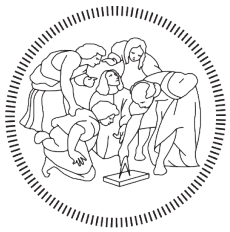

POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
CORSO DI LAUREA MAGISTRALE IN COMPUTER SCIENCE AND ENGINEERING

LEGIO: FAULT RESILIENCY FOR EMBARRASSINGLY PARALLEL MPI APPLICATIONS

M.Sc Thesis of:
Roberto Rocco

Supervisor:
Prof. Gianluca Palermo

Co-supervisor:
Ph.D. Davide Gadioli

Academic year 2020/2021

Abstract

DUE to the increasing size of HPC machines, the fault presence is becoming an eventuality that applications must face. MPI provides no support for the execution past the detection of a fault, and this is becoming more and more constraining. With the introduction of the User-Level Fault Mitigation library (ULFM), it is now possible to repair the execution, but it needs changes in the applications' code. Moreover, ULFM isn't easy to introduce in a generic MPI application as it needs deep knowledge of the problem it's trying to solve. That is the reason why ULFM has been integrated into many frameworks that encapsulate the repair path and provide an easier to use API.

Our effort proposes the Legio library, a system to easily introduce fault resiliency in an embarrassingly parallel MPI application. The focus was towards a transparent approach: the application needs almost no code changes to support Legio. The repair solution was also different from most of the already developed frameworks: while their focus is towards the recreation of the failed nodes and the restoration of a correct status, Legio bases on a shrinking approach that discards the failed nodes and lets the others proceed. This solution is expected to provide better performance in the recovery phase but will affect the accuracy of the result: a trade-off that many embarrassingly parallel applications are willing to do, like for example Monte Carlo approximators. We also the first evolution of the Legio library which includes a new communication structure that transparently reduces the size of the communicators used by the application.

We tested our solutions on the Marconi100 cluster at CINECA, showing that the overhead introduced with the library is negligible and it doesn't limit the scalability properties of MPI. We also tested the library on real applications by injecting faults, to further prove the robustness of the solution. Finally, we discussed a further evolution, which can include C/R and achieve local recovery for a complete framework.

Sommario

DATO l'aumento delle dimensioni dei cluster HPC, l'insorgere di guasti sta diventando un evento da considerare. Lo standard MPI non gestisce il comportamento dell'applicazione dopo l'insorgere di un guasto, e questo sta cominciando ad essere un forte limite agli sviluppi di architetture più potenti. Con l'introduzione della libreria ULFM è possibile gestire la presenza di guasti, riparando le strutture coinvolte e tornando ad uno stato corretto. Purtroppo, la maggior parte delle applicazioni MPI non è stata progettata considerando ULFM e adattare il codice non è un compito semplice. Queste difficoltà hanno dato alla luce dei framework che racchiudono le funzionalità proposte da ULFM ma espongono una semplice interfaccia per l'utente.

Il nostro lavoro ha portato alla creazione di Legio, una libreria in grado di introdurre resilienza ai guasti in applicazioni MPI imbarazzantemente parallele. Uno degli obiettivi principali era la trasparenza: l'integrazione con l'applicazione MPI deve avvenire senza cambiamenti nel codice. Abbiamo adottato anche un approccio alternativo per la modalità di riparazione: se la maggior parte dei framework presenti si concentra sulla rigenerazione dei processi guasti, in Legio abbiamo optato per la riparazione senza sostituzioni. Questa soluzione riduce il tempo di riparazione ma danneggia anche la correttezza dell'applicazione, che produrrà un risultato meno accurato: questo compromesso è comunque accettabile in alcune applicazioni supportate, come ad esempio nelle approssimazioni Monte Carlo. Abbiamo prodotto anche una seconda versione che ridefinisce in modo trasparente la struttura della comunicazione in MPI, per ridurre la dimensione dei comunicatori.

Abbiamo eseguito dei test sul cluster Marconi100 gestito da CINECA, mostrando come il costo di utilizzo di Legio sia trascurabile. Abbiamo inoltre testato la robustezza di applicazioni reali usanti la nostra libreria iniettando dei guasti nel sistema: queste sono state in grado di risolvere i problemi e continuare la propria esecuzione. Abbiamo infine teorizzato l'introduzione di un sistema di checkpoint orientato al recupero locale dell'esecuzione. Abbiamo identificato nel plug-in MANA di DMTCP un possibile candidato per questa integrazione, e abbiamo considerato i vari passi da eseguire per riuscirci.

Contents

CHAPTER 1

Introduction

HIGH-PERFORMANCE COMPUTING (HPC) is the field of computer science aimed at reaching the highest computation capabilities. The field is very heterogeneous since it covers both the development of powerful hardware and efficient software. During the years it evolved a lot to exploit all the innovations coming from other fields. HPC plays an important role in many scientific applications, which need computational power to complete difficult operations.

The high demands of computational science applications are leading the evolution of the current high-performance systems. This increased the complexity of HPC systems to satisfy the need for more performance. As a result, the computation capabilities are growing and will reach the exascale performances (10^{18} FLOPS) in the next years [7, 13]. This evolution is introducing new challenges in the field since problems that were overlooked before are now limiting the performance of the systems. Among these problems, there is system reliability.

