POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A journey towards transparent fault tolerance in embarrassingly parallel MPI applications

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Luca Repetti**

Student ID: 945135
Advisor: Prof. Gianluca Palermo
Co-advisors: Roberto Rocco
Academic Year: 2021-2022

Abstract

High-Performance Computing (HPC) is driving innovation in multiple fields and reaching exascale performance, even if fault tolerance represents an obstacle to its growth. The most common interconnection medium, MPI, does not yet provide reliable assumptions to continue execution after a fault. Recent efforts from the MPI Forum aim to address this need for the next generation of applications, but they still require in-depth knowledge of MPI to achieve resiliency to faults. Fault tolerance can be implemented with different methods, such as checkpoint/restart, but the existing frameworks still lack transparency and are not easy to integrate.
This work tries to understand the issues preventing MPI from offering a transparent interface to recover from faults, starting the journey from the ambitious goal of providing a completely transparent fault recovery mechanism through C/R in generic applications by hiding faults from the application and re-creating failed nodes. Due to the complexity of low-level memory management in distributed systems and the lack of support for MPI of state-of-the-art transparent checkpoint frameworks, the initial transparency goal was relaxed. We propose a transparent fault recovery framework to enable MPI to automatically recover from failures of critical processes and continue the execution after non-critical failures. We build our work on top of the User-Level Fault Mitigation (ULFM) library and Legio, a resiliency library. We distinguish between critical and non-critical processes, ensuring that only the ones crucial to the completion of the application are restarted to lower the overhead of failures.
We tested the work on a supercomputer, proving that the overhead is negligible compared to the loss of a critical rank. Finally, we discussed further evolutions of the work, which could leverage the upcoming MPI 5.0 Standard and better dynamic process management runtimes for MPI.

Keywords: MPI, ULFM, fault tolerance, checkpoint, high performance computing

Abstract in lingua italiana

Il calcolo ad alte prestazioni sta guidando l'innovazione in diversi campi e raggiungendo exascale performance, ma la tolleranza ai guasti rappresenta un ostacolo limitante per la sua crescita. Il metodo di comunicazione più comune, MPI, non fornisce ancora garanzie affidabili per continuare l'esecuzione dopo un errore. I recenti sforzi dell'MPI Forum mirano a soddisfare questa esigenza per la prossima generazione di applicazioni, ma attualmente richiedono ancora una conoscenza approfondita di MPI per tollerare fallimenti in applicazioni esistenti. La tolleranza ai guasti può essere implementata con diversi metodi, come il checkpoint/restart, ma i framework esistenti non sono ancora trasparenti nell'utilizzo o facili da integrare.

Questo lavoro esplora i problemi che impediscono a MPI di offrire un'interfaccia trasparente per il recupero dagli errori, partendo dall'ambizioso obiettivo di fornire un meccanismo di recupero trasparente attraverso C/R in applicazioni parallele, nascondendo i guasti e ricreando i nodi falliti. A causa della complessità della gestione della memoria a basso livello nei sistemi distribuiti e della mancanza di supporto per MPI dei framework di checkpoint trasparenti più avanzati, l'obiettivo iniziale di trasparenza è stato attenuato. Proponiamo un framework di recovery che permetta ad MPI di gestire fallimenti di processi critici, e continuare l'esecuzione durante guasti non critici. Costruiamo il framework sopra User-Level Fault Mitigation Library (ULFM) e Legio, una libreria per la tolleranza ai guasti. Il risultato distingue automaticamente nodi critici e non, creando overhead solamente per i processi cruciali al completamento dell'applicazione.

Abbiamo misurato l'overhead su un supercomputer, dimostrando che è trascurabile se paragonato alla perdita di un processo critico. Abbiamo poi discusso evoluzioni della ricerca che potrebbero sfruttare lo standard MPI 5.0 e migliorie nella gestione dinamica dei processi per MPI.

Parole chiave: MPI, ULFM, tolleranza agli errori, checkpoint, calcolo ad altre prestazioni

Contents

Abstract i

Abstract in lingua italiana iii

Contents v

1 Introduction 1

2 Background 5

2.1 Message Passing Interface . . . 5
2.1.1 MPI Initialization and Procedures . . . 5
2.1.2 Point-to-point and Collective Communication . . . 6
2.1.3 Group management . . . 8
2.1.4 Communicator management . . . 10
2.1.5 Dynamic Process Management . . . 13
2.2 Fault Tolerance . . . 13
2.2.1 Background . . . 14
2.2.2 MPI Error Handling . . . 15
2.2.3 User-Level Fault Mitigation . . . 15
2.3 Checkpoint/Restart . . . 17
2.3.1 Classification . . . 17
2.3.2 Checkpoint in MPI . . . 19
2.3.3 Limitations . . . 19
2.4 State of The Art . . . 20
2.4.1 Transparent Checkpoint/Restart . . . 20
2.4.2 Other techniques . . . 21
2.4.3 MPI stop-and-restart . . . 22
2.4.4 MPI resiliency . . . 23

3 Transparent fault recovery **27**
3.1 Requirements . . . 27
3.2 Legio - a fault resiliency framework . . . 28
3.2.1 Overview . . . 29
3.2.2 Multiple communicators support . . . 30
3.3 Distributed MultiThreaded Checkpointing . . . 31
3.3.1 DMTCP Overview . . . 31
3.3.2 DMTCP with local restart . . . 33
3.3.3 MANA Plugin . . . 35
3.4 CRIU . . . 36
3.4.1 Internals . . . 36
3.4.2 MPI Process Migration with CRIU . . . 37
3.4.3 Legio with CRIU . . . 38
3.5 Moving to application checkpointing . . . 40

4 Application checkpointing **43**
4.1 Overview . . . 43
4.1.1 MPI application checkpointing . . . 43
4.1.2 High-level design . . . 44
4.2 Initialization phase . . . 45
4.2.1 MPI_Init . . . 45
4.2.2 Communicators . . . 46
4.3 Failure detection phase . . . 48
4.4 Restart phase . . . 50
4.4.1 Assumptions . . . 50
4.4.2 Resiliency and Recovery . . . 51
4.4.3 Restarted process and MPI structures . . . 52
4.5 Known limitations . . . 52

5 Experimental Evaluation **55**
5.1 Experimental setup . . . 55
5.2 Restart . . . 56
5.3 Parallel application . . . 59
5.4 Conclusion . . . 61

6 Future Work and Conclusions **63**
6.1 Transparent checkpointing next steps . . . 63
6.2 Application checkpointing next steps . . . 64

6.3 Conclusions 65

Bibliography **67**

List of Figures **71**

Acknowledgements **73**

1 | Introduction

High-Performance Computing (HPC) is the field of computer science where supercomputers and large clusters solve advanced computation problems. Given the increasing computational requirements, both in academia and industry, HPC has been at the latest years at the heart of computer development.

The increased complexity and problem size in various domains called for huge improvements in systems' performances. To keep up with the demand, the capabilities of these computing systems must grow exponentially faster than Moore's law. To quantify these conditions, it is possible to reference exascale computing, which refers to a system having a minimum computing power 10^{18} floating point operations per second (FLOPS), or one exaFLOPS. The achievement of such performance is having a transformative impact on a wide range of applications, ranging from weather forecasting and physical simulation to new drug discovery, simulating health risks before human or animal testing.

The biggest challenges stand in upgrading existing platforms [10]. There is active research for a computing paradigm model that is flexible enough to port existing workloads and an interface that abstracts low-level network details, enabling low-latency interconnection between the different nodes. Considering the length of the workloads running on the platforms and the number of components involved in achieving the desired goal, the likelihood of faults increases substantially. To solve this problem, we must introduce fault tolerance, which enables systems to continue operating even in the event of one or more faults. Strategies for fault tolerance must be researched and plugged in to ensure that a single fault won't jeopardize the work previously done. This thesis explores tolerance to faults and its ties to the communication method used in distributed systems.

The standard process communication paradigm in HPC is the Message Passing Interface (MPI). First released in 1994, it is currently actively maintained by the main contributors in the industry and academia. Although the MPI Forum is the main point of reference for the semantic (API) of function calls exposed to developers, different implementations solve communication problems differently, such as MPICH, OpenMPI, and Intel MPI. Generally, parallel applications have always used MPI in its most basic features, which enable the distribution of work across multiple nodes and a general collection of the results through

collective network operations. The simple paradigm (distribute, process, and collect) has been at the core of the efforts from both the MPI Standard and the MPI implementations cited above. With new paradigms and technologies revolutionizing the computing sphere in HPC, actions must be taken to keep MPI able to satisfy the applications' requirements. In particular, the new upcoming version, MPI 5.0, is creating specific work groups to work on new approaches that will help MPI with the new generation of architectures, such as hybrid and heterogeneous programming (CUDA, OpenCL), alternate concepts for the initialization of MPI enabling more holistic, dynamic process management, and fault tolerance, to define support that enables portable fault tolerance solutions. This work focuses on the process fault tolerance techniques in MPI environments.

According to the MPI standard with versions equal or minor to 4.0, an error in any process inside the MPI application made the subsequent behavior undefined. The reason for this functioning is MPI's simplicity, thanks to which MPI has a low communication cost, but it is now becoming an obstacle. With the increasing number of components, the likelihood of faults grows, and more advanced features are needed. Development for fault management solutions in MPI has been ongoing for years, and recently a dedicated Working Group focused on different fault tolerance solutions.

To better comprehend the concept of fault tolerance, it is useful to distinguish two branches: fault resiliency and fault recovery. When an application is resilient to faults, its execution can continue ignoring the failed processes and produce a result. In some cases, fault resiliency can lead to an approximate result due to the missing contribution of the failed processes. Instead, with fault recovery, we point to techniques that users may employ to recover the execution from a failure. An example of such a technique is checkpoint/restart, where a failure can trigger the restart of the entire application (global restart) or only the failed processes (local restart).

One of MPI's most used libraries that enable fault resiliency and recovery is User-Level Fault Mitigation (ULFM) [7]. Maintained by the MPI Fault Tolerance Working Group, it exposes low-level directives to deal with faults, allowing users to integrate fault tolerance in their applications. Due to the implementation details exposed in ULFM, different frameworks have tried to simplify integration with existing applications by creating all-in-one solutions. In fault resiliency, the focus was on continuing the application with the live processes after a failure without re-computing lost samples [20, 21] or on applying changes to data structures once a failure is detected [27]. In fault recovery, the computation continued either on spare nodes from the latest checkpoints [29] or by adding new processes to the pool by forwarding the checkpoint responsibilities to the user[3].

Another approach is explored from Legio[24], which focuses on transparent fault resiliency in embarrassingly parallel applications. An essential difference from the previously refer-

enced solutions is that it does not require any change to the application code. Its main goal is minimizing the changes for the application developer while allowing transparent fault resiliency. It deals with failed processes by discarding them and continuing the execution of the application. In case the loss is a non-critical node for the application, the correctness of the result may be impacted, but the trade-off could be acceptable in case an approximate result is enough.
This thesis explores transparent fault tolerance, focusing on fault recovery. Existing transparent frameworks such as Legio can bring results only if the failures are in processes that are not critical to the completion of the application. To eliminate this inconvenient assumption, we start the exploration journey with transparent fault recovery techniques, which enable existing applications to be checkpointed and restarted without any change to the application's code, outlining the main difficulties in reaching such results with the tools and frameworks existing today. After, we continue the journey by relaxing our requirements and accepting that the applications' code must be responsible for checkpoint/restart. We then propose an extension to Legio, called Legio++, which enables the reparation of communication structures by automatically restarting critical processes in case of failures. We evaluate the results measuring the overhead and the additional accuracy of results, demonstrating that the impact on performance is limited compared to the trade-off in losing a critical process.
To summarize, the contributions of the thesis are the following:

- We explore existing checkpoint/restart frameworks and attempt to integrate them with MPI and Legio to achieve transparent fault recovery.
- We propose Legio++, an evolution of Legio which transparently repairs and respawn failed critical MPI processes in case of failures.
- We evaluate the overhead of the proposed solution by analyzing the time for a complete restart of a failed process and the completion time of a complete application.

This thesis is structured as follows: Chapter 2 gives the background knowledge needed to fully understand the problem, its solution, and the state-of-the-art of existing frameworks. Chapter 3 covers the first exploration of transparent fault-recovery techniques and the attempts to integrate them with MPI and ULFM. Chapter 4 proposes a framework to recover from failure by abstracting the MPI structures while exposing high-level directives for the developer to be responsible for application checkpointing. Chapter 5 goes through the experimental evaluation of the work by showing the overhead related to the respawn of a failed MPI process and the total overhead in an embarrassingly parallel application. Finally, Chapter 6 wraps up the thesis by showing the possible next steps of this effort.

2 | Background

This chapter introduces the basic concepts on which the following chapters are based. Section 2.1 introduces the Message Passing Interface (MPI) fundamentals. Section 2.2 focuses on MPI fault resiliency and User Level Failure Mitigation (ULFM), while Section 2.3 explores checkpoint techniques to reach fault tolerance. Finally, Section 2.4 summarizes the state-of-the-art and lays the foundations for the exploratory work in Chapters 3 and 4.

2.1. Message Passing Interface

Parallel computing is using multiple computing resources to solve a computational problem. An integral part of the flow in parallel applications is the coordination between different processes, which handle different parts of the task. The Message Passing Interface (MPI) [2] aims to empower communications in a parallel environment.
MPI is just a specification: the standard abstracts core concepts able to empower highly complex communication models. At the MPI interface's basis is the MPI operation - a sequence of steps that enable data transfer and synchronizations between different processes. To execute an MPI operation and facilitate coordination, communicators are used. They encapsulate the context of communication between a group of processes, which are identified uniquely inside the communicator through their rank, starting from 0.
The section is structured as follows: Section 2.1.1 digs deeper into MPI initialization and procedures, Section 2.1.2 focuses on point-to-point and collective communication, Section 2.1.3 describes group management, and 2.1.4 summarizes communicators handling. Finally, Section 2.1.5 explains the dynamic process model and the Process Manager Interface.

2.1.1. MPI Initialization and Procedures

The main goal of the MPI Standard is achieving a portable parallel programming model on high-end systems. To achieve this, MPI presents two models to initialize processes while being agnostic to the specific command line or environment setups: the World model, where a set of processes are created as members of a common `MPI_COMM_WORLD`

communicator, and the Session model, where the application itself manages the creation of the MPI groups, which are sets of processes. This thesis assumes the usage of the World model, which is analyzed in the current section.
The first step before interacting with the MPI interface is the invocation of `MPI_Init`; only after the call finished without errors, `MPI_COMM_SELF` and `MPI_COMM_WORLD` will be usable in operations involving communicators. The initialization assigns to each process calling the procedure a rank, starting from 0. Ranks can be used in cooperative algorithms to divide the workload between different processes.
After initializing an MPI program, communication is done through MPI operations, which are implemented as one or more MPI procedures. The main distinction is between local and non-local procedures. Non-local procedures require the cooperation of other MPI processes, while local procedures do not. To fully grasp the content of the thesis, it is also important to define operation-related procedures: they implement one of the four stages of an MPI operation, which are initialization, starting, completion, and freeing. On the contrary, non-operation-related procedures do not implement any of the four stages for MPI initialization, such as `MPI_Comm_rank`. Finally, collective procedures require all processes in the target groups to invoke the procedure. A synchronizing collective procedure, such as `MPI_Barrier`, will return only after all processes have called the matching MPI procedure. In case one or more processes belonging to the communicator on which the collective is called do not participate, a deadlock occurs. Consequently, execution cannot continue in all the processes of waiting for the collective operation to conclude.
To exchange messages, MPI sets up communication methods that can be tuned through the run-time command-line, by disabling or enabling the Modular Component Architecture (MCA) components related to the Byte Transfer Layer (BTL). Examples of such components are shared memory, which is available for exchanging messages in processes running in the same node, and TCP, which requires opening sockets to communicate.

