CPU effects. For the replay mechanism, rr acts as a gdb-server, to which a gdb debugger can be connected to for both forward- and backward-replay of the execution.[13]

Reverse debugging in parallel systems

Debugging a parallel program can be done either by debugging the individual processes, or employing a solution which captures the execution of the whole system composing of many processes. In the second approach, the debugger can be used to gain insight into the global state of the parallel application. To enable more flexibility, such a debugger might support reverse debugging as well. However, when introducing reverse debugging into a parallel debugger, the notion of causal consistency becomes relevant. Consider the case in figure 1 - if we were to use reverse debugging on this execution and stepped backwards in process a along the arrow indicating the rollback without any constraints/side-effects, we would end up in a state which is not causally consistent, as process a exists in the state where it is yet to send a message, but process b has already received this message, as well as passed a potentially causally-related message forward to process c. To avoid this kind of global inconsistency, the debugger must follow the principle of causal consistency by either restricting the user actions that would violate it, or by executing side-effects that maintain the causal consistency (in the example above, processes b and c could be reverted back to the last previous states preceding the exchange of messages).

CauDEr

One notable approach for employing reversible debugging in a parallel computing model with message-passing communication paradigm is done in implementation of CauDEr - A Causal-Consistent Reversible Debugger for Erlang developed by Lanese et al. in [12].

Causal consistency is one of the key features of this debugger. A precondition for the implementation is the development of causal-consistent reversible semantics for the Erlang language. More specifically, the authors first outline a formal model for describing the state of an Erlang program, with emphasis on the message-passing mechanism between processes. This model enables the authors to define formal semantics for rollback operations, that are employed for enabling reversibility in the debugger. The rollback semantics are defined for a single step back, as well as rollbacks up to a certain event. It is important that the rollback operations maintain the causal consistency in the global system state.[12]

Arguably, not all parallel reverse debuggers need to maintain the strict principle of causal consistency. Some common debugging workflows might focus on analyzing the behaviour of a particular node in the parallel system without the need to maintain the global causal coherence at all times. The need for such tools is demonstrated by existing commercial implementations, one of the most prevalent of these being the TotalView HPC debugger. TotalView debugger is developed for the specific purpose of parallel debugging, and employs a ReplayEngine module for reverse debugging via process recording. The Totalview debugger has wide support for different parallel programming paradigms, including MPI applications.[14]

V. Logical reversibility of MPI operations

It is worth exploring to what degree are the operations defined by the MPI specification logically reversible. As the interface defines mostly communication operations, it makes sense to consider sending and receiving a message together as a single operation for this case. A communication operation can be considered logically reversible if the input on the sending process(-es) can be deterministically deducted from the output on receiving process(-es).

Table I describes which common MPI operations can be considered logically reversible. Detailed descriptions of the MPI operations are not outlined in this report, but can be examined in the MPI standard in [11].

An MPI_SEND matched with a corresponding MPI_RECV is logically reversible, if the RECV operation is executed with an explicitly defined $source$ parameter specifying the sending process. Otherwise, it is not possible to deduct which process the message originates from, unless the $status$ parameter is used on the receiving process to capture the source.

MPI_BCAST broadcasts the same message to all processes on the communicator, hence the operation is always reversible (as the input is the same as the output payload). MPI_SCATTER and MPI_GATHER either respectively divide

send operation	receive operation	logically reversible
MPI_SEND	MPI_RECV(#)	yes
MPI_SEND	MPI_RECV(⋆)	no*
MPI_BCAST		yes
MPI_SCATTER		yes
MPI_GATHER		yes
MPI_REDUCE		no*

symbol	definition
#	receive from process with explicit rank
⋆	receive from any process (MPI_ANY_SOURCE)

TABLE I: Logical reversibility of MPI communication operations

or concatenate the payload array, but do so in a deterministic ordering given by the process ranks, hence the operation is logically reversible.

Reversibility of the MPI_REDUCE operation depends on the exact reduce function used, but reducing an array of values into a single value is usually not backwards-deterministic, hence this operation can be considered logically irreversible.

Single send and receive operations without a matching counterpart can be always considered reversible, as the operation will remain in a pending state until the message is sent/received by the other process and thus have no effect on the state of the process.

Whether the communication is carried out in a blocking or non-blocking manner does not affect the reversibility of the operation. However non-blocking operations that execute in the background (while the control flow of the program continues) exist in an ambiguous state of completion from the control flow viewpoint, before explicit checks are made to verify/wait for the completion of the operation.

VI. Towards causal-consistent reversible debugging in MPI

The prevalence of MPI as a tool for implementing distributed-memory parallel applications provides reasoning for implementing a causal-consistent debugger for MPI. The ongoing academic and commerical efforts such as the ones described above for similar goals validate that implementation of these kinds of tools can exhibit practical benefit. The possible implementation requires solving multiple non-trivial challenges, hence this chapter outlines some foreseeable efforts towards implementing such a tool.

The efforts towards the implementation of the causal-consistent reversible debugger for Erlang described in IV present a possible approach for solving this problem at on the theoretical side. Similarly to the work done in [12], the implementation of global causal consistency requires the modeling of the global system state with emphasis on the inter-process communication via messages. As is done for the Erlang debugger, a global ”mailbox” can be considered for managing the state of messages in the system. This enables the semantics to be defined for both forward execution of the message-passing operations, as well as rollback operations for reversing the defined forward-operations.