The presence of faults in an HPC system may cause errors in the application, and those can stop the execution without obtaining any result. This last consequence limits the efficiency of the system since all the energy spent on the computation is lost without producing anything and the time needed to obtain a meaningful result is increased. The probability of a fault increases with the complexity of the system: while in the past the error frequency was negligible, it is getting so high that applications must account for the presence of errors. And this may pose some problems, since an application may have to be restarted several times before being capable to reach the end of the computation correctly.

The de-facto standard for intra-process communication is the Message Passing Interface (MPI). This standard was formalized in 1994 and received updates since then. As of now, the fourth main version of the standard is under development, and there are

various developed implementations such as MPICH, OpenMPI, and Intel MPI. Many HPC applications leverage MPI since it is the main way to exploit resources scattered through multiple nodes connected in a network. The success of MPI is partially due to its simplicity, which introduces a low communication cost: a desirable feature for HPC applications. While the simplicity made it successful, it is now becoming a constriction: the first version of MPI lacked many important features that were introduced in the next versions. Among those introduced features it is possible to find file operations, remote memory access, dynamic process management, and many more. The next version of MPI will introduce a system for better fault management.

Up to the latest version of MPI, the occurrence of an error causes problems in an MPI program since the status of the execution is undefined. This choice has been made to allow a simpler implementation (which can assume the absence of faults and give no grants in their presence), but it is now beginning to limit the applicability of MPI due to the increasing size of HPC systems. While the new versions of MPI will integrate the missing feature, all the applications that are leveraging the current standard will not see the changes and must receive some adaptation.

The development of fault management features in MPI originated a few years ago but is culminating recently with the creation of the User Level Fault Mitigation (ULFM) [10] library. It is one of the most important new introductions in the field and will be probably integrated into the MPI standard in the future versions. It defines new methods that can be used to recover from faults and lead back the computation to a consistent state. While introducing just a handful of new functions, it requires proper knowledge of the problem to be able to mitigate faults effectively and does not specify a way to recover the execution. This led to the development of all-in-one frameworks that combine ULFM with a method to restore the execution, to simplify the introduction of fault tolerance within an application (like Fenix [16], CPPC [25], LFLR [31]).

While these frameworks enhanced the reliability of an MPI application, their usage is not transparent and the application code has to be adapted accordingly. This solution is acceptable when designing a new application, but becomes problematic when targeting an already developed one.

This aspect is limiting the impact of those frameworks and led us towards the development of a solution that does not need changes in the application code. In this work, we limit our attention to embarrassingly parallel MPI applications, a very common and scalable type of parallel program that reduces to the minimum the interactions between the processes, and they are envisioned to be among the first ones capable to fully exploit the performance of future systems.

In this thesis we present Legio, a framework that introduces fault resiliency in embarrassingly parallel MPI applications. It shares many aspects with the previous frameworks ([16, 17, 25, 29, 31]), such as the usage of ULFM, but focuses more on the transparency of the integration. Since embarrassingly parallel applications can continue their execution even if some processes do not provide any result, we opted for fault resiliency. Upon noticing an error, the failed processes are discarded and the execution continues only with the non-failed ones. This approach is also faster compared to the standard C/R proposed in the other frameworks, but impacts the correctness of the application result: an acceptable trade-off in applications producing an approximate result, like for example Monte Carlo solvers [27], or high-throughput in-silico virtual

screening applications [1].

Legio supports most used MPI calls in embarrassingly parallel applications and many other advanced features. We also provide an alternative solution, capable of constructing a networking layer transparent to the application to reduce the impact of a fault to a few processes. We evaluate Legio on the Marconi100 cluster at CINECA [2] to measure the introduced overhead. Those analyses demonstrated that the proposed framework introduces fault resiliency with only a very limited impact on the performance of the application.

To summarize, the contributions of this thesis are the following:

- We propose the Legio framework able to transparently introduce fault resiliency in embarrassingly parallel applications;
- We implemented an alternative organization of MPI communicators to improve scalability;
- We experimentally evaluate the overheads and performance impact of the proposed solutions considering both the single MPI calls and full applications;

This thesis is structured as follows: Chapter 2 will be devoted to the analysis of background knowledge needed for a complete understanding of the problem and its solution; it will also contain a classification and an analysis of the state of the art of fault handling frameworks. Chapter 3 will cover the design and implementation of the Legio framework, and Chapter 4 will do the same on its hierarchical alternative, explaining its algorithms and its theoretical effectiveness. Chapter 5 goes through the experimental evaluation of our work by showing the overhead at the MPI call and application-level. Chapter 6 will show the possible future evolution of this effort. Finally, Chapter 7 will wrap-up the thesis.

CHAPTER 2

Background

THIS chapter is devoted to the analysis of all the preliminary concepts that serve as base for the rest of the thesis. Section 2.1 will present the Message Passing Interface (MPI) basics, Section 2.2 will cover some dependability systems fundamentals, Section 2.3 will focus on the new features introduced with ULFM. Lastly, Section 2.4 will consider all the efforts present in the field, comparing their results with the approach of this thesis.