2.1.2. Point-to-point and Collective Communication

Point-to-point operations focus on the simplest model of communications: exchanging messages between single processes. Given a communicator, processes are identified with their respective rank in the communicator and can send and receive messages. The simple procedures require a communicator, a rank, and either a full data buffer to send data or one to receive the data, which must have the correct type and size to account for the sent buffer. The operations can be either blocking or non-blocking; the difference is that blocking operations do not return until the completion of the procedure, waiting for the transmission to end. For example, a blocking receive procedure will not return until a

message has been correctly received. On the other hand, a non-blocking procedure will initialize the communication but will not wait for the communication to complete. In this way, the process can perform other work and only periodically check if it is ready for communication.
`MPI_Send` and `MPI_Recv` are the most commonly used pair of operations in blocking point-to-point communication. They allow exchanging of messages between two different processes. When sending messages, it is necessary to pass the data being sent, the number of elements and their type (to correctly initialize the buffer), the rank of the receiving process, a tag, and a communicator. The receive operation is very similar, taking similar parameters: the main difference is that instead of initializing a buffer to send the data, it will receive the data from the corresponding send. The specular non-blocking functions are `MPI_Isend` and `MPI_Irecv`: due to their non-blocking nature, it is possible to proceed with other work while waiting for their completion. Both functions return a `Request` object, which can be used in a `MPI_Wait` to wait for the result of the operation. If all non-blocking operations need to complete simultaneously, the `MPI_Waitall` can be used. Another example of a point-to-point operation is the `MPI_Probe` (`MPI_Iprobe` is the non-blocking version). It obtains information about a message waiting for the reception but does not acknowledge it. Since it just checks for incoming messages, it is unnecessary to know the length of the message beforehand. Similarly to the operations above, it takes the source rank, a tag, a communicator, and the status object, which will contain whether a message has been probed.
Collective operations enable complex communication models by involving all processes within the used communicator and must be issued by all the processes in the given communicator. When considering the usage of collective operations, it is important to analyze the flow to avoid possible deadlocks. It would be incorrect to call a collective operation in an MPI process on a communicator in which other processes would not reach the call, since the program would block. Finally, in a multi-threaded implementation, particular attention must be taken to ensure that the same communicator is not simultaneously used in a collective communication operation that is not thread-safe.
The simplest collective and synchronizing operation is the `MPI_Barrier`, which is useful in cooperative contexts to ensure all ranks reach a predefined step in the flow of the application. Similarly to point-to-point operations, also collectives can be either blocking or non-blocking. For example, `MPI_Ibarrier` is an example of a non-blocking collective operation. Since work can be done while waiting for synchronization, non-blocking operations can save waiting times.
Other important examples of collectives operation are the `MPI_Reduce` and `MPI_Reduceall`. Both operations combine data from multiple processes by reducing it into a single result.

Users can define custom functions to combine the data or use predefined ones (such as MPI_SUM). Both functions take as parameters the pointer to the data to combine, the number of elements and their type, the pointer to the data in which the result will be written, and finally, the type of operation to apply to combine the data. The main difference between the two functions is that MPI_Reduce combines data in a unique process specified in the parameters as root. MPI_Reduceall instead writes the result to all the ranks participating in the operation.

2.1.3. Group management

Groups are used in MPI to create complex interconnections between the various processes in an MPI application. They allow us to divide and conquer the necessary problem, by breaking it down into two or more sub-problems, potentially with completely different procedures run in each group of processes.
In MPI, groups are an ordered set of references to MPI processes. The order is achieved through ranks, representing single application processes. They are represented with a local object which can be freely manipulated with a local procedure since cooperation is not required. The direct consequence is that groups cannot be moved between different processes, and their change is not propagated in the MPI universe. They are the key element enabling the description and evolution of communicators, whose ranks are defined by the related group of the communicator. The local procedures MPI_Group_rank and MPI_Group_size return the rank of the calling process in the group and the size of the group, respectively.
Groups can be constructed starting from a communicator: MPI_Comm_group is a local procedure that extracts the group describing the communicator. Starting from a group, it's also possible to maneuver the process identifiers to create new groups through functions with the MPI_Group prefix, such as the following:

- MPI_Group_incl. It creates a new group from the input one, which contains only the ranks passed as an argument. It is useful if we already know which ranks we want to include in a new group.
- MPI_Group_excl. It creates a new group by deleting ranks from an existing group.
- MPI_Group_union. It merges two groups in a single one, with potentially overlapping rank identifiers.
- MPI_Group_difference. It creates a group with all the elements of the first group that are not in the second group.

- `MPI_Group_intersection`. It creates a new group from the intersection of two existing groups.

In all set-like operations for groups seen above, the order of the processes of the output group is determined primarily by the order in the first group and then by the order of the second. Figure 2.1 shows the exemplify of the sets operation as described above.

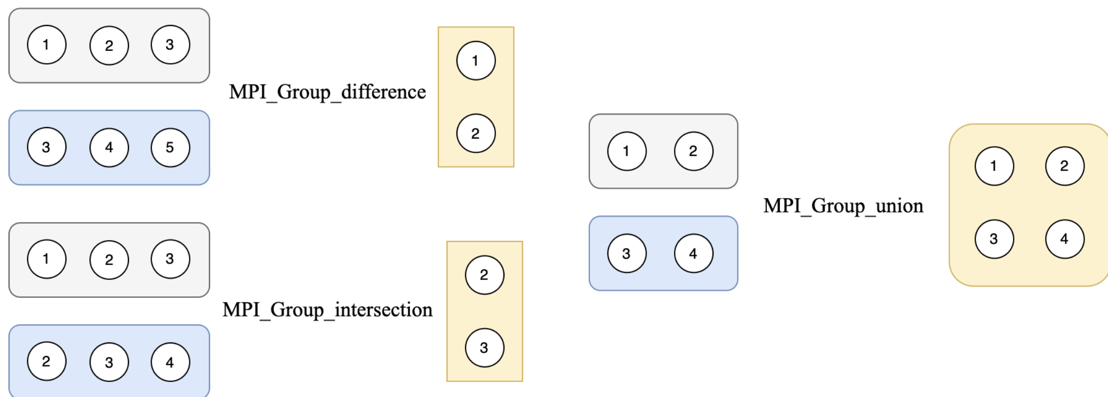

Figure 2.1: The figure shows sets operation on `MPI_Groups`: `MPI_Group_union`, `MPI_Group_difference`, and `MPI_Group_intersection`. The rectangles represent different `MPI_Groups`, while the number inside the circle is the rank according to `MPI_COMM_WORLD`.

When working with different groups, a single process can exist in multiple ones. It might then be helpful to translate the processes' ranks in one group to those in another. For example, starting from a known rank in the `MPI_COMM_WORLD`, it could be necessary to retrieve the related rank according to a specific group that must be used for the current operation. It is sufficient to use the `MPI_Group_translate_ranks` function, which takes as arguments the starting group, the ranks to translate and their number, the destination group, and the pointer in which the translated ranks should be written. If a given rank does not exist in the new group, `MPI_UNDEFINED` is returned.

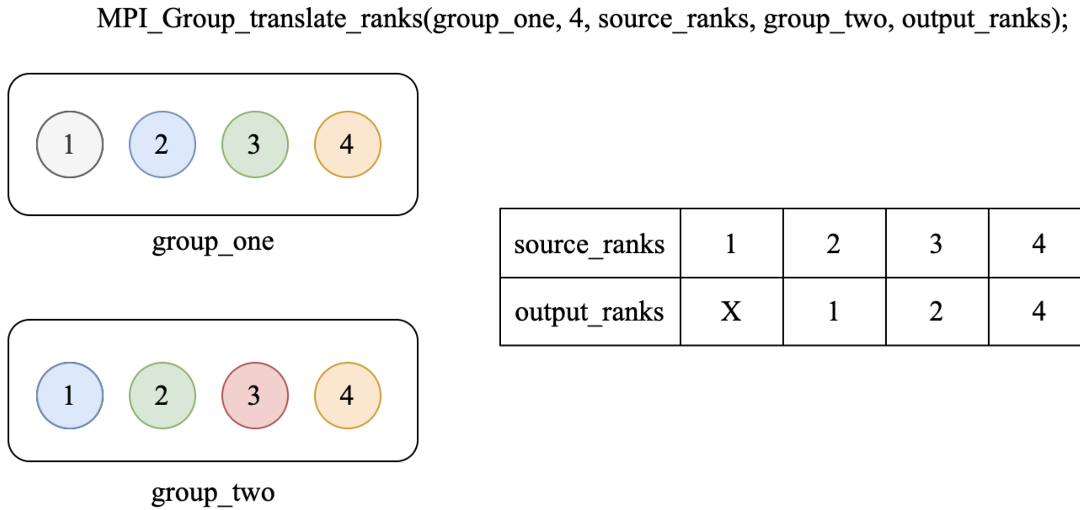

Figure 2.2: The figure shows an example of the `MPI_Group_translate_ranks`. The rectangles represent two `MPI_Group`. The number in the circles is their rank according to the group, while two circles with the same color are the same process. `MPI_UNDEFINED` are indicated with X for brevity.

Figure 2.2 shows the effects of the translation in two groups with four ranks. The figure shows that the white circle (rank 1 in `group_one`) is not present in `group_two`, therefore the operation returns `MPI_UNDEFINED`. Similarly, the blue and green circles have shifted positions.

2.1.4. Communicator management

Communicators encapsulate a group providing the base for collective and point-to-point communication. They can be divided into two types:

- Intra-Communicators. They are the most used communicators and are straightforward in their composition: they contain a group whose processes can exchange messages through the communicator itself. There are predefined intra-communicators that are available for usage without any additional work, such as `MPI_COMM_WORLD` and `MPI_COMM_SELF`, made available after the `MPI_Init` function is called. The user can also create communicators through the functions explored below.
- Inter-Communicators. They empower exchanges between two different groups.

Suppose, for example, an application follows a partitioned approach in a parallel problem. In that case, exchanging information between the different communicators executing the application code may be needed at a certain point. Another example would be the dynamic creation of MPI processes: other communicators are created related to processes designed during application startup and automatic scaling to increase the resources currently working on a problem. To interact between the old communicator and the new one (which uniquely identifies the new processes), it is possible to bind a communicator to two groups which are local and remote, based on the current process.

Similarly to group management functions, communicators can be manipulated through the MPI library. Functions that are used to manipulate communicators are listed below:

- `MPI_Comm_dup`. A simple but often used operation that duplicates an existing communicator.
- `MPI_Comm_create`. It creates a new communicator starting from a given communicator, and a group passed as a parameter. The `MPI_Group` must be a subset of the group internal to the communicator passed as a parameter unless an error is raised.
- `MPI_Comm_split`. It creates a disjoint group of communicators by taking a communicator to split, an integer (the color), and another integer called the key. The output is a set of communicators where the processes in each communicator have the same color and are ordered through the key parameter (ties are broken with rank in starting communicator). Thanks to the key parameter, it is possible to use the split for re-ordering an existing communicator. An example of the function's behavior is in Figure 2.3, where the color of the ranks represents the color parameter, and the number of each process is the related key.

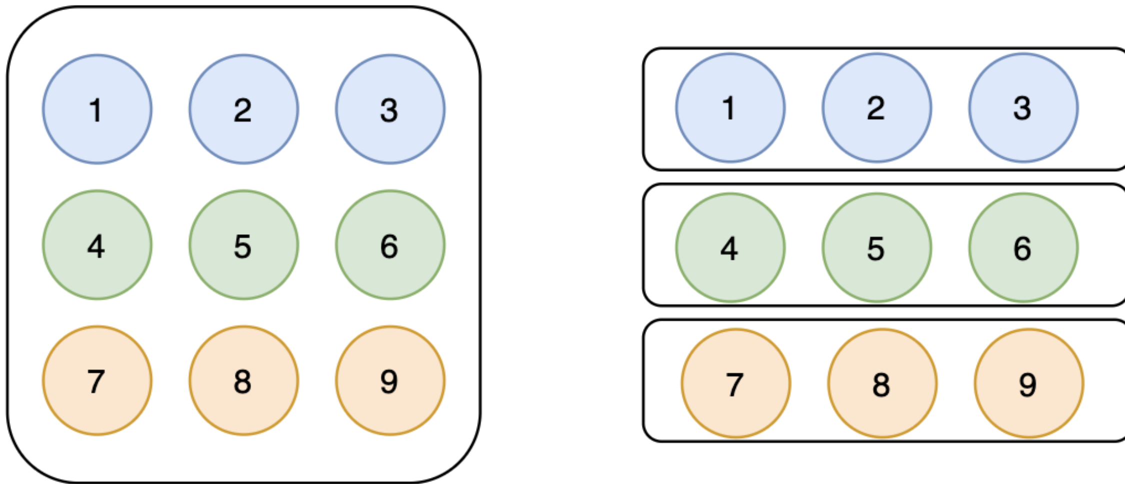

Figure 2.3: The figure shows the behaviour of the `MPI_Comm_split` operation. On the right a communicator containing nine processes. Each one of the processes calls the split operation with the color and key, as shown in the figure. The operation's output is the set of three communicators on the left.

.

A specific set of functions also deals with inter-communicators. They are significant in this work since they allow the management of MPI applications with processes spawned after the initialization of the application (dynamic process management):

- `MPI_Intercomm_create`. It allows the creation of an inter-communicator between two disjoint groups. Note that a similar output inter-communicator is also created in the context of dynamic process management. In particular, the function `MPI_Comm_spawn` dynamically creates a new process, constructing an inter-communicator with the local group containing the pre-existing processes and the remote group containing the spawned ones.
- `MPI_Intercomm_merge` enables the conversion of an inter-communicator to an intra-communicator. This is useful to allow the usage of easier collective and point-to-point communications between all the processes.

An essential difference with the operation on groups is that operations on communicators are non-local and collective; therefore, the participation of every process in the communicator we are modifying is needed. An exception to this is the `MPI_Comm_create_group` operation, which is called only from the processes in the group of the communicator being created.

2.1.5. Dynamic Process Management

MPI allows for the creation and termination of MPI processes after the initialization of the universe. When processes are created outside of the application's initialization, they must be able to connect to the pre-existing universe. To do so, it is necessary to share a common means of communication.
To achieve this and start processes that can interact with existing ones, MPI defines two similar functions: `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`. They execute one or multiple processes, with the given executable, arguments, and related hosts to spawn them.
Similarly to any communicator-modifying operation, spawning a new process is a blocking collective which must be called from every process in the communicator. Moreover, the operation blocks until the spawned child(s) contact the `MPI_Init` operation. The joining of a new process into an existing universe is made possible by three sets of features that are implemented in the MPI Standard, described below:

- `MPI_Publish_name` and `MPI_Lookup_name`. They are primitives needed for MPI processes to publish the port names in a public registry, which must be used to connect the processes.
- `MPI_Comm_accept`. It is a function used by a server application that indicates the status of waiting for the connection from a new process.
- `MPI_Comm_connect`. It establishes communication with an application waiting for a new process to join.

The three features described above enable the dynamic creation of processes and the more generic client-server paradigm through MPI parallel applications.

2.2. Fault Tolerance

When managing exascale computing applications, dependability must be considered: the increase of processors, computational nodes, and in general, points of failures makes it necessary to take the system's health into account.
One of the many sides of dependability is fault tolerance, which serves the purpose of understanding the risks, the mitigation, and the prevention of faults. In an MPI application, the analysis of faults must consider that the behavior after a fault is undefined, as stated in the MPI Standard. It is impossible to continue execution with the guarantees and assumptions outlined in the standard. It is then clear that a single fault of a single component in an MPI application makes it impossible to reach a successful run. Given

the high risk, the rest of the thesis will approach the problem of fault mitigation through different techniques in the realm of dependability, and the section expands on the concepts related to fault tolerance in an MPI environment.
The section is structured as follows: Section 2.2.1 outlines the necessary Fault Tolerance background for the rest of the thesis, Section 2.2.2 explains the MPI user-level error handler, Section 2.2.3 digs deeper into the User Level Fault Mitigation (ULFM) proposal. This section could also include a part for checkpoint/recovery, one of the most common fault recovery techniques, but it will be treated separately in Section 2.3.

2.2.1. Background

To deeply understand the need for fault tolerance, it's first important to analyze the taxonomy of faults. Faults (which could be software errors, such as a mishandled bug, or hardware errors, such as a component failure) can lead to errors - which are inconsistencies in the system's state. Finally, the error becomes a failure if the application cannot handle the inconsistent state. If a failure happens, the component in which the inconsistent state is present may not be repaired from the higher-level components, causing an escalation of the fault to the entire system - blocking the successful completion of the task at hand.
To share a common terminology for the rest of the thesis, it's necessary to define three key terms used in the context of this work:

- Fault tolerance. It is the high-level term that describes any technique which enables an application to deal with faults.
- Fault resiliency. It indicates methods that allow the application to continue execution, even in the presence of faults, without recovering the lost processes.
- Fault recovery. It indicates methods that, in the event of a fault, recover from it by restarting the execution of the lost process.

After understanding the basics of fault tolerance, it is also crucial to outline why it is a key factor in current HPC systems, which are the main users of MPI as a communication protocol. The main goal of such systems is to provide a result, which may be constrained in hours or days, and empowered by thousands of processes running on different nodes. It is therefore straightforward that a hardware fault in a single node could escalate to a system failure for multiple applications running in the HPC system, provoking massive loss in terms of computation in the unfortunate case of missing fault tolerance.

2.2.2. MPI Error Handling

Before diving deeper into error handling in the MPI ecosystem, it is crucial to reference the following quote from the MPI Standard in Section 9.3

> An MPI implementation may be unable or choose not to handle some failures during MPI calls. These can include failures that generate exceptions or traps, such as floating point errors or access violations. The set of failures that MPI handles is implementation-dependent. Each such loss causes an error to be raised. The above text precedes any text on error handling within this document. Specifically, text that states that errors will be handled should be read as they may be handled.