A parallel debugger can be implemented either as a group of standalone processes that each connect to a specific process in the MPI application, or as a singleton process which is bound to all MPI processes. With the first approach, the standalone debugger processes would need to be connected to a central process which would orchestrate the execution of the individual debuggers. This kind of implementation has a benefit of separated concerns, as each process handles a concise scope of functionality, but on the other hand the orchestration of the group of debugger processes introduces additional overhead in terms of system complexity and quite probably in terms of performance. As for the second approach, a singleton process bound to all debugged processes would be architecturally simpler, but managing a high number of debugged processes simultaneously is bound to introduce high complexity as well.

For the debugger process to access the contents of the MPI message queues, work has been done in [15] for describing an interface that the TotalView debugger uses for accessing the MPI message queues.

As for locating all the MPI processes by a debugging tool, [15] describes a solution based on proctables that can be used for this purpose. In [16], an alternative interface is proposed for supporting identification of dynamically created MPI processes.

In terms of implementing a debugger, it can be considered to extend an existing debugger with functionality that would support causal-consistent debugging of MPI applications, instead of developing a debugger from scratch. One possibility of partial existing debugger reuse would be to implement the debugger only on the level of the debugger server, and use an existing debugger client for the user interface.

As for the requirement of being able to maintain causal consistency, the debugging tool must be able to identify the inter-process communication to establish the causal links between the processes. This task is non-trivial and cannot be solved by static analysis of the source code alone - for example, the parties doing communication are often evaluated dynamically during the program run-time, and hence are not known prior to the execution. This means that the debugger application needs to actively monitor the communications happening between the processes to establish causal links between the communicating parties.

VII. Conclusion

In this report, the application of the principle of reversibility in parallel message-passing systems was explored. The advances towards enabling fault-tolerance with the help of reversibility were surveyed, with focus on MPI-based systems, as well as state-of-the-art reversible debugging projects of both academic and commercial scopes. These domains are actively developing and the principles of reversibility have been shown to yield practical benefit. In the second half of the report, the inherent reversibility of MPI operations was explored, and some introductory and exploratory reasoning was aimed at implementing a causally-consistent reversible debugger for

MPI applications. Though there are numerous challenges to be solved for the implementing such a tool, it appears to be a feasible endeavour.

REFERENCES

[1] Landauer, R.: Irreversibility and heat generated in the computing process. IBM J.Res. Dev.5, 183–191 (1961)

[2] Bennett, C.H.: Logical reversibility of computation. IBM J. Res. Dev.17(6), 525–532 (1973)

[3] Fredkin, E., Toffoli, T.: Conservative logic. Int. J. Theor. Phys.21, 219–253 (1982)

[4] Leeman Jr., G.B.: A formal approach to undo operations in programming languages. ACM Trans. Program. Lang. Syst.8(1), 50–87 (1986)

[5] Yokoyama, T., Glück, R. A reversible programming language and its invertible self-interpreter. ACM SIGPLAN, 144–153. (2007)

[6] GDB and Reverse Debugging. GDB, the GNU Project debugger. https://www.gnu.org/software/gdb/news/reversible.html

[7] Francalanza, A., Mezzina, C.A., Tuosto, E.: Reversible choreographies via monitoring in Erlang. In: Bonomi, S., Rivi'ere, E. (eds.) DAIS 2018. LNCS, vol. 10853,pp. 75–92. Springer, Cham (2018).https://doi.org/10.1007/978-3-319-93767-06

[8] Bosilca, G., Bouteiller, A., Herault T., Robert, Y.: Fault-tolerance Techniques for HPC. The International Conference for High Performance Computing, Networking, Storage, and Analysis. (2021) https://fault-tolerance.org/2021/11/12/sc21-tutorial/

[9] Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.J.: Post-failure recovery of MPI communication capability: Design and rationale. International Journal of High Performance Computing Applications. August 2013 27: 244-254, doi:10.1177/109434201348823

[10] Giachino, E., Lanese, I., Mezzina, C. A., Tiezzi, F.: Causal-consistent rollback in a tuple-based language. Journal of Logical and Algebraic Methods in Programming. Elsevier. 2017. 88, pp.99 - 120. ff10.1016/j.jlamp.2016.09.003ff. ffhal-01633260

[11] MPI: A Message-Passing Interface Standard (Version 4.0). Message Passing Interface Forum. June 2021. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[12] Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: a causal-consistent reversible debugger for Erlang. In: Gallagher, J.P., Sulzmann, M. (eds.) FLOPS2018. LNCS, vol. 10818, pp. 247–263. Springer, Cham. 2018. https://doi.org/10.1007/978-3-319-90686-716

[13] O'Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A., Partush, N. Engineering record and replay for deployability. In Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17). USENIX Association, USA, 377–389. 2017.

[14] Totalview debugger. Perforce Software. https://totalview.io/

[15] MPI debugging interface. MCS Division, Argonne National Laboratory. https://www.mcs.anl.gov/research/projects/mpi/mpi-debug/

[16] Gottbrath, C., Barrett, B., Gropp, W., Lusk, E., Squyres, J. An Interface to Support the Identification of Dynamic MPI 2 Processes for Scalable Parallel Debugging. 2006. 115-122. 10.1007/11846802_22.