2.1 MPI Background

It is very important to understand a few core concepts about MPI before proceeding since the whole thesis uses them widely. The standard supports both point-to-point and collective communications and defines the syntax and semantics of a core of library routines useful for writing portable message-passing programs. It is designed over a few concepts that can abstract the various concepts involved in communication. The first and most important is the communicator, which represents the connections between a group of processes in the session. Each process within a communicator will receive a rank, a unique identifier within it. Ranks start always at 0 and will go on until reaching the number of processes within the communicator (size) minus one. Upon initialization, MPI creates two communicators per process: `MPI_COMM_WORLD`, which will contain all the processes, and `MPI_COMM_SELF`, containing only a single process. All the details about the standard can be found on related documents [3, 11].

This section is structured as follows: subsection 2.1.1 will cover the calls that can be used to realize point-to-point communication and general setup, subsection 2.1.2 will deal with collective communication and synchronization, subsection 2.1.3 will go through group management operations, subsection 2.1.4 will analyse communicators

management functions, and the following two subsections will cover file operations and remote memory access respectively. The last subsection will analyse the error handling capabilities of the MPI standard.

2.1.1 General setup and point-to-point communication

To properly use MPI there is a function call that must be issued before any other: that is `MPI_Init` (or its multithreading version `MPI_Init_thread`). It takes as argument the ones passed to the main function and will setup the environment. The symmetric of `MPI_Init` is `MPI_Finalize`, which terminates the execution environment.

Some of the most used calls after the `MPI_Init` are `MPI_Comm_rank` and `MPI_Comm_size`: these are functions that return the rank and the dimension of the communicator passed as parameter. They are usually used on `MPI_COMM_WORLD` to define the roles of each process involved in the communication.

Point-to-point communication is mainly realized using a pair of functions called `MPI_Send` and `MPI_Recv`. The first takes as argument a pointer to the data to be sent, its count (number of elements to be sent), its type, the rank of the receiving process, a tag, and the communicator that will host the communication. The `MPI_Recv` takes similar parameters but with opposite meaning: the pointer will refer to the buffer that will get the data, the rank will be the one of the sending process (or `MPI_ANY_SOURCE`) and there is also an additional parameter that will store the status of the operation.

The point-to-point operation will match only if the communicator is the same, the tags correspond (or the receiver specifies `MPI_ANY_TAG`) and the ranks of the destination/source are correct. These are synchronous operations, and will not return until the communication is completed (or an error occurred). There are other variants of point-to-point calls that further specify how the operation is executed and also that perform the calls asynchronously but they were not used in the effort. Additional information about point-to-point operations can be found in chapter 3 of the MPI standard [3].

2.1.2 Collective communication

Collective operations involve all the processes within a communicator. Various types of collective perform different data movements. All these calls will need as a parameter at least the communicator that will be used and must be issued by all the processes within it to complete. The most simple collective is the `MPI_Barrier`, which just synchronizes all the processes in the communicator provided. Another important collective call is the `MPI_Bcast`, in which a single process (its rank has to be provided as the parameter root) sends the same data to all the others. All the processes within the same communicator have to call it with the same root. It has also similar parameters to the `MPI_Send` operation for what concerns the pointer to the data to be sent, its count, and its type.

Another important function is the `MPI_Reduce` function call, which can combine data from different processes into a single result. Aside from the pointer to the data to be combined, its number and its type, the function requires a root (rank of the process that will hold the result), a pointer to the buffer that will receive the result, and the operation that will be performed to combine the data. This operation can be specified by the

user or can be one of the predefined ones (like `MPI_SUM`, `MPI_MAX`, `MPI_PROD`, and many others).

Similar to the above function is the `MPI_AllReduce`, which takes the same parameters except for the root, which is missing. The function operates like the `MPI_Reduce`, but its result is given to all the processes within the communicator.

There is another function that combines values and it is called `MPI_Scan`: it combines the values of all the processes whose rank is minor or equal to the one of the process, so it behaves like an `MPI_Reduce` but provides also all the partial results to all the processes. Its parameters are the same as an `MPI_Allreduce`.

The last two functions that will be analysed are `MPI_Scatter` and `MPI_Gather`, which have a specular functionality: the first is used to split a chunk of data across all the processes within a communicator, the second will put together pieces from all the processes in a single block. Both operations will use the ranks of the processes to determine the order of the parts: for example, in the `MPI_Scatter` the process with rank 0 will receive the first part, the one with rank 1 will receive the second, and so on. They will take similar parameters: a pointer to the data to be sent, together to its number and type; the root (the rank of the process that distributes or collects the data), and a pointer to the data to be received by the root with its number and type. Figure 2.1 shows the functioning of these operations and can be used to better understand what they achieve. Additional information about collective communication can be found in chapter 5 of the MPI standard [11].

2.1.3 Groups management

`MPI_Group` is a structure of MPI that represents a collection of processes. It is very similar to a communicator but cannot be used to perform message passing operations. Groups are important since it is possible to manipulate them more freely than communicators and all the operations that involve them are not collective. Usually, groups are created from communicators with the function `MPI_Comm_group`, which will return the group of processes within the communicator provided as parameter. It is possible to obtain the rank of the process within the group and its size with functions similar to the ones used for communicators: `MPI_Group_rank` and `MPI_Group_size` respectively.

Operations used to manipulate groups are inspired by the algebra of sets: there are functions to create groups from the intersection or union of other groups. An operation used in the effort is `MPI_Group_difference` which creates a group of all the processes present in the group passed as the first parameter but not in the one passed as the second parameter.