Any MPI-provided object which helps deal with errors is therefore not sufficient by itself to guarantee the correct and successful execution of an application.
In the context of error handling, the standard supports predefined handlers which are tied to MPI objects (communicators, windows, files, and sessions):

- `MPI_ERRORS_ARE_FATAL`. It is the default error handler for any MPI object unless otherwise specified. In case an error is detected, the program (the process receiving the error, and the other connected processes in the MPI application) abort.
- `MPI_ERRORS_RETURN`. The error handler has no effect, and the failure is returned to the user. Note that this does not imply that the application can proceed normally - in fact, the behavior of a standard MPI application is still undefined after an error occurs.

To reach fault tolerance, `MPI_ERRORS_RETURN` is a necessary but insufficient tool.

2.2.3. User-Level Fault Mitigation

Given the significance of the fault tolerance topic in HPC systems, and therefore in most MPI applications, the omission of a standardized Fault Tolerance framework inside the MPI Standard can be startling. Multiple efforts have tried to implement measures to reach fault tolerance and recovery in MPI implementations. An example is the LAM/MPI checkpoint/restart framework [25], which tried to introduce coordinated and transparent checkpointing by using the Berkey Lab Checkpoint/Restart library [15], created in 2005 and whose development has been abandoned since 2009.
One of the most active projects is the User-Level Fault Mitigation proposal (ULFM) [7]. The proposal's main goal is the continuous operation of MPI programs after crashes

without impacting the execution. No MPI call should block indefinitely but raise an error if it cannot succeed.
The approach taken from the proposal does not go in the direction of a specific fault-resiliency framework or approach (such as Checkpoint/Restart). Still, it provides ergonomics APIs to enable users to integrate fault tolerance in their application. The general scope of the goal also succeeds in making the ULFM proposal one of the main goals of the Fault Tolerance Working Group, to include it in the MPI standard. Moreover, the upcoming new version of OpenMPI (one of the most used MPI standard implementations) includes ULFM, empowering numerous projects to adopt Fault Tolerance easily. Also, MPICH, another notable implementation, has proven the implementation of ULFM as possible and with low run-time overheads.
The ULFM proposal defines three supplementary error codes as return codes of MPI operations:

- `MPIX_ERR_PROC_FAILED`. It is raised when a failure prevents the completion of an MPI operation.
- `MPIX_ERR_PROC_FAILED_PENDING`. It is raised when a potential sender matching a non-blocking wildcard source receive has failed.
- `MPIX_ERR_REVOKED`. It is raised when another rank in the application has called the

Moreover, five additional interfaces are introduced to handle faults:

- `MPI_Comm_revoke`. Revokes the communication on the communicator used in the call.
- `MPIX_Comm_revoke(MPI_Comm comm)`. Interrupts any communication pending on the communicator at all ranks.
- `MPIX_Comm_shrink(MPI_Comm comm)`. Creates a new communicator where dead processes in comm were removed.
- `MPIX_Comm_agree(MPI_Comm comm, int *flag)`. Performs a consensus (i.e., fault-tolerant allreduce operation) on a flag (with the operation bitwise or).
- `MPIX_Comm_failures_get_acked(MPI_Comm, MPI_Group*)`. Obtains the group of currently acknowledged failed processes.
- `MPIX_Comm_failure_ack(MPI_Comm comm)`. Acknowledges that the application intends to ignore the effect of known failures on wildcard receive completions and agreement return values.

To detect when a failure occurs (and raise to the user a `MPIX_ERR_PROC_FAILED`), ULFM uses a timeout for each MPI operation. Due to spurious network errors, this approach can cause problems; to solve this, it is possible to adopt two approaches, configurable through Modular Component Architecture (MCA) configuration parameters:

- `mpi_ft_detector_thread` spawns a separate thread that checks the liveliness of the active MPI process.
- `errmgr_detector_priority` which employs the new (present in MPI 5.0) PMIx Reference RunTime Environment (PRRTE) [8], which is a scalable run-time used in HPC network stack clients, supporting the detection of failures natively. Unfortunately, this is only supported in OpenMPI 5.0, an experimental version at the time of writing, and therefore not suitable for production usage.

2.3. Checkpoint/Restart

One of the most widespread techniques to reach fault recovery in computing systems is checkpoint/restart, enabling rollback strategies that empower applications to complete successfully as if no fault ever occurred. The high-level approach consists of periodically writing the state of the application (at the different layers discussed in the next sections) in a parallel file system. After the failure of a component, it is then possible to restart the application starting from the state previously written, avoiding losing the work done beforehand.
To further explore the topic, Section 2.3.1 dives deeper into the taxonomy of different checkpoint/restart techniques, Section 2.3.2 outlines the existing interactions between MPI and checkpoint frameworks, and Section 2.3.3 shows the limitations of the checkpoint/restart approach concerning other fault-tolerance techniques.

2.3.1. Classification

Checkpoint/Restart is a generic methodology to save and restore the state of a program. However, the central concept has been implemented in many different ways according to the specific needs of the users. For example, many applications benefit more from a transparent approach that does not require code changes, while others require the minimization of disk usage. Given these differences, it's helpful to consider a taxonomy of the different approaches:

- Application-Level. It indicates a dedicated library is integrated inside the application code to implement application-aware checkpoint/restart. For example, it can be

implemented as a periodic checkpoint phase of the data necessary for the application to run and an initialization phase to load already check-pointed data optionally. It is the least transparent checkpoint technique since it cannot be used on existing applications without code additions (which may not always be possible, especially in the case of large legacy applications).

- User-Level. It indicates a transparent checkpoint-restart of processes by capturing system calls in user space. Although implementation cost seems not existent for this type of checkpoint, it's important to note that the framework must fully support the application's system calls and needed environment, and the size of the image is larger due to additional user-level information, which is not strictly needed for the application, such as internal run-time states. For example, in case an `MPI_Reduce` is performed, the node which accumulates the results is critical, and in case of a failure, the entire application would be jeopardized without recovery.
- System-level. The operating system actively supports the checkpoint of a process thanks to a kernel module. The method is fully transparent to the application. Still, it requires high privileges, and it is very low-level, tied to the support of a particular Operating System, and therefore has low portability.
- Hypervisor. The virtualization environment used to run the application can save, stop, and restart the application transparently. It presents similar pros and cons as the system-Level checkpoint but requires low administrative rights and generally has higher overhead (due to the restart of an entire virtualized environment).

Given the above classification, it is clear that there is no gold solution to the checkpoint/restart problem, but each alternative has its trade-offs. After analyzing the layer at which the checkpoint happens, we can explore when the different nodes of a distributed system may perform the actual checkpoint.
In particular, we will focus on checkpoint-based protocols with no log-based rollback recovery: the restart is performed solely with the checkpointed image; we can then distinguish three different types of checkpoints:

- Uncoordinated Checkpoint. Each process performs checkpoints with maximum autonomy without synchronizing with the other processes. At the time of restart, the set of checkpoints that produce a consistent state of the execution is computed. The main drawback is the domino effect: when restarting, a large amount of computation can be lost since a consistent state may incur a considerable rollback with respect to the application progress. In the worst case, a consistent state may not be found, making it necessary to restart the entire application from scratch.

- Coordinated Checkpoint. The checkpoints are determined in such a way as to make every single checkpoint consistent. Restarting is greatly simplified, but a higher overhead is paid due to the necessity of a regular checkpoint. To ensure coordination of the checkpoints, there must be synchronization between the different processes through collective operations.

- Communication-Induced Checkpoint. The checkpoint is based on the information received in the messages from other processes - therefore, a consistent state may always exist.

The three different techniques offer different trade-offs concerning the sunk work of performing the checkpoint in case of no failures and the restart cost.

2.3.2. Checkpoint in MPI

After reviewing checkpoint taxonomy, together with the main related to fault resiliency in MPI applications (achieved thanks to ULFM), it is possible to distinguish the two main high-level techniques used to checkpoint MPI applications, focusing on the restart phase:

- Stop-and-restart. The entire application aborts if failure is detected in one or more processes. There is no need to add complexity inside MPI itself since the entire application will restart execution. It is necessary to have a consistent checkpoint between all processes from which they can restart.

- Resiliency. After failure detection, the application can continue to execute without any undefined behaviors. To compensate for the failed processes, the reconstruction phase takes place by restarting the execution of the failed processes. To achieve this, it is possible to roll back all the existing approaches to the latest checkpoints (backward recovery) or restart only the failed processes on their latest checkpoint (forward recovery) without interrupting other processes' execution. The failed processes can start execution either on new nodes (through dynamic process management) or spare ones. Note that the application's behavior mustn't be undefined after a failure to achieve this state.

2.3.3. Limitations

The biggest drawback seen in checkpoint/restart (C/R) solutions is the overhead in terms of time and disk load given by introducing a periodical checkpoint, which must be repeated during the entire application life cycle. In case no failure occurs, the overhead has no tangible benefit. Therefore the need for a checkpoint/restart framework must be evaluated

beforehand concerning the likely hood of failures in the current system. Together with the risk of failures, also the costs and benefits of restarting execution for failed nodes of the application should be estimated. It is often sufficient to reach an approximated result, accepting a margin of errors due to the missing contribution of the failed processes without restarting their work.
Finally, each checkpoint technique at each layer has its trade-offs. In the case of the application-level checkpoint, it may be unfeasible to integrate state-of-the-art frameworks in legacy applications. Moreover, the time cost of adopting the solution can be estimated as superior to the related benefits. The same applies to more "transparent" techniques, such as OS-level checkpoints, which rely on the execution of specific systems and environments. A possible solution to deploying a checkpoint/restart solution is the usage of virtual environments [6]. Thanks to a different climate where memory and state are contained, transparent checkpoint frameworks pose fewer problems. However, a more significant limitation of this approach is the lack of shared memory communication between processes in the same node, which is otherwise used with substantial performance gains. By encapsulating the application, problems such as the checkpoint of memory can be.

2.4. State of The Art

With a background in fault-resiliency and checkpoint, it is possible to review the most up-to-date solutions and the efforts which helped shape the domain. The main distinction taken when analyzing the current work in MPI follows the classification outlined in Section 2.3.2, demarcating stop-and-restart approaches (where the entire application restarts after a failure of one of the components) versus resiliency approaches. Given the focus on transparency of this work, the following sections explore transparent approaches to the problem.
To give a comprehensive outline, the section is divided as follows: Section 2.4.1 analyzes transparent checkpoint/restart techniques, comprehensive of both user-level and system-level approaches. In contrast, Section 2.4.2 explores fault recovery without a transparent checkpoint/restart. Finally, Section 2.4.3 digs deeper inside stop-and-restart techniques for MPI applications, while Section 2.4.4 concludes the section with the analysis of resiliency techniques (with backward and forward recovery).

2.4.1. Transparent Checkpoint/Restart

Checkpoint/Restart is a branch of C/R that focuses on providing fault recovery to the application. In particular, transparent C/R focuses on keeping the application unaware of

the C/R framework. User-level and system-level checkpoint techniques are transparent to user applications since the checkpoint is initiated outside the application's execution. Due to the transparency, consistency issues between different checkpoints must be handled; there needs to be a set of consistent checkpoints which can be used to restart the application.
One of the historic system-level frameworks - using kernel modules to integrate into running applications fully - is Berkeley Lab Checkpoint Restart (BLCR) [15]. To use the software, it is necessary to load a custom kernel module that provides what cannot be otherwise used from user space. Although it is less time and space efficient than application checkpoint due to the missing context of the application, it can optimize for error precursors (such as elevated error rates in memory). The latest development date back to 2013, making it difficult to use with up-to-date applications, and it does not support statically compiled applications.
User-level checkpoint approaches do not require any change to the OS, making them easier to deploy. An example of a framework currently used for fault recovery and process migration is Checkpoint/Restore in Userspace (CRIU) [22]. The library's primary goal is to provide tooling to the checkpoint and restore any running Linux application without super-user privileges and with full transparency concerning network and file protocols used (it fully supports TCP and UDP sockets). The user-space C/R tool is made possible thanks to the new option `CONFIG_CHECKPOINT_RESTORE`, from Linux 3.3: it allows access to parameters of the kernel which needed a custom kernel module otherwise. A novelty introduced to checkpoint/restore is the migration of TCP connections; to perform a live migration of containers, along with the active connections, it was necessary to introduce the possibility of disassembling and recreating TCP sockets without exchanging any packet. Unfortunately, CRIU does not have any active and maintained support for distributed systems.

2.4.2. Other techniques

Other solutions aim to reach fault recovery without using checkpoint/recovery. In case it is possible to use algorithms to re-create the data loss during the fault, it is possible to use Algorithm-Based Fault Tolerance methods [11]. Other than the lack of transparency, such solutions are not explored further due to not being highly generalizable.
Finally, another approach is application checkpointing. The saved state of the application has only the data needed for restoring it. Therefore, it is space and time efficient. The main drawback is that it lacks transparency since it requires direct modification to the existing code. A checkpoint system library that writes checkpoints to disk, RAM, or parallel file systems, is the Scalable Checkpoint/Restart (SCR) library [19]. Its main

novelty is exposing the usage of multi-level checkpointing, which accounts for different levels of resiliency in a single run. For example, it employs frequent lightweight (on RAM) checkpoints for common failure modes and uses more expensive checkpoints only for less common but severe failures.
A similar approach is taken from the Fault Tolerance interface (FTI) [5], which allows computational scientists to granularly select the data needed to be checkpointed. As a result, it minimizes the utilization of space and energy. The main goal of the framework is to reduce the I/O bandwidth used by HPC systems since it is a scarce resource concerning computational capabilities. Adopting complex storage hierarchies and redundancy schemes using error codes empowers the users to choose the best checkpoint strategy for their problem.
The main drawback of the past application-level approaches is the coupling with the application code, which removes transparency. However, such techniques might be helpful in HPC systems if integrated with existing recovery methods to restart or continue the application. Such examples will be explored in Section 2.4.4.

2.4.3. MPI stop-and-restart

Analyzing the checkpoint/restart techniques in MPI, considering that resiliency is not a part of the standard, it's clear that most rely on a global restart technique, where all processes are restored after failure. Although the frameworks explored below incur overhead due to the lack of resiliency, they are helpful to better understand strategies to checkpoint an MPI application transparently.
The Distributed MultiThreaded Checkpointing (DMTCP) [4] is a great utility that achieves a similar goal as CRIU, transparently checkpointing a process without any modification to the code or system. Moreover, it provides native support for job schedulers (such as SLURM), networking communications commonly used in HPC systems (such as InfiniBand), and initial support to C/R for distributed applications. The main novelty concerning CRIU is the support for distributed applications, such as MPI. To further support MPI checkpoint/restart in HPC systems, a special plug-in of DMTCP was created. MPI-Agnostic, Network-Agnostic MPI (MANA) [13, 31] innovates by dividing the memory of an MPI application into two parts: an upper part, which is restricted to application-level objects, and a lower part, which is used only for MPI low-level details. The checkpoint is focused on the application-level part, while the MPI memory is rebuilt by replaying the previous MPI calls, reaching a consistent state before the failure. Although development is active, the plug-in is currently working mainly on Cori, a system that is part of the NERSC supercomputer.

Another significantly different approach that was proposed is Reinit [9]. The model assumes that in case of a failure, MPI processes reinitialize themselves going through `MPI_Init`. Furthermore, the approach takes a global synchronous restart of the application. To achieve this, all the processes must be notified of the fault promptly to avoid deadlocks. The main flaw is the limitation to the recovery procedure adopted: any failure leads to a global synchronous restart, with the risk of unnecessary overhead for non-critical tasks. Finally, an effort, which is now abandoned but provides insight into how checkpoint/restart frameworks can be integrated, is the usage of CRIU with MPI [23]. It is transparent to the user of the application. It is integrated directly inside the C/R infrastructure of the OpenMPI project and its ORTE run-time, providing hooks to the checkpoint with CRIU for the processes running at each node. Unfortunately, the C/R infrastructure on which the plug-in was based is now deprecated, and ORTE run-time is no more actively developed.
The approaches require the application to stop in case of any type of failure, therefore they share the following drawbacks:

- In case of non-critical failures, the application is stopped and work is lost. This causes overhead in latency and resource usage in case the failure was not critical and would have not impacted the result.
- Since the application performs a global stop-and-restart, a checkpoint must be done on all the nodes of the application, producing overhead in terms of disk usage and unnecessary latency.
- The recovery is backward to the latest consistent checkpoint, therefore the work done from the other non-failed processes is lost.

2.4.4. MPI resiliency

In this final section, efforts that aim at the reconstruction and correct completion of the application after a failure, without any global restart, are explored. To understand the high-level differences in resiliency solutions, it's useful to define the following two terms:

- Global backward. All the processes recover backward to the latest consistent checkpoint, both alive and failed ones that are restarted, losing the progress done in the meantime.
- Local backward. The failed processes recover backward to the latest consistent checkpoint, while the alive processes can continue to perform work and not necessarily participate in every part of the reparation.