Another important function for the effort is `MPI_Group_translate_ranks`. It takes as parameter two groups, two arrays of integers and an integer n representing the size of the two arrays. The function translates the ranks contained in the first array and referring to the first group into ranks referring to the second group and will store them in the second array. This function is usually used putting n to 1, so it just translates the rank of a single process from a group to another. If the process is not present in the second group, the rank produced will be `MPI_UNDEFINED`, a special macro defined in the standard.

When groups are not needed anymore, it is possible to delete them with the

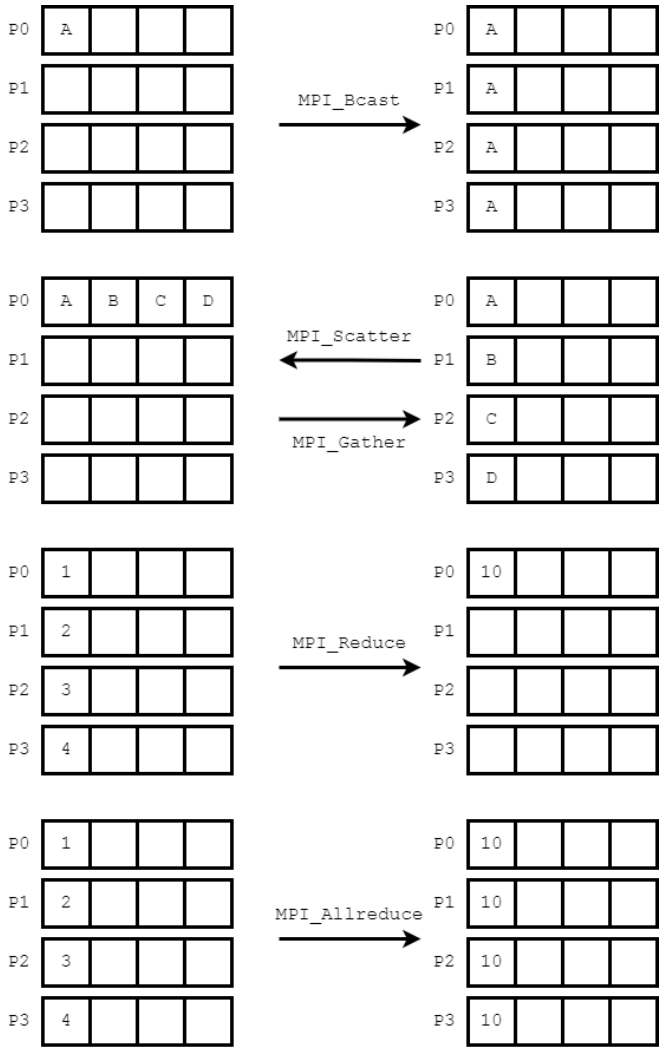

Figure 2.1: *Patterns of the MPI collective calls.*

`MPI_Group_free` call. Additional information about groups and their management can be found in sections 6.2.1 and 6.3 of the MPI standard [3].

2.1.4 Communicator management

MPI provides functions to create new communicators from existing ones and groups. All the functions that will be discussed in this section are collective. Among the most used functions, there is `MPI_Comm_dup`, which creates a duplicate of an existing communicator. `MPI_Comm_create` will instead create a communicator based on a communicator and an `MPI_Group` passed as parameters. The `MPI_Group` parameter must be a subset of the group of the communicator. The newly created communicator will have inside only the processes part of the group.

Another important function for communicator management is `MPI_Comm_split`: it creates a group of disjoint communicators and it is useful to reorder ranks. It takes as a parameter a communicator to split, an integer called colour, and another integer called key. The function will create a set of communicators in which all the participating

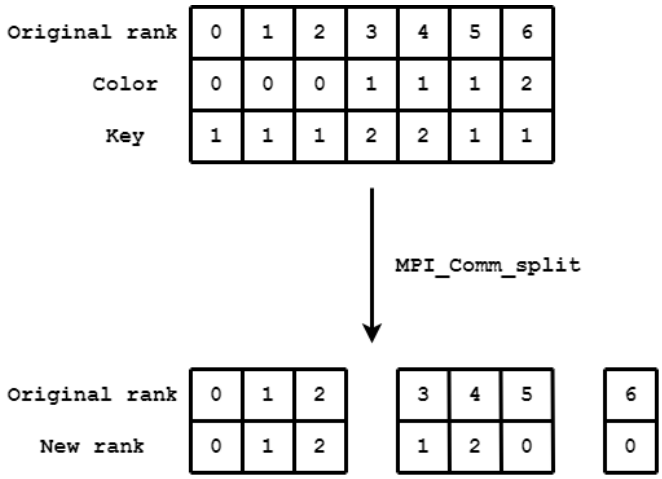

Figure 2.2: *Behaviour of the* `MPI_Comm_split` *call.*

processes used the same color parameter. Among the created communicators, ranks are ordered following the values of the key parameter, with ties broken according to the rank in the starting communicator.

`MPI_Comm_split` can be used to obtain a communicator with the same processes but with different ranks by calling it with a unique color and keys accordingly to the desired order. This technique will be used extensively in the thesis since it is the main way to reorder ranks in communicators. To create communicators from bigger ones is not the only way to proceed: the standard provides functions to create inter-communicators, born from the connection of disjoint communicators. When discussing inter-communicators it is best to refer to the standard ones as intra-communicators to avoid confusion, since they both use `MPI_Comm` objects.