Local backward is not faster than global backward for all processes since restarting processes may have to participate in the reparation procedure. Moreover, local backward could require more attention while saving the state of checkpoints, considering that some processes will not repeat synchronizing calls. On the other hand, the main advantage is less coordination during restart phases since processes not involved with the failed ones can continue processing without interruptions. This is particularly significant for embarrassingly parallel distributed applications, where synchronization is rare.

The next chapter focuses mainly on the second approach - due to its numerous performance advantages, such as skipping the re-initialization of the entire process. Analyzing the resilient solution, two main approaches make it possible to reconstruct a consistent state:

- Shrinking solution - in case of failures in processes that are not crucial to the completion of the application, it is possible to continue without restarting them. However, in this case, we are not restoring the failed process.
- Non-shrinking solution - for critical processes where a restore is necessary, due to the importance of the process in the flow of the application, it is possible to reconstruct the failed process through the dynamic process management of MPI applications, or by using spare processes started during initialization.

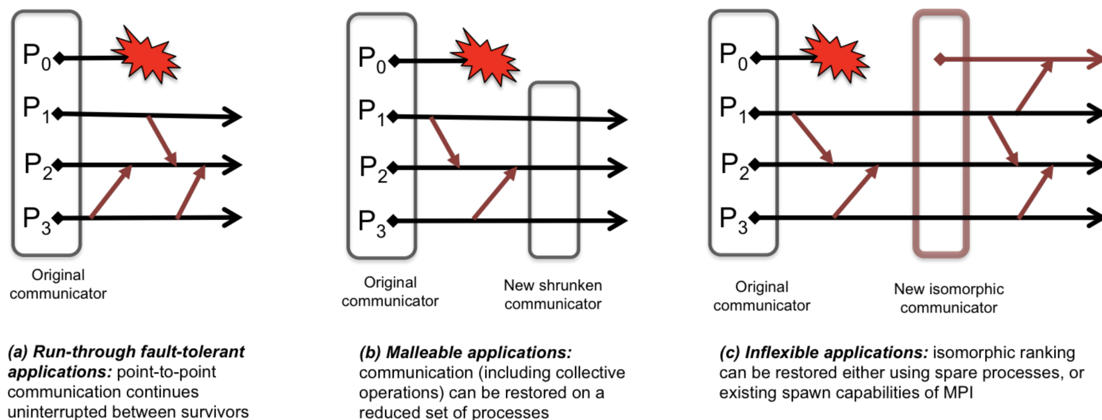

Figure 2.4: The figure shows the difference between shrinking and non-shrinking operations on communicators after a failure. Taken from [18]

.

The figure 2.4 shows how fault-tolerant applications must consider changes in the size of communicators (malleable applications). Inflexible applications, on the other hand, can

use a non-shrinking approach to keep the same communicator.
We start by analyzing the non-shrinking approaches in the current literature. The first framework analyzed is Reinit++ [14], which builds on Reinit [9]. To use the library, the application is modeled with a hierarchical topology - when the root node detects a process failure, it proceeds with recovery by rollbacking all survivor processes. The survivor processes continue execution from the `MPI_Reinit` function call, which defines a rollback point. The application is then responsible for autonomously recovering the state. The approach is a backward restart, with the risk of losing work already done by survivors. Moreover, only the global communicator is repaired after the rollback, which can lead to additional complexity in repairing the existing communicators.
A series of other libraries build upon ULFM to implement fault recovery. For example, Checkpoint-Restart and Automatic Fault Tolerance (CRAFT) [26] provide a generic library for application-level checkpointing and dynamic process recovery in case of failures. It introduces the support for non-shrinking and shrinking recovery and enables the checkpoint of application-variable through an easy-to-use interface. Different from the goal of this work, it does not repair MPI objects such as communicators, leaving the responsibility to the users; this can limit the usability of the framework with existing applications. Moreover, it cannot differentiate shrinking or non-shrinking recovery based on the process. The communication recovery method must be used at the communicator level, limiting the possibility of respawning only the critical processes.
Similarly, Fenix [12] uses ULFM to detect failures, and, differently from CRAFT, it also recovers the communicators. It does not allow skipping the respawn of the new processes since it only supports the non-shrinking model. The approach lacks support for the failure of critical nodes that are needed for the completion of the operation (for example, the root of an `MPI_Reduce` operation).
An innovation to save disk space and achieve diskless checkpointing is introduced from Local Failure Local Recovery (LFLR) [30]. Instead of using MPI's dynamic process management utilities to implement non-shrinking global backward recovery, it pre-emptively creates spare processes during startup. Although it saves time during restart, it incurs higher latency overhead during startup, and resource usage in a shared cluster.
The ComPiler for Portable Checkpointing (CPPC) [17] is an example of a library that can be used on top of ULFM. It implements portable data recovery by not tying the application-level checkpoint technique to the MPI communicators. The main advantage with respect to other frameworks is that the integration of an application with CPPC can be re-used with multiple C/R frameworks.
About shrinking solutions, it's crucial to distinguish approaches that require modifications to the existing code or transparent ones. The first ones are built on top of ULFM, allowing

us to solve problems where the application is malleable thanks to the specific support of the algorithm. Examples of applications that can be adapted dynamically without loss of accuracy are Partial Differential Equation solvers, which are made fault resilient through ULFM in [28]. To conclude with transparent solutions, novelty is introduced by Legio [24]. It builds on the ergonomic ULFM APIs to provide resiliency to MPI applications by repairing communicators and skipping any rollback to previous checkpoints. No restart capabilities are included since the recovery phase does not include respawning the lost processes. This property implies that the failure of a critical node could jeopardize the entire application without the possibility of recovery.

3 | Transparent fault recovery

This chapter will focus on the various attempts to reach transparent recovery with checkpoint/restart. In complex applications, resiliency (allowing continuation after a fault) is not enough to reach a meaningful enough result. The optional possibility to introduce recovery in the most critical processes while maintaining resiliency in the rest would empower embarrassingly parallel applications to reach the desired results even in the presence of faults. Moreover, a novelty introduced from distinguishing critical processes is to execute checkpoints only where needed, with important savings in disk usage. The journey starts with integrating the fault resiliency framework Legio and different fault recovery tools to reach the end goal.
Section 3.1 introduces the initial critical requirements of the project, outlining the needed functionalities and nice-to-haves, and Section 3.2 outlines the basics of the Legio framework, on which the subsequent efforts are based. Then, Section 3.3 explores user-space checkpointing and the integration between the DMTCP and MANA framework, while Section 3.4 focuses on the attempt to integrate CRIU with MPI in different OpenMPI versions. Finally, Section 3.5 explains the rationale behind abandoning a transparent approach for C/R and lays the foundation for the application checkpointing work of Chapter 4.

3.1. Requirements

At the high level, it is possible to summarize the key requirements which make up transparent fault recovery as follows:

- Fault resiliency must be supported. In the presence of faulty process(es), the application should be able to continue without any problems. This ensures that the application can continue without non-critical processes and that it can start the recovery process in case of critical failures.
- Fault recovery must be supported, and activating it on specific ranks should be possible. The overhead created from a restart procedure may not be worth it for

non-critic processes, for which an acceptable error in the final result may be accepted with the goal of faster execution. On the other hand, failure on critical ranks jeopardizes the execution of the application, and recovery is necessary.

- In the presence of fault(s), the semantics and guarantees of MPI commands must not change. The behavior of MPI operations must be defined and transparent to the user after a failure.
- No change should be needed in the application code. Note that changes in the run-time environment, run-time commands, and general libraries installed in the run-time OS are acceptable. The necessity of transparency may indeed pose constraints on the environment used. On the other hand, given that the recovery procedure should be transparent, the application developer should not autonomously save the application's state before a fault. Moreover, the user can still decide the timeline in which the checkpoint is performed (by modifying the code) or tuning a periodic job to checkpoint the application.
- The overhead of fault resiliency and recovery should be minimum. This is particularly important in the absence of failures: many jobs could have a life period for which failures are unlikely. Therefore, introducing heavy overheads on time and disk usage in the absence of failures may be a net loss concerning restarting from zero in case of faults. In practice, keeping the overhead to a minimum also in the presence of a fault drives the thesis towards reaching a local restart framework, which avoids the global restart of the entire application and removes the need for synchronization between the active processes.

3.2. Legio - a fault resiliency framework

Legio is a framework designed to offer resiliency in MPI applications[24]. Its main focus is the absence of any intrusiveness to the target application by wrapping MPI calls to manage faults and handling failed processes gracefully. The overall performances are radically unaffected, and the library is based on the ULFM framework, which is included in OpenMPI version 5.0.0. These properties make Legio a perfect candidate for the backbone of the work of the thesis, providing transparent fault resiliency.
Section 3.2.1 outlines the high-level approach employed by Legio to support the most common MPI operations and the necessary glossary from the underlying ULFM library to grasp the subsequent sections fully. In contrast, Section 3.2.2 goes deeper into the Legio architecture to understand the underlying algorithm enabling the framework, with particular attention to the presence of multiple communicators.

3.2.1. Overview

Before digging deeper into Legio's architecture, it is necessary to define a few key terms and concepts used in the rest of the thesis and the Legio domain:

- `MPI_ERR_PROC_FAILED` is the return code of MPI calls after a failure has been noticed. Note that this is present in the ULFM proposal and not the official MPI standard.
- A faulty communicator is a communicator where one of the processes failed, but it has not been noticed.
- A failed communicator is a communicator where (at least) one process noticed the failure.
- Local operations work in faulty and failed communicators, such as `MPI_Comm_rank` and `MPI_Comm_size`. In addition, point-to-point communications work in faulty communicators if the processes are not failed, and collective communicators will not work in failed communicators.

Note that the rest of the thesis focuses on local, collective, and point-to-point operations mentioned above since they are the most used in MPI applications. Legio supports other types of operations (such as File operations) but is not explored further in the context of fault resiliency and recovery.

Given these assumptions, it is now possible to dig deeper inside Legio; the framework reaches failure resiliency by ensuring that every supported MPI operation is wrapped in custom structures by using the profiling interface `PMPI` to intercept any call. By avoiding the direct use of the application communicators, Legio can prevent failure and faulty communicators from being used directly by the user. We can then distinguish two types of operations: the first are the ones where a process can notice a failure. Therefore the communicator can become a failed one, and they are the collective operations; the second ones are point-to-point operations, but they do not involve all the processes in a communicator. Therefore, it does not introduce any particular algorithm to recover from faults other than retries, which aim to limit transient faults.
An outline of how the framework currently reacts to a fault in a collective operation, exemplified as a `MPI_Barrier` on `MPI_COMM_WORLD` and the high-level operations taken in the wrapped call from Legio:

1. The `MPI_Barrier` is executed on the application code, and thanks to the `PMPI` interface, the call is forwarded to Legio.

2. The first operation is retrieving the MPI structure, which maps the MPI_COMM_WORLD component to a Legio internal structure. This operation ensures that the communicator used from the application (alias, in the rest of the thesis) is not used in actual MPI operations but only its duplicate.

3. The MPI_Barrier is performed on a duplicate communicator of the alias, which is created during the creation of the communicator (which is during MPI_Init for MPI_COMM_WORLD.

4. In case the operation succeeds, Legio returns the return code to the application, and no further step is required. However, if a failure occurs and the return code is equal to MPI_ERR_PROC_FAILED, Legio must replace the communicator of the failed process. To do so, it uses the MPIX_Comm_shrink operation to create a communicator without any error, the communicator saved in the internal MPI structure is substituted with the shrunk one, and the operation is repeated until success.

The failure of a process in a communicator can therefore cause the shrinking of the communicator itself; this assumption requires additional steps to keep the loss transparent to the rest of the application. When MPI operations from the application code need to specify the exact rank number of a process (for example, the MPI_Bcast), they will use a rank number that is not consistent with the ranks of the processes after the communicator has shrunk. To maintain compatibility, Legio uses the MPI_Group_translate_ranks function, which must be applied to all application-provided rank numbers to ensure the correct rank is referenced in the substitute communicator.

3.2.2. Multiple communicators support

The example described in the previous section focuses on the MPI_COMM_WORLD, which is a single communicator. In MPI applications, multiple communicators are often created and used to separate chunks of work. Any alias communicator created in the application must then be mapped to an internal object in Legio, which contains the authentic communicator used for MPI communication: Legio implements this wrapper as a ComplexComm. The translation from an alias communicator to a ComplexComm is possible thanks to a key-value map. To link the two communicators, Legio extracts an integer from the alias communicator with the MPI_Comm_c2f function (present in MPI for compatibility with Fortran). The MPI operation ensures that each returned integer is unique and associated with each MPI object - even two duplicated communicators will produce a different result. By exposing this key-value map, all the communicators passed as parameters to MPI calls can be translated into the related ComplexComm. The support is integrated into each MPI

operation which causes the creation of a new communicator: the `alias` communicator, coming from the user code, is transformed into the communicator internal to Legio (the `ComplexComm`), and the MPI routine is applied to the real communicator.
The problem is fascinating in the context of fault recovery since it shows how much MPI applications can rely on different communicators, which can be wrapped but are saved as MPI structures in MPI memory regions. Moreover, a pointer to a communicator could also change its value during the application's lifetime, making it even more challenging to keep track of it from the underlying framework. This pitfall will be discussed more in the following sections, where user-space checkpointing will overcome it.

3.3. Distributed MultiThreaded Checkpointing

As suggested in the future work of the Legio paper [24], one of the main frameworks for checkpoint/restart supporting distributed systems is Distributed MultiThreaded Checkpointing (DMTCP)[4]. In this section, we explore DMTCP with its main plugin, MANA [13, 31].
One of the main characteristics of DMTCP, differentiating it from other frameworks, such as CRIU, is the support of distributed applications like MPI. Rather than supporting the specific distributed computing framework, DMTCP checkpoints the main application thread and intercepts forking system calls, managing to affect also the forked threads. Although it satisfies most requirements and appears to be fully fledged, the main drawbacks concerning the goal of the thesis are its higher overhead during the restart phase. It performs stop-and-restart and misses fault resiliency without recovery for non-critical processes.
This section outlines the main characteristics of the DMTCP architecture in Section 3.3.1 and focuses on the attempts to modify the framework to make it work with Legio and local restart in Section 3.3.2. Finally, Section 3.3.3 digs deeper into the MANA plugin, which better supports DMTCP specifically for MPI applications.

3.3.1. DMTCP Overview

Four main features characterize DMTCP:

- Multithreaded - each checkpointed program can have dynamic threads, which will be recursively checkpointed without any additional configuration.
- Distributed - it allows the checkpoint of a network of programs connected by sockets (TCP in particular).

- Transparent - no modification is needed in the binary source code.
- User-level - the kernel is not modified, unlikely other checkpoint frameworks such as BLCR. Therefore, DMTCP can be directly bundled with the application.

At the backbone of the multithreaded support is MTCP, the MultiThreaded Checkpointing component; the DMTCP framework builds on top of it to reach distributed processes support.

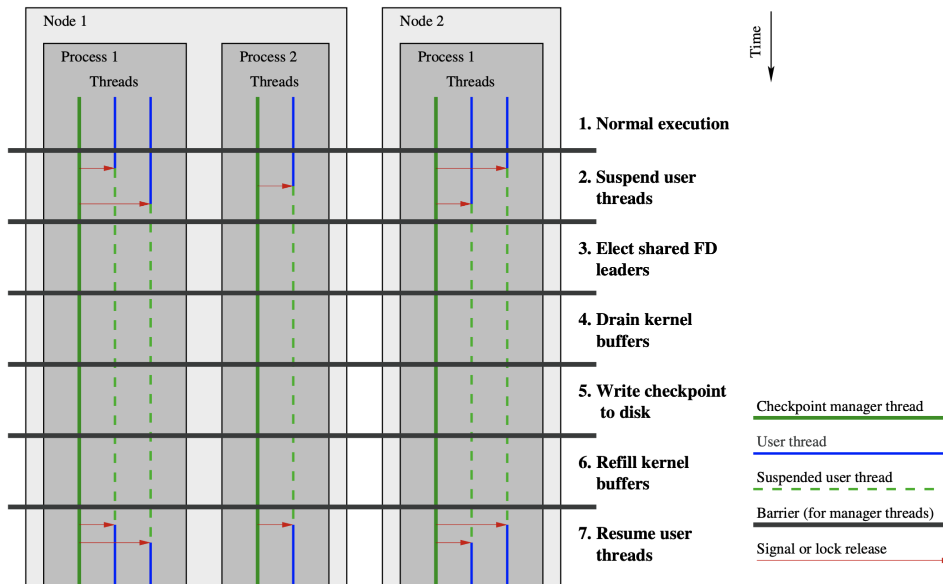

Figure 3.1: The figure shows the checkpoint operation in DTMCP in two nodes with 2 and 1 threads respectively. Taken from [4]

During the startup of a new process run with the DMTCP wrapper, the framework adds a library that accomplishes two main goals: a checkpoint coordinator thread is spawned, and a TCP/IP connection is opened between the application and the coordinator. Moreover, DMTCP wraps the necessary calls to the `libc` library, to ensure that system calls which must be known to DMTCP are intercepted.
The manager waits until the coordinator requests a checkpoint (which can happen either programmatically or periodically). Once the checkpoint is requested, the user threads (where the application binary is running) are suspended, sockets information is saved, kernel buffers are drained (by receiving until there is no more available data), the checkpoint is written in the disk, and the drained socket buffer is sent back to the sender. Finally, MTCP resumes the user threads, so the application can continue (optionally, DMTCP can skip this step if the checkpoint is a destructive operation, such as in the cases of process

migrations). Figure 3.1 shows the checkpoint procedure in the case of two nodes, with 2 and 1 threads respectively. Each thread is attached to a checkpoint manager which suspends the execution while the data is written to disk, and resumes the user threads afterward.

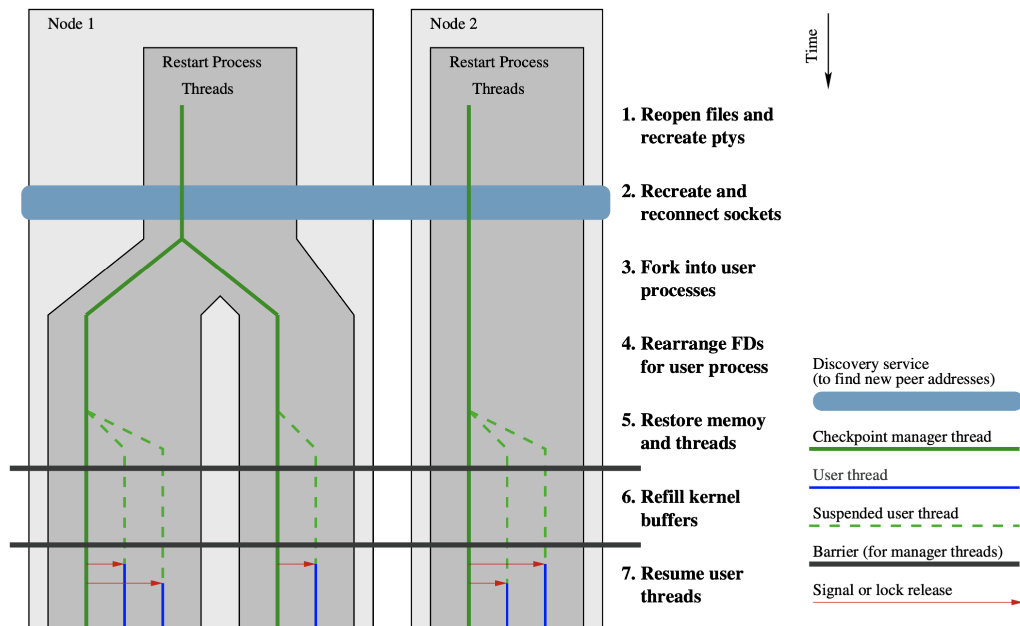

Figure 3.2: The figure shows the restart operation in DTMCP in two nodes with 2 and 1 threads respectively. Taken from [4]

During restart, each host of the distributed application participates in the process; the files are re-opened and sockets reconnected, and the user processes are restarted by checking the number of open user processes running at the checkpointing time. Finally, the MTCP routine is run and restores the local process memory, and the threads are free to be restarted. Figure 3.2 shows the restart procedure after figure 3.1 the checkpoint procedure is completed. Each node spawns a single thread to recreate and connect the sockets, before optionally forking and resuming the original user-threads.

3.3.2. DMTCP with local restart

The high-level description of the checkpoint-restart phases of DMTCP shows the global nature of the framework's architecture. Given this assumption, the goal of the thesis is twofold: experiment with OpenMPI to determine whether the DMTCP framework is

working as described with the latest stable version, and modify the necessary parts to skip a full restart and support the restart of a single process, with a special look into how resiliency for non-critical processes can be implemented.

The first step in the research was ensuring whether DMTCP works with MPI without additional modification. A simple test showed already that the framework lacks full compatibility with generic MPI applications (the last update of the framework dates back to 2019) since various errors prevented the correct checkpoint of the application. In particular, MPI opens disk devices (for shared memory communication purposes) that DMTCP does not directly support. The library presents a list of supported hardcoded devices' names, which does not include the device in which shared memory communication was performed. The support was easy to introduce, adding the device's name to the list and reusing the same utilities for draining the buffers before a checkpoint.

After this fix, it was possible to achieve restart of an MPI application. Moreover, introducing the ULFM layer and Legio library didn't create problems during compilation time. Although not optimal for the requirements of this work, it would also easily be possible to perform a stop-and-restart only when critical failures occur by using DMTCP's API inside Legio as follows:

- Checkpoint is taken only for critical ranks since they are the ones that must be restarted in case of failure.
- In case of a failure of a critical rank, DMTCP is expected to perform a stop-and-restart of the processes still alive from the latest checkpoint.
- In case of a failure of a non-critical rank - no global restart is required; the application is expected to continue thanks to fault resiliency.

The proposal does not satisfy the local restart requirement since a stop-and-restart procedure would be needed.

The leading blocker in achieving this behavior is the architecture of the manager of DMTCP and the concept of a coordinator, which has three primary states: `RUNNING`, `CHECKPOINTED`, or `RESTARTING`. An assumption for the checkpointing of multiple threads is that such states are shared between the entire group of processes. The constraint is particularly tied to the sockets and their ties to the state of these processes. The following steps happen during the flow of the application to the sockets:

1. During the application's startup, socket connections are opened between the different ranks of the MPI application. DMTCP intercepts the system calls automatically (wrapping TCP, Unix, or terminal calls), saving the parameters used.

2. During the restart operation, the sockets will be re-opened before restarting the main application. This is done to ensure that if sockets are shared between the various application processes, they can be shared since they are opened only once.

In the case of a restart of a single node (with state `RESTARTING`), the DMTCP coordinator will try to open the socket connections which were already opened are currently used from the already `RUNNING` processes. Therefore, the socket information saved during the startup cannot be used to reconnect to the running application. Moreover, it cannot reconnect to the existing sockets as the old (failed) process is being restarted. OpenMPI sets the process as failed concerning the socket it was connected to, making it impossible to rejoin the communication: the attempt to reconnect goes into deadlock. A possible solution to this problem would be to use re-connection methods internal to MPI, such as `MPI_Comm_connect/accept`. This strategy is not pursued in the context of DMTCP, since it goes beyond the nature of the framework itself (which should autonomously handle distributed processes). Instead, it will be explored in Section 3.4, since CRIU is more flexible in checkpointing a single process.
A final attempt was also tried by using the primitive MTCP (which does not support distributed processes) framework, the backbone of DMTCP; unfortunately, it cannot efficiently work since the DMTCP coordinator is also responsible for infecting the forked processes when forked, and with MTCP it is not possible to checkpoint processes performing forks.

3.3.3. MANA Plugin

Given the failed attempt to implement DMTCP with a local restart, the MPI-Agnostic, Network-Agnostic MPI (MANA) [13] plugin for the DMTCP framework was explored. Built on top of DMTCP, MANA is a plugin that hooks in the main phases of DMTCP and was extremely inviting for a few reasons: it seems to be actively developed since the efforts for transparent checkpointing for MPI tied to DMTCP are moved on the MANA plugin. It focuses on reducing overhead by separating the memory zones for checkpoints. In particular, since it divides memory related to the application and the MPI low-level structures, it can perform a fast checkpoint, restart user-level processes, and recreate only the necessary MPI structures by repeating the MPI calls done previously to the checkpoint. By requiring a statically linked version of MPI, MANA can differentiate memory zones that will contain MPI-specific structures and ensure that the rest of the application's memory will go to other predefined zones that can be easily checkpointed.
Unfortunately, the project is far from general enough to provide a generic checkpointing system. Instead, MANA and its research group focused on the Cori supercomputer

at National Energy Research Scientific Computing Center (NERSC), which is used for checkpointing and restarting processes. Still, the support for different architectures is low. Moreover, there is no compatibility with OpenMPI since MANA builds up on different assumptions starting from how the MPI communicators are saved in MPICH.

3.4. CRIU

Checkpoint/Restore in Userspace (CRIU) is a tool for the Linux operating system which can freeze running applications, checkpoint it to persistent storage, and use the saved files to restore it and restart the application from the point it was frozen. Given the relevancy of CRIU in the C/R space, it was a natural target for tools to explore to reach transparent C/R in an MPI context. Many domains use CRIU as the tool of choice for C/R and live migration of processes, ranging from IoT Edge Functions to Kubernetes pods.
This section explores CRIU's internals and attempts to integrate them inside different MPI versions. Section 3.4.1 outlines the leading architecture and internals of CRIU, while Section 3.4.2 explores the previous attempt at integrating CRIU and MPI. Finally, Section 3.4.3 investigates a possible flow for integrating CRIU with Legio and outlines the problems encountered with the various MPI versions.

3.4.1. Internals

CRIU provides its services through three main interfaces: the Command Line Interface, Remote Procedure Calls (RPC), and application libraries in different languages (such as C and GO) which wrap the RPC interface. When checkpointing a process, it must be indicated the PID to dump to disk; CRIU can then save in a provided directory information about the system, memory dump files (which contain memory content, open file descriptors, and sockets), and auxiliary files.

The main steps are the following, starting from the dumping phase:

1. Code is injected into the target PID through `ptrace` - a system call that provides a means to observe the state and manage the execution of another process. CRIU can then list the relevant sub-processes reading the proc$pidtask/$tid/children files. While collecting the processes, the tracer requests to stop each one.

2. After all threads attached to the target PID are frozen, CRIU starts dumping the resources. Virtual Memory Areas, memory-mapped files, and file descriptors are all dumped, together with the registers and the other core parameters.

3. In case CRIU is used as a C/R tool and the process should be kept running after the checkpoint, CRIU cleans up the injected code and resumes the processes where they've been stopped.

After the dumping phase is finished, CRIU can be used to restore the process as follows:

1. The restorer process reads all the dumped images, to understand which resources must be restored, and restores the processes by calling the fork system call. Note that to assign the same PID as the one that was assigned, it sets the `/proc/sys/kernel/ns_last_pid` file to the desired PID minus one. Alternatively, to avoid slowdowns and race conditions, the `clone3` system call can be used to set the PID of a cloned process directly.
2. After creating all the files, CRIU restores all the resources dumped previously, from memory areas to file descriptors, from the forked processes in the precedent step.
3. Finally, CRIU switches context to the restorer process, cleans up the memory, and lets the forked processes run from where they were initially checkpointed.

One of the significant similarities with DMTCP is that CRIU is a user-space C/R tool; this allows to keep the kernel clean, avoiding problems of compatibility and maintainability. However, it currently requires root permissions, which can be hard to obtain on shared supercomputer resources. Work is ongoing to support unprivileged runs of CRIU, and modified versions of CRIU already support it [1], by losing some capabilities such as restoring sockets upon restoration.

3.4.2. MPI Process Migration with CRIU

Given the trend in High-Performance Computing, a major need is to augment the management approach by enabling process migration, moving processes that are running to the location which is best for their resource need (which can change over time). To reach the high-level goal, CRIU integrated with OpenMPI, by re-introducing the old checkpoint/restart module [25] in the MPI codebase.

In the work from Adrian Reber, a new module was added that allowed to call the `orte-checkpoint` command; by communicating directly with `mpirun`, it automatically checkpoints also the child processes. During restart (which happens in case of a process migration, when the initial checkpointed program has terminated in another node), the `orte-restart` utility spawns a new `mpirun` process, which has the main goal of restoring the original processes. The main problem encountered during the implementation is the collision with the MPI expectation of having the restored processes as a child of `mpirun`.

The spawned processes were instead children of `criu`. The problem was solved by running an `exec` call (with root privileges). With this new flow, the library `libcriu.so` is not used and the processes created are children of `mpirun`.
Unfortunately, the work done in the paper was based on an older MPI version, and currently, the checkpoint restart/framework powering the integration has been completely removed from MPI.

3.4.3. Legio with CRIU

Given the tool's robustness and the previous successful attempt at integrating CRIU with OpenMPI, we investigated the integration between Legio (and the underlying ULFM framework) and CRIU. The problem to face is very different in the case of the local checkpoint/restart of a single MPI process since it requires the entire application to be transparent over the migration of a single process, concerning MPI low-level details (such as communicators and ranks).
The high-level execution flow relies on the dynamic process management capabilities of MPI and is described below:

1. The first step is the checkpoint of the current process, which is done synchronously before each MPI call. This always ensures a consistent state of checkpoints between the different processes and avoids deadlocks.

2. In case the application detects a failure of a non-critical process, CRIU should not be involved in the failure resiliency operations; Legio will autonomously repair the existing communicators, and the operations will restart without any problems.

3. In case the application detects a critical process failure, Legio will handle the failure by restarting the failed process(es) and re-joining it with the already running MPI application by repairing the communicators removing the failed process, and including the new one.

Dumping the current process to permanent storage is explored first; in the context of maintaining consistent checkpoints, it's necessary to checkpoint the application before every MPI operation; this ensures that in case of a failure, the latest checkpoint always has the necessary context to restart, without creating any deadlock. To achieve this goal, the check-pointing flow must be aware of the application execution and checkpoint before any MPI call. To implement this behavior, CRIU was integrated inside Legio, using libcriu, a C API for CRIU, which wraps the RPC, providing C function calls for each RPC command to interact with the CRIU daemon. The library integration into the project was seamless, and the checkpoint phase succeeded without problems.

Legio already partially supports the second step. It was sufficient to implement a list of non-critical processes which would not have gone through any restart procedure; instead, they repaired the communicator removing the failed ranks.
The third step, which is the restart, was the one that caused more problems during the implementation. First, the restart flow should be implemented using MPI dynamic process management operations. A caveat is that the most commonly used one, `MPI_Comm_spawn`, expects the created process to call `MPI_Comm_Init`; this cannot happen in case of a restart with CRIU since the restarting process is already initialized. MPI also exposes a set of routines to mimic a client-server execution flow, `MPI_Comm_connect` and `MPI_Comm_accept`; the two operations could then be used to link the restarted process with the already running one, and repair afterward the remaining processes by re-constructing their communicators taking into account the new process. Below is a code snippet that shows the functionality achievable with the two routines:

```
// server.c
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;
/* ... */
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);
=
MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
// intercomm enables connection with the client

// client.c
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
printf("enter port name: ");
gets(name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
// intercomm enables connection with the server
```

To complete the restart (let CRIU pass control of execution to the restarted MPI program), it was necessary to disable the shared memory communication layer of MPI through the Modular Component Architecture (MCA) configuration options. CRIU could not checkpoint and restart MPI programs that were using shared memory; instead, setting TCP as Byte-Transfer Layer (BTL) allowed CRIU to close and open the sockets of the restarted process correctly. Note that this change comes with a great performance drawback in the communication between processes in the same node, where shared memory communication shines. Although this could represent a violation of one of our requirements in our work, we assume that our application is embarrassingly parallel. Therefore the communication

and synchronization between different processes should be minimal.
Unfortunately, the integration was still problematic due to the poor support of the dynamic process management routines inside the different MPI versions:

- The main version of Legio is built with ULFM dating back to November 2018, with the related OpenMPI version 4.0.2. Unfortunately, the OpenMPI version had a critical bug that made it impossible to use `MPI_Comm_connect` and `MPI_Comm_accept` calls; the two dynamic process management functions worked with the assumption of a shared broker, called `orte-server`, which allowed the communication and discovery between different MPI applications. Unfortunately, in the specific version of OpenMPI, the broker proved to be non-functional. Although later minor versions fixed the problem, ULFM support was tricky to re-introduce since the next supported version of ULFM was the major 5.0.0.
- Given that OpenMPI version 5.0.0 is in beta and contains ULFM, the integration was attempted by updating the Legio framework to the new version. The first steps of the restart flow (the sharing between restarter and restarted process of a common port to connect their sockets) worked. However, after the `MPI_Comm_connect` and `MPI_Comm_accept` functions were called, an internal error was raised, which aborted the application. The new OpenMPI runtime, introduced in the 5.0.0 version, sets the restarted process communicator as malfunctioning internally, making subsequent MPI operations fail and rendering the socket unusable.

After the attempts described above, it is even more apparent that a transparent approach to reach C/R in OpenMPI would be tricky to reach without complete holistic support of such a feature internally to the OpenMPI runtime.

3.5. Moving to application checkpointing

Although the need for a completely transparent solution to the problem of checkpoint/restart in a parallel environment remains strong, the conditions of the existing tools prove to make it currently unfeasible:

- CRIU explicitly states that it does not support distributed frameworks such as MPI; other sources proved that it could be used for C/R in MPI contexts wrapped by a container [6], but such examples focus on the restart of the entire MPI application.
- The support for the dynamic process management functions still seems exceptionally raw. In recent findings, [16], it is shown that most MPI operations used are the Point-to-Point ones and the collective. This justifies the low interest and support for

dynamic process management capabilities.