Being the connection of two disjoint groups of processes, inter-communicators have a different structure compared to intra-communicators. During their use, one of the groups plays the role of the local one while the other will be the remote one depending on who initiated the communication. Inter-communicators can be used for point-to-point operations but a process can communicate only with processes of the group it is not part of.

Inter-communicators are obtained using intra-communicator creating routines on inter-communicators (like `MPI_Comm_split` or `MPI_Comm_dup`) or with the function `MPI_Intercomm_create`. This last function allows the user to connect two split intra-communicators in a single inter-communicator. Each process that wants to be part of the inter-communicator must provide a communicator (will be used as a local group) and a rank that will work as leader of its local group. Leaders have also to specify a communicator used to coordinate with the other group leader and the rank of the other leader within it.

It is possible to convert an inter-communicator into an intra-communicator with the function `MPI_Intercomm_merge`, which is collective across all the processes of the inter-communicator. Other than the inter-communicator, it requires as a parameter an integer that must be equal for all the processes within the same local group and is used to determine the ranks of the new communicator. In particular, the group that sets the variable to non-zero will have higher ranks and the rank order within a group will be

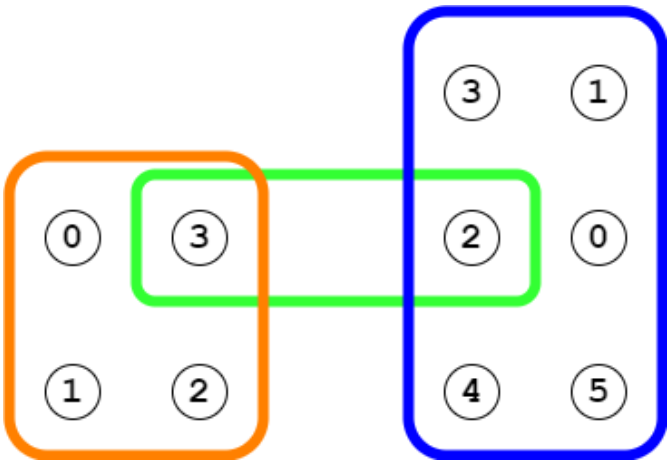

Figure 2.3: *Inter-communicators behaviour. The orange and blue rectangles represent the local groups and the green one connects the two leaders.*

kept in the new communicator.

Similarly with groups, it is possible to delete communicators that are not needed with the `MPI_Comm_free` call. Additional information about communicators and their management can be found in sections 6.2.3, 6.2.4, 6.4, and 6.6 of the MPI standard [3].

2.1.5 File operations

During the years MPI evolved and, with its second version, it started supporting reads and writes to file. It introduces its file handler, called `MPI_File`, which is used by its calls. To create a file handler the function `MPI_File_open` must be used: it takes as a parameter a communicator, the name of the file, and an integer representing the access mode. This function is a collective operation and so requires the collaboration of all the processes within the communicator.

After opening a file, it is possible to read and write it in different ways. The most simple way to operate on it is by direct access: the functions `MPI_File_read_at` and `MPI_File_write_at` provide this functionality. They require the user to specify an offset from the beginning of the file and will operate only on the data that is present after the offset. Other than the file handler, they require a pointer to the buffer that will be filled or written respectively, and also its type and quantity. There exist also collective variants of these functions, called `MPI_File_read_at_all` and `MPI_File_write_at_all` respectively. They perform the same operation but all the processes do it at the same time.

Another way to access files is through individual file pointers: each file handler provides one and can be considered like a cursor in the file. It is possible to move the individual file pointer with the function `MPI_File_seek`, which needs, besides the file handler, an offset and a whence that specifies the starting point of the offset. It is possible to retrieve the position of the individual file pointer with the `MPI_File_get_poistion` function, that, given a file handler, will show the offset from the beginning of the file.

Individual file pointers can be used for reading and writing with the functions

MPI_File_read and MPI_File_write respectively. Those functions are similar to the ones specified for direct access but they do not need any offset since they start operating from the individual file pointer. After the operation, the individual file pointer will be moved forward by the quantity of data read/written. Like for direct access, there are the collective versions of those calls, called MPI_File_read_all and MPI_File_write_all respectively.

A third way to access files is through the shared file pointer, which is similar to the individual one but is the same across all the processes within the communicator. Analogously, it is possible to move the shared file pointer with the function MPI_File_seek_shared, which has the same parameters as MPI_File_seek but is also collective. The function MPI_File_get_position_shared can be used to obtain the offset of the shared file pointer from the beginning of the file and works similarly with its individual file pointer counterpart.

Shared file pointers are used by the MPI_File_read_shared and MPI_File_write_shared functions, that behave similarly to the individual file pointers versions. It shall be noted that more care should be used when dealing with shared file pointers since they are single across all the processes in the communicator and access should be managed properly not to fall into race conditions. A solution to this problem can be the use of their collective versions, MPI_File_read_ordered and MPI_File_write_ordered respectively, that operate following the rank order.

Another important function is MPI_File_sync, which takes only a file handler and flushes all the pending operations to disk. It is used to ensure consistency between the various operations.