- Other system-level approaches such as DMTCP/MANA proved not general enough and required high effort to configure system parameters and ensure the application is working; moreover, they proved not to be tested and working with OpenMPI.

The above reasons force the re-evaluation of the requirements initially outlined in the current chapter. In particular, two requirements are affected:

- Transparency. Given the analysis in the previous sections, we should define an alternative to the transparent checkpoint initial goal. Given that ULFM gives us the primary tooling to repair the communication structures, we focus on the MPI flow, leaving the responsibility of checkpointing to the user.
- Periodicity and checkpoint flow. Given that the system is no more external to the application, our work should have no control over when the checkpoint is taken. Instead, the application developer will have complete control over the frequency and position of checkpoints.

4 | Application checkpointing

This chapter continues the journey by using application checkpoint to reach non-transparent fault resiliency and recovery. In the context of embarrassingly parallel MPI applications, we evaluate and propose a framework built on top of Legio and ULFM.

Section 4.1 gives an overview of application checkpointing in the context of MPI applications, outlining a high-level solution to the problem at hand. Section 4.2 explores the initialization phase, while Section 4.3 digs deeper into failure detection. Finally, Section 4.4 concludes the flow with details about the restart phase, and Section 4.5 lists the limitations of the current approach.

4.1. Overview

Application-level checkpoint is the alternative approach to system-level (which includes both kernel and user-level checkpoint/restart). At its core application checkpoint leaves the job of saving (and restoring) the needed variables to the developer - while the system-level checkpoint aims at full transparency in the process. The chapter explores the application checkpoint as a failure recovery mechanism for critical processes and outlines a framework that allows restoring the MPI structures (such as communicators) transparently.

Section 4.1.1 explores the application checkpoint frameworks in MPI distributed applications and compares them to system checkpoints. In contrast, Section 4.1.2 outlines a high-level design of the proposed solution in the rest of the chapter.

4.1.1. MPI application checkpointing

In the previous sections, the system-level checkpoint was explored, and the difficulty in the feasibility of reaching a functional prototype was clear; in addition to that, application checkpointing has other advantages concerning system-level checkpointing:

- It's common for a system-level checkpoint to have to checkpoint the entire group of MPI processes in the application, to reach a consistent checkpoint. Due to MPI being unable to dump and restore from file low-level structures automatically, memory content must be dumped periodically for all the parallel processes. This is not needed in application checkpointing when only the most critical processes in the critical paths (where an error is most likely to occur) can be checkpointed. The application awareness given from the application checkpoint can help minimize the overall overhead in terms of time and disk usage.
- In case peak memory usage and checkpoint time collide, the risk is saving a lot of data that is not needed at all. Low-level details about an ongoing computation could be far more accessible to re-compute on-demand after the restore. The application developer chooses each checkpoint's granularity, frequency, and criticality. It is then possible to easily integrate multi-level checkpointing where the likelihood of failure influences the storage system of the checkpoint, following a similar approach as Scalable Checkpoint/Restart [19].
- System-independent; although many system-level checkpoint/restart alternatives aim to be highly portable, it's natural that the application run-time environment must be adapted. In the case of application checkpoints, the only requirement is a storage layer that allows to dump and restore checkpoints.

A common approach, suggested for example in Reinit [14] when it comes to application C/R is to modify the application flow into a loop, or series of loops. The architecture of a similar program reflects the embarrassingly parallel nature of the MPI application. The approach is incredibly efficient since it enables an iterative checkpoint of the minimum data necessary to restore the application.

4.1.2. High-level design

In the context of the journey of the thesis, what we are most interested in is the ability to restore, after a failure, a process together with its MPI communication abilities without removing any MPI assumption from the developer using the C/R framework. For this reason, saving and restoring application-specific variables is outside the scope of the framework. To enable application developers to test the framework, a simple function that checks whether the current process is restarted or the original is provided. It is then easy to expand it with complex dump/restore logic. Considering the importance of Legio in the current thesis, the proposed framework will be called in the rest of the thesis Legio++.
At the high level, the main parts of the application in which our framework plays an

important part are three:

1. Initialization - the application code should use specific routines to ensure our framework can keep track of the MPI structures.
2. Failure detection - during process failure, the framework will notice a fault and act accordingly based on the criticality of the failed process. Failure resiliency without recovery is already implemented and mainly based on Legio, but it needs a revamp in the context of possible restarted processes.
3. Restart phase - the routine that enables a failed process to be restarted and re-join the existing group of MPI processes is at the core of the thesis work.

To have a solid basis with fault resiliency built in, the recovery framework is built as a modification of the Legio framework. The following sections will dig deeper into each one of these three points, explaining the taken solution and problems faced on the way.

4.2. Initialization phase

This section explores the initialization phase of the Legio++ framework. The MPI objects created during this phase will support resiliency and recovery since Legio will transparently intercept and repair subsequent failures to the user.
Section 4.2.1 explores the `MPI_Init` routine call and the additions to ensure the functioning of the application in case of failure, while Section 4.2.2 outlines the initialization of failure-resistant communicators.

4.2.1. MPI_Init

The starting phase of an MPI application is marked from the `MPI_Init` routine call, which initializes the internal MPI environment of the current process. It must be called by all the processes interacting with MPI. As described in the last chapters, Legio intercepts the MPI calls with the PMPI interface, so it is possible to initialize additional structures during the MPI initialization call.
`MPI_Init` initializes the most important structures internal to the framework:

- `ComplexComm`. A structure that wraps A communicator, ensuring that failures are not exposed to application developers. It ensures that, even after an error, operations on communicators still work flawlessly. For example, the rank of the current process will not change for a communicator in which a failure happened. During initialization, two `ComplexComm` are initialized, containing the following

communicators: MPI_COMM_WORLD and MPI_COMM_SELF.

- MultiComm. It contains a map of ComplexComm and all the necessary operations to interact with them (such as retrieving one starting from a MPI_Comm coming from the application).

Legio++ additionally parses a series of command-line arguments to enable customization at run-time of the behavior of the application:

- List of critical processes that should be restarted, specifying their rank. In case no process is specified, the application fallbacks to fault resiliency.
- (Restarted process only) List of failed processes. A failed process must know during startup the series of processes that already failed processes to avoid performing operations with them.

4.2.2. Communicators

Legio does manage both MPI_COMM_WORLD and MPI_COMM_SELF, the two communicators created during initialization, but also the other ones created during the application flow. As a result, communicators' values can change (for example, overwriting an existing communicator with a new one with different processes inside). This poses a risk in tracking their usage and restoring them in case of critical failures. For example, consider this situation:

```
MPI_Comm new_comm;
MPI_Comm_create_group(MPI_COMM_WORLD, group_1, tag_1, &new_comm);
/* ... do first computation ... */
MPI_Comm_create_group(MPI_COMM_WORLD, group_2, tag_2, &new_comm);
/* ... do second computation ... */
```

It's clear that the value of the communicator changes between the first and the second computation, but the application, when restarting, cannot know which one is the correct one.
To overcome this risk, a simple solution is adopted: the creation of (failure-resistant) communicators must be done through a specific function exposed from the library, initialize_comm(int n, const int *ranks, MPI_Comm *newcomm):

- n defines the number of ranks inside the communicator
- ranks is an array of size n containing an ordered list of ranks that should be included in the communicator

- newcomm is the pointer to the communicator object in which the newly created communicator will be saved.

The communicators initialized with the function above are saved together with the ranks they are included in them in an internal structure. When a failure occurs, the related rank is saved in internal structures. Having the initial and failed processes for each communicator, we can ensure that local functions such as `MPI_Comm_Rank` and `MPI_Comm_size` work in case of failures and restart.
The standard function `MPI_Group_translate_ranks` was enough before the introduction of failure recovery to maintain transparency in retrieving the ranks and size of a communicator. However, with dynamic process creation, MPI cannot translate any more ranks from the `alias` communicator to the new one. Therefore, a correspondent rank in the `alias` can differ from that of the newly created communicator. For example, in the case of newly spawned processes, the new rank in the new communicator is not present in the `alias`, so it will produce an `MPI_UNDEFINED`. To achieve the same goal, we implemented an internal `translate_ranks` which is exposed directly from the `MultiComm` structure and uses the internally saved information about failed processes to translate the ranks.

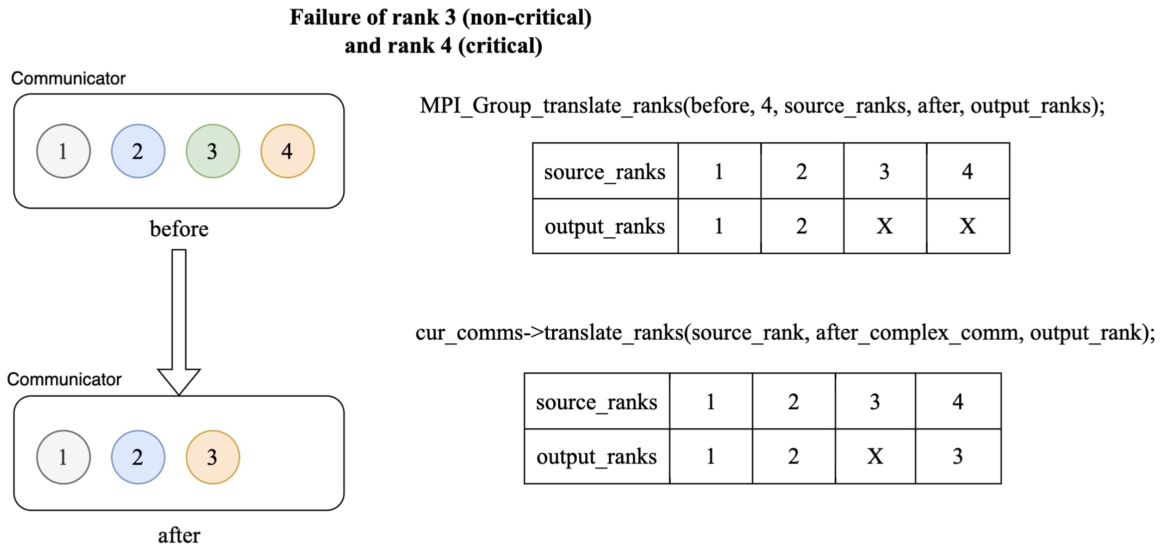

Figure 4.1: The figure shows the difference between the `MPI_Group_translate_ranks` and the custom `translate_ranks` in the presence of failed and restarted processes. The rectangles on the left represent the same communicator before and after two failures: rank 3 (non-critical, therefore not repaired) and rank 4 (critical, therefore repaired). The number in the circles represents the rank's number according to the related communicator. Circles with the same colors indicate that the rank is equal for the internal `MPI_COMM_WORLD`. On the right, the two operations performed after the failure and with the results.

Figure 4.1 shows the value created from the custom `translate_ranks` function. In the presence of non-critical and/or critical faults, it is able to translate the ranks according to the `alias` to the internal communicator. Analyzing the situation, we start from the `before` communicator with 4 ranks. Supposing that rank 3 (non-critical) and rank 4 (critical) fail, the result communicator is composed of three ranks. Using the `translate_ranks` it is then possible to convert the ranks of the `alias` to the alias of the internal communicator correctly.

4.3. Failure detection phase

When an MPI operation detects a process failure, the error code `MPIX_ERR_PROC_FAILED` is returned. Legio handles the behavior gracefully by detecting the problem, shrinking the communicator that caused the problem removing the failed process, and repeating the operation. The flow works flawlessly since only the processes involved in the communication

are affected and should join in the clean-up of the communicator.
In the high-level design envisioned in the journey, the old flow has two main drawbacks. The first one is that failure detection and resiliency are applied only to one communicator simultaneously. Therefore, if a process fails, every time a communicator containing the process is used, the reparation would restart. This adds complexity to the flow since not all communicators agree on the same status of the aliveness of the processes when the application is running. Different communicators may see some processes as failed or not based on whether a remote MPI communication has run on them after the failure.
The second is that the failure recovery operation cannot work reliably without the full participation of all processes in the MPI application. When recovering the state and restarting an MPI process, reaching a consistent `MPI_COMM_WORLD` between the alive processes and the restarted one is crucial. In case processes do not participate in the restart procedure, this goal is unfeasible due to how the dynamic process creation works. Without the participation of all the processes in the re-spawning of failed processes, it would not be possible for processes to contact each other. This would make it necessary to repeat repair procedures in case they will be needed, paying an extra cost in terms of latency overhead.
To solve the two problems above, the flow is slightly modified by ensuring that the resiliency procedure (the shrinking of the communicator) and the restart procedure (described in the next section) are performed from all the processes in the main MPI application. To achieve this goal, we implement the following high-level steps:

1. During initialization, a thread is started that probes for messages with a specific tag.
2. When a failure is detected, a message is sent to the list of alive MPI processes inside `MPI_COMM_WORLD`, with a particular tag allowing the thread to read the message.

To expand on the first point, we implemented the spawn of a demonized thread that runs until the application ends. The main goal is to listen on the (failure resistant) `MPI_COMM_WORLD` to wait for messages indicating that a failure happened, and the process must participate in the resiliency/recovery process. It is necessary to notice a failure from all the processes in the application since reparation of failures in case of fault recovery. When spawning a new process in place of a failed one, all the survivor processes must participate. Otherwise, the risk is noticing the fault from a group of disjoint ranks, which could spawn the failed processes twice or more. Since a thread is used, the routine `MPI_Init_thread` replaces calls to `MPI_Init`.
The thread uses the `MPI_Probe` command to check if any messages are sent to the communicator, filtering them with a specific tag to ensure that application messages are not considered. The receival of one of these messages indicates the possibility of

having to start the failure-reparation mechanism. In case a message is received, the `MPI_Recv` command is used to read the content and start the restart mechanism, which will be described in Section 4.4. In both cases, an exclusive mutex lock is used before accessing `MPI_IProbe` and `MPI_Recv`. This is necessary since the main application could detect a failure and change the internal communicator used from the `ComplexComm` of `MPI_COMM_WORLD`. To avoid this behavior, a mutex is used to ensure no change happens while probing for failure messages. If no message is probed, the threads sleep for a configurable amount before repeating the same steps. The choice for the sleeping time between each cycle is a trade-off between velocity in detecting failure and overhead in probing continuously.

With this change in the flow, every alive process inside `MPI_COMM_WORLD` will participate in the procedure, either in case of resiliency (for non-critical processes) or recovery (for critical processes), ensuring that after the procedure, each process will have a consistent state for the present communicators. A significant difference with respect to Legio's solution is that a failure of a single process will be detected and acted upon only once, by repairing all the necessary MPI communicators. In Legio, on the other hand, the failure was detected only by processes acting on the communicator with the failed rank.

4.4. Restart phase

The core of the solution is the restart phase, which ensures that after the procedure is completed for each rank, the application can continue working and operating with MPI without any problems.

This section starts with Section 4.4.1, outlining the assumptions before going inside the restart phase. Then, in Section 4.4.2, the main flow for resiliency and recovery is explained in detail. Finally, Section 4.4.3 explores the custom flow in the restarted process of rebuilding MPI structures.

4.4.1. Assumptions

The primary assumption built upon the modifications in the previous sections is the synchronization of all ranks during the restart phase. To achieve this, a separate thread runs in every MPI process, ready to manage failure messages from other ranks. This creates two different points at which the reparation routine can start: the process which detects the failure in the main thread and the secondary threads which receive the message prompting to participate in the routine. This makes it necessary to create safeguards against a possible deadlock or race condition issues:

- A mutex is used to ensure that only one single thread can enter the recovery procedure; before entering the function, an exclusive lock is taken, ensuring that no race condition can occur by two threads modifying the internal state of MPI structures at the same time.
- MPI operations should not be able to complete while the restart procedure is ongoing - to achieve this, a shared lock is taken every time an MPI operation is ongoing.

Moreover, since different messages starting the restart procedure can arrive multiple times, it must be idempotent; in case no process is failed, it should complete fast, reducing the overhead on the rest of the application.

4.4.2. Resiliency and Recovery

The core of the modification is the actual restart procedure, which ensures that resiliency and recovery are tightly coupled with the detection of failure.

Firstly, the failures present in `MPI_COMM_WORLD` are retrieved using the `MPIX_Comm_failure_ack` function. In case no failure is present, it means that the reparation has already been completed. Therefore the reparation procedure can terminate.
After, failed ranks are compared with the critical processes parsed during initialization time; if no process must be restarted, the execution continues. Otherwise, the list of processes that are failed and must be restarted are retrieved and restarted using the `MPI_Comm_spawn_multiple` routine, re-running the original process. Additional arguments are passed to the process, such as the lists of critical and failed processes; in this way, the restarted process can re-create the structures present before the failure. It is also essential for the restarted process to have `MPI_COMM_WORLD` as an intra-communicator containing all the ranks present in the MPI applications; this is different from what happens in a spawned process through `MPI_Comm_spawn_multiple`, where `MPI_COMM_WORLD` is the group of the respawned processes. To achieve this, the following steps are taken:

1. The `MPI_Intercomm_merge` routine is used to merge the inter-comm between the spawned processes (which is their parent) and the intercommunicator produced from the `MPI_Comm_spawn_multiple` in the processes that perform the routine. The result is an intra-comm that contains all the processes (the ones that respawned and the ones which started the respawn).
2. The `MPI_Comm_split` is used to re-order the ranks, ensuring they are consistent with the ordering before the failure.