The standard MPI allows also the creation of views, that are the set of data visible and accessible from an opened file. By default, the view is set such that all the bytes are accessible by all. It is possible to manipulate this with the function MPI_File_set_view, which takes various parameters and among those a file handler, an offset, two types called etype and filetype respectively. The offset represents the start of the view: all the data before it will not be seen by the process. Etype will become the unit of data accessing and positioning and filetype will define what portions can be accessed and what cannot. Views are very important since it is possible to assign them differently among processes and obtain a transparent scatter/gather approach on the information stored in the disk.

File handlers, like groups and communicators, can be deleted when no more needed with the function MPI_File_free. Additional information about file access can be found in chapter 13 of the MPI standard [3].

2.1.6 Remote Memory Access

The MPI standard has not supported remote memory operations until the third version when the concept of window was introduced. MPI_Win is a structure that corresponds to portions of memory shared across a communicator. The creation of an MPI_Win object can be done with different functions: the most used are MPI_Win_create and MPI_Win_allocate. The first one will make accessible a part of memory specified by the user, while the second will create the area and then share it. They both need parameters that specify the size of the memory region, its displacement unit, a

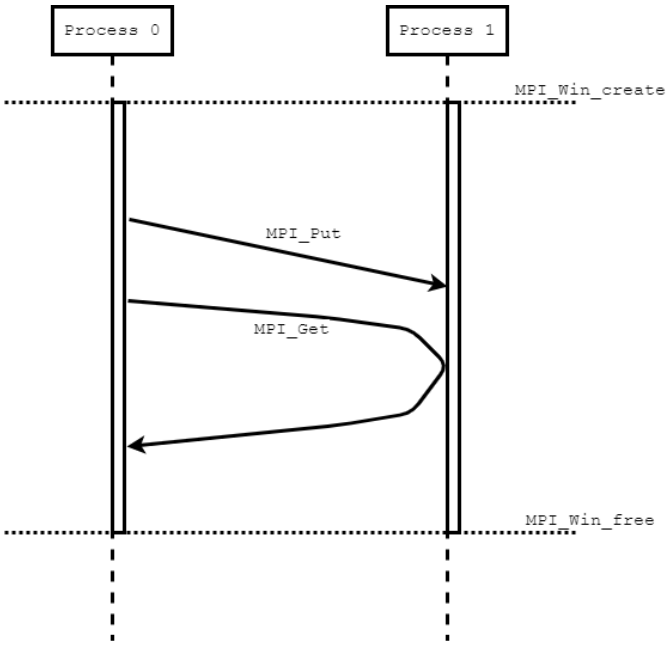

Figure 2.4: *The image shows the way `MPI_Get` and `MPI_Put` operate.*

communicator, and some other information and will produce an `MPI_Win` object. The only difference in the parameter list resides in a pointer: in the first function it is used as input and will specify the location of the memory area, in the second it is used as output and will point to the area allocated. It shall be noted that these functions are collective: an `MPI_Win` object will not just refer to the shared memory area created locally, but also to the ones present in the other processes created at the same time. Each process can specify different values for size and displacement to best fit their need.

Access to the memory areas referenced by an `MPI_Win` object can be performed with `MPI_Get` and `MPI_Put` operations: the first will retrieve data, the second will write in there. Both need the same parameters: the address of the buffer locally with its dimension and type, the displacement in the remote memory area with the number of elements to be accessed, the of the process whose area has to be accessed, and the `MPI_Win` object. A key feature of these calls is that, differently from point-to-point operations, they do not need a counterpart in the remote process: this is why remote memory operations are also called One-Sided Communication (OSC) within the standard.

These functions will not assure that the actual data movement will be performed within the duration of the call, so this could lead to coherency problems. To synchronize the operations, the function `MPI_Win_fence` is used: it divides the time into epochs such that all `MPI_Put` and `MPI_Get` operations can be sure to see the effect of all the calls happened up to the last completed epoch. `MPI_Win_fence` is a collective operation and will take as arguments a window object and an integer used for optimization.

Also, `MPI_Win` objects can be deleted when no more needed with the function `MPI_Win_free`. Additional information about remote memory access can be found in chapter 11 of the MPI standard [3].

2.1.7 Error handling

The standard MPI provides a structure that helps to deal with errors: it is called `MPI_Errhandler` and it contains a function that describes what to do. `MPI_Errhandler` objects must be linked to structures that may rise errors, which are communicators, windows, and files. Upon encountering an error, the execution will pass to the function within the error handler. The MPI standard specifies two predefined error handlers but the user can define new ones. The two predefined error handlers are:

- `MPI_ERRORS_ARE_FATAL`, which will cause the program to abort on all executing processes and it is the default for every structure;
- `MPI_ERRORS_RETURN`, which does nothing but the error is returned to the user.

The following statements, quoted directly from the standard, should be remarked since they are core for the problem faced by this thesis:

> *After an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or `MPI_ERRORS_RETURN`, does **not** necessarily allow the user to continue to use MPI after an error is detected. The purpose of these error handlers is to allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers) before a program exits. An MPI implementation is free to allow MPI to continue after an error but is not required to do so.*

This subsection will not focus on user-defined error handlers since they will not be used in the rest of the thesis, which exploits mainly `MPI_ERRORS_RETURN`. Additional information about error handlers can be found in section 8.3 of the MPI standard [3].