3. The generated communicator is set as the new `MPI_COMM_WORLD`, directly inside the `ComplexComm` structure.

After the restart, the steps taken are identical between recovery and resiliency situations. The list of communicators is iterated, and each is re-created, considering only the remaining non-failed ranks. This ensures that all communicators are now free from failures, and the control can be passed again to the application.

4.4.3. Restarted process and MPI structures

During the recovery procedure, it is necessary that also the restarted process(es) participates in it since operations to create communicators are blocking all the processes involved. Therefore, the operations done on communicators must be synchronized and ordered correctly. The following changes in the flow for a restarted process are then applied:

- When the `MPI_Init` routine is called, the process must call the `MPI_Intercomm_merge` and `MPI_Comm_split` procedures, synchronized with the alive processes, to re-create a consistent `MPI_COMM_WORLD`.
- The `initialize_comm` function call must behave differently in case of the restarted process since ranks that are passed may be failed. To achieve this, the work follows these steps:
    1. Retrieve the ranks in the `MPI_COMM_WORLD` and the communicator to create which are not failed and should be used in the process.
    2. Use the custom `translate_ranks` to translate ranks from the communicator (the `alias` for the restarted process) to the ranks according to the newly created `MPI_COMM_WORLD`.
    3. Create a communicator by using `PMPI_Comm_create_group` and passing the ranks found at point 2.
    4. Save the newly created communicator internally, with the related information about the currently alive and failed ranks.

4.5. Known limitations

The approach designed above proved to be working in re-constructing the MPI structures in case of failures (either critical or non-critical, or a sequence of a combination of both). One drawback noticed is in the context of multiple failures, where it is not easy to assume that

all processes will recognize the same list of failed ranks. This could mean race conditions when restarting processes and cause application failure. Note that this case has been considered rare since the reparation procedure should be relatively fast and the likelihood of more than one contemporary failure is low.
Another found limitation is the best effort in the `MPI_Iprobe` routine, which does not necessarily detect messages sent from other ranks promptly; this can cause a slowdown during the reparation procedure, and a solution has not yet been found.
Finally, it is essential to note that full control is left to the application developers, which should ensure that the application checkpoints the current state before each MPI operation. This is important to have consistent checkpoints across the different processes and avoid deadlock in the presence of collectives where some ranks cannot participate because they are involved in various MPI operations. An example of such a situation is the following:

```
1 // 0 is a critical process
2 if(rank == 0)
3     raise(SIGINT);
4 else
5     MPI_Barrier(MPI_COMM_WORLD);
6
7 MPI_Barrier(MPI_COMM_WORLD);
```

In the simple example above, if rank 0 is a critical process, Legio++ will restart it after the failure injected through the `SIGINT`. After restart, rank 0 will perform the second barrier, while the rest of the ranks in the application will complete the first one. In this case, a deadlock will occur since every survivor process will indefinitely wait for rank 0 at the second barrier, which has already been performed. Such situations can be common in complex applications with many communications but should be manageable in embarrassingly parallel applications. Moreover, Legio++ exposes APIs to fix the above problem:

```
1 // 0 is a critical process
2 if(is_respawned() && rank == 0)
3     raise(SIGINT);
4 MPI_Barrier(MPI_COMM_WORLD);
```

By simply injecting the failure only in rank 0 if it is not respawned, it is possible to avoid a deadlock.

5 | Experimental Evaluation

This chapter introduces the experimental results of the application checkpointing framework defined in chapter 4, with the main goal of measuring the overhead given by a dynamic restart.
Section 5.1 outlines the purpose and environment in which the experiments are run; Section 5.2 explores the overhead focusing on the main restart routine, while Section 5.3 analyzes further the overhead given from the entire checkpoint/restart flow. Finally, Section 5.4 wraps up the chapter with the conclusions.

5.1. Experimental setup

During the validation of a framework for HPC computing, checking edge cases and whether the application works as expected is as essential as evaluating potential overheads, which could make it unfeasible to run at the desired scale. Given this, and following the requirements defined in Section 3.1, we measure the overhead in different conditions in an HPC environment for our framework.
The experiments conducted are two folds - first, we focus on the overhead of the restart of an MPI process. Afterward, we analyze a more holistic, embarrassingly parallel application to review the tradeoffs of recovery versus resiliency better.
We conducted both experiments on Antarex, a single node at Politecnico di Milano, featuring 2 x Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz processors and 128 GB of RAM. We also used Karolina, a supercomputer with 829 computational nodes, for the parallel application campaign. Each node is a powerful x86-64 computer, equipped with 128/768 cores (64-core AMD EPYC™ 7H12 / 64-core AMD EPYC™ 7763 / 24-core Intel Xeon-SC 8268) and at least 256 GB of RAM. For our experimental campaign, we used 2 nodes with 128 cores.

5.2. Restart

The first experimental campaign focuses on understanding the impact of the restart operation and its overhead, described in chapter 4. The following code snippet featuring the `MPI_Barrier(MPI_COMM_WORLD)` highlights the operation in which overhead has been evaluated.

Below is a code snippet exemplifying the benchmark:

```
// if the process is not already respawned, raise a SIGINT; otherwise, continue
if(!is_respawned() && rank == 0)
    raise(SIGINT);

// measure barrier with a failure
start = MPI_Wtime();
MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();
```

As shown above, the `SIGINT` signal is injected in one of the processes, making it impossible to continue and abort immediately. Therefore, the `MPI_Barrier` right after will not succeed immediately - making it necessary for the framework to act with either resiliency or recovery. To distinguish whether the given rank (in the above case, the rank 0) should be respawned or the application should continue without it, it is sufficient to add the related command-line argument with the lists of critical processes. If rank 0 is critical, the application will initiate a respawn procedure after failure. Otherwise, resiliency will occur, and the application will continue repairing the existing communicators without additional changes.

In both cases, the `MPI_Wtime` functions are used to extract reliable measurements about overhead; for simplicity, it is measured on one rank only, which has no failure during the run. The following type of test is run 40 times each, with the number of processes in the application varying from 8 processes to 256, showing how the overhead changes with the size of the application:

1. Legio, which is the latest version of the Legio framework without any additional changes. In this case, no recovery will be executed, only resiliency.

2. Legio++, considering the failing process (rank 0) as a non-critical process. Therefore, also, in this case, no recovery will be performed.

3. Legio++, considering rank 0 as a critical process, performing a recovery for it.

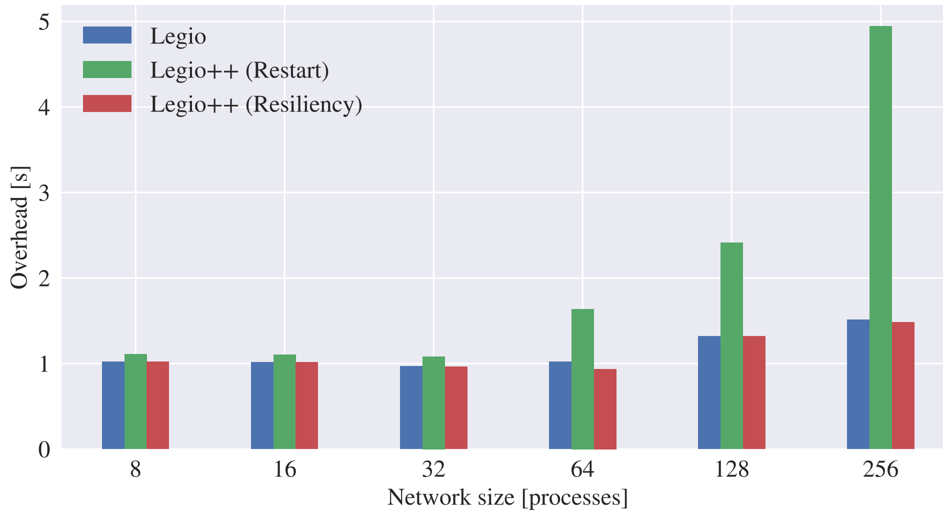

Figure 5.1: Failure recovery time by varying the number of processes in the `MPI_Barrier` operation and the failure tolerance method.

Figure 5.1 show the result obtained following the guidelines described above. It clearly shows that resiliency in Legio++ does not create visible latency overhead concerning Legio. On the other hand, failure recovery through restart has a more significant overhead which grows with a higher factor than resiliency. Two factors can explain this result:

- The network size between 64 and 256 nodes is oversubscribed. This provokes a non-negligent overhead into the MPI operations and could be directly linked to the exponential increase of the overhead.
- A restart includes dynamic process management operations, which are by design costlier.

To further explore this result, we perform a granular benchmark to understand the impact of the various parts of the restart operation; the following times are stacked below, in order of execution:

1. Failure propagation - the time needed to communicate to the alive processes the failure through a `MPI_Isend`, which will be picked up from the secondary thread.
2. Failure ack - time to ack the failure and understand the failed process.
3. `MPI_COMM_WORLD` shrinking - Time to shrink the `MPI_COMM_WORLD` communicator to remove the failed processes.
4. Respawn - Time to restart the failed processes (in our test, one).

5. Restart reparation - Time to re-order the `MPI_COMM_WORLD` ranks, ensuring they are consistent as before the failure.
6. Communicator reparation - Time to repair an extra communicator created during the application's startup as a failure-resistant communicator. For simplicity, the communicator used in the test has the same size as `MPI_COMM_WORLD`.

The test has been run 40 times, and a mean value has been taken for each measurement. Figure 5.2 shows each operation contributes to the restart procedure for a different number of processes in the MPI network - 8, 16, 32, 64, 128.

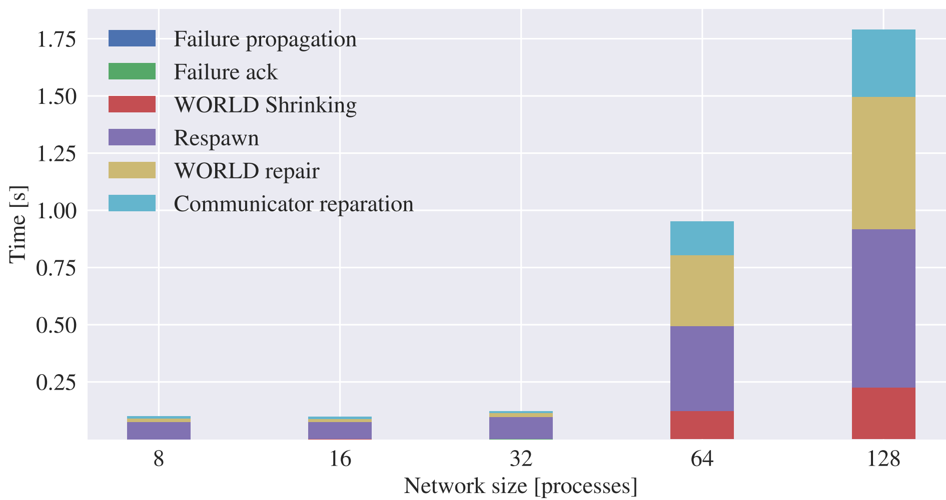

Figure 5.2: Overhead of each operation in the restart procedure, varying by the number of processes involved.

Results outline how the contribution of a dynamic process management operation, such as the spawning of a new process, is heavier with the number of processes involved. Similarly, the reparations of each communicator grow with the number of processes involved, which can be attributed to the over-subscription of processes in the application.
To wrap up the campaign regarding the restart operation, we further analyze the restart time concerning the communicator's reparation. Legio++, differently from Legio, repairs all the communicators after a failure, even if they are not directly involved in the current failure. This achieves consistency in the application regarding the status of failed and alive processes. A possible drawback is a higher overhead if multiple communicators are not used extensively in the application but are still repaired in Legio++. On the other hand, if communicators are frequently used, repairing them once avoids reparation for every use of different communicators with the failed process.
To analyze the situation, we prepare a test that performs a `MPI_Barrier` when one process

fails. In particular, we perform three suites of tests comparing Legio and Legio++, with 10 iterations each, using 32 MPI processes. The suite of tests is as follows:

- 1 barrier - 1 communicator: we consider an application with only one communicator and a `MPI_Barrier`. In this case, the work performed from Legio and Legio++ are similar.
- N communicators - 1 barrier: we consider an application with 100 additional communicators and a `MPI_Barrier`. Legio++ will have to repair all the additional communicators, even if they are not used.
- N communicators - N barriers: we consider an application with 100 additional communicators and N `MPI_Barrier` on each one of the communicators (100 `MPI_Barrier` in total). Legio++ will repair all the communicators only once, while Legio will repeat the failure procedure for each barrier.

The results can be seen in Figure 5.3, confirming a low overhead in any of the three cases reported above. It proves that the resiliency approach taken from Legio++ has no strong performance drawback.

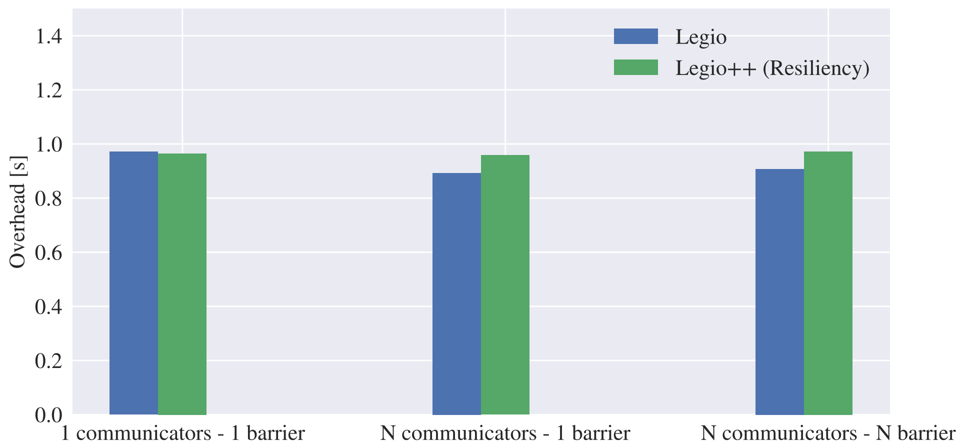

Figure 5.3: Average execution time of the restart procedure, comparing Legio and Legio++ with resiliency. The three tests show how the time varies by changing the number of communicators and synchronization operations on each.

5.3. Parallel application

The second type of experiment comprehends a series of tests on an embarrassingly parallel application: a Montecarlo simulation computing the value of π.

The main interest is evaluating the entire execution time and how it evolves with the

number of processes. Results in the previous sections show that Legio++ improves the management of MPI processes without incurring significant overhead in the case of non-critical processes. Therefore we focus this experimental campaign on Legio++ with critical and non-critical processes.
To run the test, the computation is distributed in a different number of processes: 8, 16, 32, and 64, and tests are repeated for both experiments 10 times.
The experiment showcases the time to completion for a Montecarlo simulation which computes π: random (x, y) points are simulated in a 2D plane, and an approximation of the value of π is computed knowing that the area of a square is $4 * r^2$, while the area of a circle is $\pi * r^2$. To check if a randomly generated point is inside the circle, it is sufficient to check whether $x^2 + y^2 <= r^2$. To test the overhead tied to recovery, the rank 0 (which is the root of the `MPI_Reduce` accumulating the results of the processes) fails in the middle of the computation. For resiliency, a non-critical process (rank 5) fails - reducing the accuracy of the result.
The results can be seen in Figure 5.4, confirming the minimal overhead of restart concerning resiliency considering the major impact of a critical node.

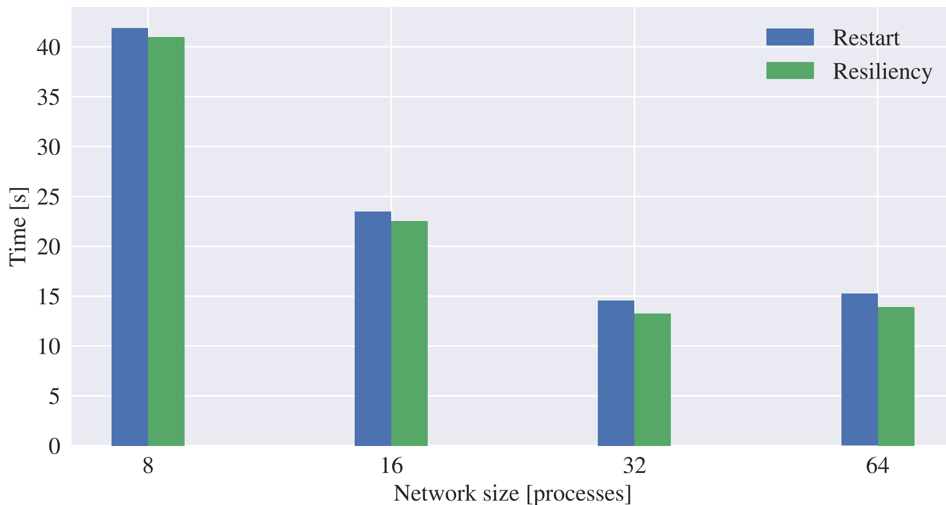

Figure 5.4: Average execution time of a Montecarlo simulation by varying the number of processes involved and the failure tolerance method.

We also evaluate the performances of a parallel Montecarlo application on the cluster Karolina, evaluating only Legio with a restart. We repeat the testing campaign using 64 and 127 cores in one node and 192 and 240 cores in two nodes. The results shown in Figure 5.5 confirm the minimal overhead of fault recovery through the restart and its usability in larger clusters.

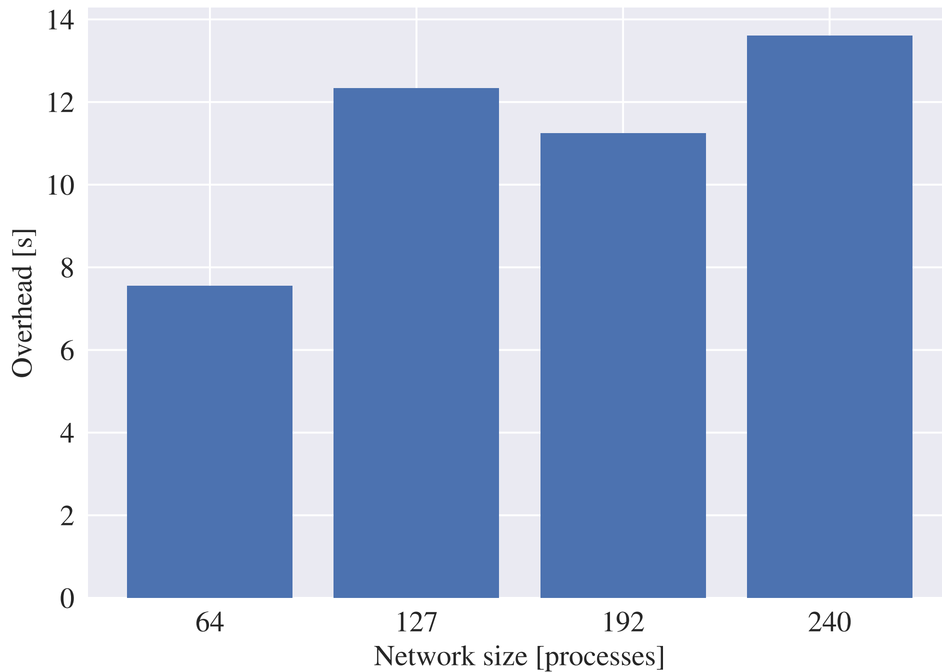

Figure 5.5: Average execution time of a Montecarlo simulation with an injected failure using Legio with fault recovery by varying the number of processes involved.

5.4. Conclusion

The testing campaign proved that the proposed work has contained latency overhead for fault recovery and resiliency. Furthermore, it recognized that the overhead for reparation grows with the number of processes, mainly due to the spawning of new processes and the reparation of the `MPI_COMM_WORLD` communicator. Nonetheless, reparation's benefits to existing applications can be worth it in case of failures in critical nodes. Moreover, we proved that the proactive reparation of communicators does not bring latency overhead and concentrates the restart cost in a single operation.

The developers are in the best position to evaluate whether a process should be marked critical, considering fault recovery's more significant performance overhead than fault resiliency. Therefore, Legio++ gives its users the necessary flexibility to make this choice autonomously.

6 | Future Work and Conclusions

This chapter will discuss the next steps in the journey of fault tolerance and summarize the results achieved in this thesis.
Section 6.1 focuses on the transparent user-level checkpoint, while Section 6.2 explores possible improvements in application checkpointing. Finally, Section 6.3 wraps up the work with the conclusions.

6.1. Transparent checkpointing next steps

The first goal was the introduction of a transparent checkpoint/restart framework for MPI applications. Then, the overhead of the C/R could have been applied only to critical processes, accepting approximate results in case of non-critical failures. But unfortunately, complex technological limitations and unexpected failures made the task impossible. The direction and research path are still open, with the actively developed tools referenced in chapter 3 trying to solve the problem with a different scope described below.
The system-level checkpoint approach in MANA, which focuses on separating MPI and application-level memory during a checkpoint, is promising; the priorities of the team working on it are tied to their internal systems. Therefore a possible effort could entail a deeper study of the memory checkpoint model adopted in MANA and trying to port it to other platforms.
The most encouraging tool remain CRIU; its features are apparent, and the support for a wide range of systems and network (such as full TCP support) makes it a perfect candidate. To continue the work in that direction, it would be possible to act in two paths: containerization and modifications to the new MPI runtime. Regarding the first, CRIU has been proven to work in the MPI context with single containers in a stop-and-restart. Although it would relax the constraint of a local restart and incur additional overhead, containerizing an MPI application could be easy (and maybe already done in existing applications); it should then be explored whether the overhead incurred for stopping the processes during the checkpoint is acceptable, and directly integrate the checkpoint framework inside Legio itself. Finally, regarding the MPI runtime, significant

steps have been made recently with the new PMIx runtime in OpenMPI 5.0; this could open new possibilities by making possible plugins to the runtime, which handle the low-level rejoining (without direct MPI routines) of a process after it has been respawned with CRIU. Unfortunately, the integration between ULFM and the new runtime is still at its early stage at the time of writing, and work is underway to better document how dynamic process management can be used together with ULFM.

6.2. Application checkpointing next steps

The proposed work, Legio++, goes toward minimizing the cost for applications that can be easily checkpointed (or already are) at the application level and easily abstract the checkpoint about MPI structures and communicators. Different improvements could be made to this path, such as providing a more ergonomic API for application checkpointing and integrating the support with the new OpenMPI 5.0 version once a stable version is released. The two future work paths will be described below.
The respawn flow proposed in chapter 4 mainly focuses on the restart of the process by using dynamic process management features. Therefore the ergonomics and ease of use of the APIs exposed from the library to perform a checkpoint/restart could be greatly improved. To support existing applications which already use checkpoint, without any dynamic flow for restarting the application in case of crashes, might be integrating and supporting commonly used libraries for checkpoint in scientific applications, such as FTI, which allows for multi-level checkpointing offering flexible configurations to choose the best checkpoint strategy to minimize overhead.
With an active working group focusing on Fault Tolerance in the MPI domain, the ULFM framework is being revamped - it's even more important to confirm the support of the work of this thesis with newer MPI versions. The MPI Forum has been working on a new coarse-grained fault-tolerance interface since September 2022 (after the completion of the project work of this thesis) in the context of Reinit [9, 14]. Its main goal is to ensure MPI can automatically recover from failures, without any additional code from the developer, by leveraging the existing application checkpoint/restart model.
Moreover, the recent revamp in OpenMPI 5.0 of the runtime aims to have better fault-detection methods and revamp the dynamic process management.
The changes above go in the direction of ensuring that the MPI fault tolerance will be battle-tested and ready for production usage.

6.3. Conclusions

This thesis presents a journey through fault tolerance in MPI applications, starting with the ambitious goal of full transparent checkpointing and finishing with automatic failure recovery in all process ranks when a failure is detected.

The thesis starts with an extensive attempt to integrate user-level C/R techniques with MPI, aiming to reduce the involvement of the application developer to zero and bring failure recovery to an existing application without changes.

After evaluating the difficulties in reaching the goal, we propose Legio++, a prototype that allows application developers to tolerate failures without significant changes to the code for what regards MPI usage; we shield the behavior of MPI under failure by making the application developer unaware of the underlying failures of processes.

The experimental evaluations demonstrate that, although the latency and disk overhead of fault recovery is greater than fault resiliency, the trade-off is acceptable regarding critical processes.

The problem and challenges to deal with MPI failures remain, and breakthroughs are still needed to make dependability a first-class citizen in exascale architectures. However, the MPI Forum is working on the challenge. Although the proposed solution is not general enough to solve fault recovery problems, it leverages C/R techniques without involving the developer in adding code to detect and handle low-level MPI failure details.

Bibliography

[1] Fastfreeze: checkpoint/restore for applications running in linux containers, 2020. URL https://github.com/twosigma/fastfreeze.

[2] Mpi: A message-passing interface standard, version 4.0, 2021. URL https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[3] M. M. Ali, P. E. Strazdins, B. Harding, and M. Hegland. Complex scientific applications made fault-tolerant with the sparse grid combination technique. *The International Journal of High Performance Computing Applications*, 30(3):335–359, 2016. doi: 10.1177/1094342015628056. URL https://doi.org/10.1177/1094342015628056.

[4] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, pages 1–12, Rome, Italy, 2009. IEEE.

[5] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pages 1–32, 2011.

[6] G. Berg, M. Brattlöf, A. Blanche, and T. Lundqvist. Evaluating distributed mpi checkpoint and restore using docker containers and criu. 01 2019.

[7] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013. doi: 10.1177/1094342013488238. URL https://doi.org/10.1177/1094342013488238.

[8] R. H. Castain, J. Hursey, A. Bouteiller, and D. Solt. Pmix: Process management for exascale environments. *Parallel Computing*, 79:9–29, 2018. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2018.08.002. URL https://www.sciencedirect.com/science/article/pii/S0167819118302424.

[9] S. Chakraborty, I. Laguna, M. Emani, K. Mohror, D. K. Panda, M. Schulz, and

H. Subramoni. Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications. *Concurrency and Computation: Practice and Experience*, 32(3): e4863, 2020. doi: https://doi.org/10.1002/cpe.4863. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4863`. e4863 cpe.4863.

[10] G. Da Costa, T. Fahringer, J. Rico-Gallego, I. Grasso, A. Hristov, H. Karatza, A. Lastovetsky, F. Marozzo, D. Petcu, G. Stavrinides, D. Talia, P. Trunfio, and H. Astsatryan. Exascale machines require new programming paradigms and runtimes. *Supercomputing Frontiers and Innovations*, 2:6–27, 09 2015. doi: 10.14529/jsfi150201.

[11] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 225–234, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311601. doi: 10.1145/2145816.2145845. URL `https://doi.org/10.1145/2145816.2145845`.

[12] M. Gamble, R. Van Der Wijngaart, K. Teranishi, and M. Parashar. Specification of fenix mpi fault tolerance library version 1.0. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[13] R. Garg, G. Price, and G. 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, page 49–60, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366700. doi: 10.1145/3307681.3325962. URL `https://doi.org/10.1145/3307681.3325962`.

[14] G. Georgakoudis, L. Guo, and I. Laguna. Reinit++: Evaluating the performance of global-restart recovery methods for mpi fault tolerance. In P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, editors, *High Performance Computing*, pages 536–554, Cham, 2020. Springer International Publishing. ISBN 978-3-030-50743-5.

[15] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics. Conference Series*, 46, 9 2006. doi: 10.1088/1742-6596/46/1/067.

[16] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana. A large-scale study of mpi usage in open-source hpc applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[17] N. Losada, I. Cores, M. J. Martín, and P. González. Resilient mpi applications using an

application-level checkpointing framework and ulfm. *The Journal of Supercomputing*, 73(1):100–113, 2017.

[18] N. Losada, P. González, M. J. Martín, G. Bosilca, A. Bouteiller, and K. Teranishi. Fault tolerance of mpi applications in exascale systems: The ulfm solution. *Future Generation Computer Systems*, 106:467–481, 2020.

[19] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, page 1–11, USA, 2010. IEEE Computer Society. ISBN 9781424475599. doi: 10.1109/SC.2010.18. URL `https://doi.org/10.1109/SC.2010.18`.

[20] S. Pauli, P. Arbenz, and M. Kohler. A fault tolerant implementation of multi-level monte carlo methods. Report, Zürich, 2013. See also: http://e-citations.ethbib.ethz.ch/view/pub:151723.

[21] S. Pauli, P. Arbenz, and C. Schwab. Intrinsic fault tolerance of multilevel monte carlo methods. *Journal of Parallel and Distributed Computing*, 84:24–36, 2015. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2015.07.005. URL `https://www.sciencedirect.com/science/article/pii/S0743731515001239`.

[22] A. Reber. Criu: Checkpoint/restore in userspace, 2012. URL `https://criu.org`.

[23] A. Reber and P. Vaterlein. 'checkpoint/restore in user-space with open mpi. In *Proceedings of the BW-CAR Symposium on Information and Communication Systems (SInCom 2014), Germany, Villingen-Schwenningen*, pages 50–54, 2014.

[24] R. Rocco, D. Gadioli, and G. Palermo. Legio: fault resiliency for embarrassingly parallel mpi applications. *The Journal of Supercomputing*, 78(2):2175–2195, 2022. doi: 10.1007/s11227-021-03951-w. URL `https://doi.org/10.1007/s11227-021-03951-w`.

[25] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *The International Journal of High Performance Computing Applications*, 19(4):479–493, 2005. doi: 10.1177/1094342005056139. URL `https://doi.org/10.1177/1094342005056139`.

[26] F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein. Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):501–514, 2018.

[27] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2):129–173, may 2014. ISSN 1094-3420. doi: 10.1177/1094342014522573. URL `https://doi.org/10.1177/1094342014522573`.

[28] P. E. Strazdins, M. M. Ali, and B. Debusschere. Application fault tolerance for shrinking resources via the sparse grid combination technique. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1232–1238. IEEE, 2016.

[29] K. Teranishi and M. A. Heroux. Toward local failure local recovery resilience model using mpi-ulfm. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, page 51–56, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328753. doi: 10.1145/2642769.2642774. URL `https://doi.org/10.1145/2642769.2642774`.

[30] K. Teranishi and M. A. Heroux. Toward local failure local recovery resilience model using mpi-ulfm. In *Proceedings of the 21st european mpi users' group meeting*, pages 51–56, 2014.

[31] Y. Xu, Z. Zhao, R. Garg, H. Khetawat, R. Hartman–Baker, and G. Cooperman. Mana-2.0: A future-proof design for transparent checkpointing of mpi at scale. In *2021 SC Workshops Supplementary Proceedings (SCWS)*, pages 68–78, 2021. doi: 10.1109/SCWS55283.2021.00019.

List of Figures

2.1 The figure shows sets operation on MPI_Groups: MPI_Group_union, and MPI_Group_intersection. The rectangles represent different MPI_Groups, while the number inside the circle is the rank according to MPI_COMM_WORLD. 9
2.2 The figure shows an example of the MPI_Group_translate_ranks. rectangles represent two MPI_Group. The number in the circles is their rank according to the group, while two circles with the same color are the same process. MPI_UNDEFINED are indicated with X for brevity. 10
2.3 The figure shows the behaviour of the MPI_Comm_split operation. On the right a communicator containing nine processes. Each one of the processes calls the split operation with the color and key, as shown in the figure. The operation's output is the set of three communicators on the left. 12
2.4 The figure shows the difference between shrinking and non-shrinking operations on communicators after a failure. Taken from [18] 24

3.1 The figure shows the checkpoint operation in DTMCP in two nodes with 2 and 1 threads respectively. Taken from [4] 32
3.2 The figure shows the restart operation in DTMCP in two nodes with 2 and 1 threads respectively. Taken from [4] 33

4.1 The figure shows the difference between the MPI_Group_translate_ranks and the custom translate_ranks in the presence of failed and restarted processes. The rectangles on the left represent the same communicator before and after two failures: rank 3 (non-critical, therefore not repaired) and rank 4 (critical, therefore repaired). The number in the circles represents the rank's number according to the related communicator. Circles with the same colors indicate that the rank is equal for the internal MPI_COMM_WORLD. On the right, the two operations performed after the failure and with the results. 48

5.1 Failure recovery time by varying the number of processes in the MPI_Barrier operation and the failure tolerance method. 57

5.2 Overhead of each operation in the restart procedure, varying by the number of processes involved. 58
5.3 Average execution time of the restart procedure, comparing Legio and Legio++ with resiliency. The three tests show how the time varies by changing the number of communicators and synchronization operations on each. 59
5.4 Average execution time of a Montecarlo simulation by varying the number of processes involved and the failure tolerance method. 60
5.5 Average execution time of a Montecarlo simulation with an injected failure using Legio with fault recovery by varying the number of processes involved. 61

Acknowledgements

First, I am extremely grateful to my supervisors, Gianluca and Roberto, for their valuable time, support, and patience during my thesis.
This endeavor would not have been possible without my classmates. Thanks for the support during this long journey.
Finally, I would like to express my gratitude to my family, with a special mention to my parents, Giuseppina and Vittorio. Your belief in me has kept my motivation high, and I will be forever thankful.