2.2 Fault Tolerance Background

The analysis of faults and the methods able to tackle them comes from the dependable systems field. The field plays a key role especially in others where the effect of faults cannot be mitigated or it may introduce high risks.

This section is structured as follows: subsection 2.2.1 will define how faults evolve in the environment, introducing important definitions; subsection 2.2.2 will define dependability and will go deeper considering all the different characteristics that may make a system dependable; subsection 2.2.3 will introduce a way to evaluate the dependability of a complex system. Subsection 2.2.4 will cover a dependability analysis applied to an HPC environment. Lastly, subsection 2.2.5 will analyse conceptual solutions used to introduce dependability in the HPC environment.

2.2.1 Faults, Errors and Failures

Usually, the terms fault, error, and failure are used as synonyms, but their meaning is slightly different. Distinguishing them is mandatory in a dependability analysis since their differences reflect the escalation phases that will occur in absence of countermeasures.

The term fault represents a non-correct functioning of a system (or one of its components) that turns the execution into an invalid state. Faults may arise from different reasons: from production defects to electromagnetic effects, from a human error to wear-out-related ones. Also, faults may arise at different levels, ranging from hardware to software. We will not discuss in detail all the possible faults that may happen in a system, because our analysis is focused on the software level and it would be difficult to distinguish them from there.

The term error indicates an observable effect of a fault in a system. The error notifies the presence of a fault, but may not be able to identify it. It shall be remarked that errors are not detected straight away, but it may require some time to identify them. Upon noticing an error, one of these two things will happen:

- The system can be repaired and continue the execution. This solution removes the fault and stops its escalation;
- The error may lead to failure, which is the impossibility of the system to complete a certain given task. This failure can be seen as a fault from components that encapsulate the failed one, so the initial fault is escalating.

The escalation may become dangerous when it reaches its top-most level since it may introduce hazards for the safety of the people using the system. Luckily for us, HPC systems tend not to have this problem, but the escalation will lead to an unsuccessful execution.

2.2.2 Dependability analysis

The escalation of faults shows the importance of having ways to reduce or even remove the impact of faults on a system. To measure how effective are those solutions it is better to introduce some useful metrics, the first of which is reliability.

Reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period. Reliability can also be defined as a function of time that represents the probability that the system will operate correctly in a specified operating environment up until the given time, or mathematically:

$$R(t) = P(not\, failed\, during\, [0, t]) \tag{2.1}$$

Reliability is also strictly related to the measure of the Mean Time To Failure (MTTF), which is often used as an alternative and can be defined as:

$$MTTF = \int_0^\infty R(t)\, dt \tag{2.2}$$

Reliability is particularly important in systems in which even momentary periods of incorrect behaviour are unacceptable: these include the ones with strict performance requirements, timing requirements and which are impossible to repair. The typical HPC environment tends to have those characteristics.

Another important metric in the dependability analysis is availability, that is the degree to which a system or component is operational and accessible when required for use. It can be seen as the probability that the system will be operational at a given time, or mathematically:

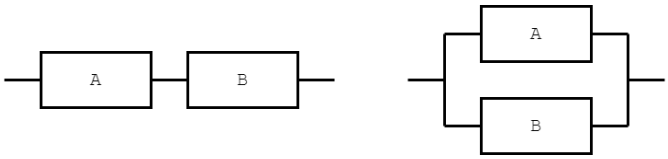

Figure 2.5: *On the left, two component in series. On the right, two components in parallel.*

$$A(t) = P(not\, failed\, at\, time\, t) \tag{2.3}$$

Availability represents the readiness of the service provided by the system and it is best suited for the ones that can be repaired and can accept a brief interruption of functionalities. Availability will not be analysed further since HPC systems tend not to be repairable and, even if they do, they are focused on performance.

Aside from these two main dimensions, different metrics evaluate other aspects of the system, like maintainability, safety, integrity, security, survivability, and testability. These will not be introduced since are far from the HPC case study, but more information about them can be found in [22].

An important metric for our analysis is performability, that is the probability that the system performance will be at least at some given level at a given time. The concept of performability introduces the one of Graceful Degradation, which is the ability of a system to automatically decrease its level of performance to compensate for hardware and software failures. Since the HPC environment is strictly connected to the concept of performance, those definitions will be useful in our next analyses.

2.2.3 Reliability Block Diagrams

While dealing with reliability, it is often useful to have ways to combine components' reliability to evaluate the metric of the whole system. There are various ways to do this but we will focus on Reliability Block Diagrams (RBD).

RBD is an inductive model in which a system is divided into blocks that represent the distinct components or subsystems it is made of. These blocks will be connected according to system-success pathways: a system will work only if there exists a path from start to finish that crosses only non-faulty components. After creating the diagram, it is possible to combine the metrics of the various components to obtain the ones of the whole system.

Metrics are combined analysing series and parallels of components: the first is a group of subsystems that will work only if all the parts are healthy, while the second is a group that will stop working only if all the parts are faulty. In RBD, series are represented as chains of components, while parallels are a list of components sharing the same entry-point and the same end-point.

From those definitions it is possible to compute the aggregate reliability:

$$
\begin{aligned}
R_{series}(t) &= P(noone\ failed\ in\ interval\ [0,t]) = \\
&= P(\bigcap_i (component\ i\ not\ failed\ in\ interval\ [0,t])) = \\
&= \prod_i P(component\ i\ not\ failed\ in\ interval\ [0,t]) = \prod_i R_i(t)
\end{aligned} \tag{2.4}
$$

$$
\begin{aligned}
R_{parallel}(t) &= 1 - P(all\ failed\ in\ interval\ [0,t]) = \\
&= 1 - P(\bigcap_i (component\ i\ failed\ in\ interval\ [0,t])) = \\
&= 1 - P(\bigcap_i \neg(component\ i\ not\ failed\ in\ interval\ [0,t])) = \\
&= 1 - \prod_i (1 - P(component\ i\ not\ failed\ in\ interval\ [0,t])) = 1 - \prod_i (1 - R_i(t))
\end{aligned} \tag{2.5}
$$

Given those formulas, it is possible to compute the reliability of most complex systems given the metrics of the parts. We will use these notions in the next subsection, focusing on a use case related to the rest of the thesis.

2.2.4 HPC use case analysis

To use RBD in the HPC use case it is better to first define which are the components that are involved in the analysis. For this analysis, we will assume that faults may happen in each core involved in the computation and we will consider only them as a possible source of faults. The system will consist of a distributed application that will run in parallel on a high number of cores (one process per core) for a given amount of time.

Let's assume that each core has an exponentially distributed reliability function, with MTTF equal to one century: this is a big over-approximation but is useful to further remark the analysis.

Let's assume that the application uses MPI to realize communication between the computing processes. As stated in Section 2.1.7, the state of an MPI application after an error is undefined, so it is not possible to continue the execution or to repair the system. A single fault will escalate making the execution on the whole system stop, so all the cores must show non-faulty behaviour to obtain the desired result. This last statement is helpful to realise the RBD analysis since it suggests that all the cores can be viewed as in series for what concerns reliability study. Given that, it is possible to compute the reliability of a system of n cores as:

$$
R\_sys(t) = [R(t)]^n = e^{(-n/(MTTFt))} = e^{\frac{-1}{(\frac{MTTF}{n})t}} \tag{2.6}
$$

The most important remark from the above formula is the evolution of the MTTF, which is divided by the number of cores in the system. And if we consider one of the biggest clusters in the world, the IBM developed Summit with 2,397,824 cores [5], the MTTF of the system comes down to about 21 minutes.

Given the MTTF of the system, it is possible to evaluate how much time is needed on average to complete an execution that would require a certain time in absence of errors. We can view the reliability of the system also as the probability that the execution requiring t time will finish before the insurgence of a fault. Upon fault, the execution must start from the beginning, so we can view the execution as a series of tries, each with probability R(t) to succeed. The average amount of tries to get a positive result is 1/R(t), which shows an exponential growth with time.

As it is, the situation is not acceptable: the time needed to complete execution for an HPC application is very long, in the order of hours if not days. The waste of time due to faults is becoming the bottleneck of these applications and it is mandatory to evaluate some strategies to solve the problem.

2.2.5 Dependability solutions for HPC

In the previous subsection, we discussed the reliability absence problem in the HPC environment. The causes of the problem can be found in the impossibility to repair and the need to restart computation from the beginning every time. The easiest way to reduce the impact is to approach it from the second cause, introducing some ways of storing redundant information that can be used upon restart. Checkpoint and Restart (C/R) solutions follow exactly this path [8, 16, 17, 20, 25, 26, 29, 29, 31].

The C/R approach consists of the periodical saves by the application of its status. Upon fault, the execution is stopped and restarted but it will happen from the point saved rather than the beginning of the application. This approach reduces the losses in case of faults but introduces also a loss of performance (called overhead from now on) due to the saving operations. The overhead is important since is a net loss: the status saved is not useful if there is no fault, so the application is losing efficiency. The overhead containment is especially critical in the HPC environment since performance should never be sacrificed in vain.

There are various ways to realise C/R, the differences come from the coordination patterns used, the level of abstraction, the position of the stored data (disk or memory), and so on. The main distinction comes from the level of abstraction: some solutions are defined as system-level since they work on a lower level than the executing application [8,20], others are user-level since they require the programmer to directly specify some information to be used during the saving process [16,17,25,26,29,29,31]. Application-level C/R is usually preferred in the HPC environment due to its higher configurability, which usually leads to better performance.

The type of coordination is a core problem in C/R. Coordination requires communication, which leads to a bigger overhead. On the other hand, uncoordinated C/R may incur in the Domino effect, which greatly increases the distance from the last valid state [28].

Aside from C/R, other solutions try to reduce the amount of work lost upon fault: Algorithm-Based Fault Tolerance (ABFT) [14] bases on the possibility to re-compute lost data exploiting the characteristics of the application. This approach has a lower overhead impact compared to standard C/R, but lacks generality and cannot be viewed as a unique solution.

Another approach seldom used is the log-based one [24]: it consists of saving all the messages that processes exchange to be able to repeat them in case of failure. The

main issue of this solution is that the overhead is very big and it is better used in other fields (like database management).

2.3 ULFM Background

Many efforts in the past tried to create a fault-tolerant MPI implementation in various ways: some focused on C/R within MPI, others on process replication, others on migration. These efforts received some citations but did not reach out enough to justify long-term support: most of them have not received an update for the last ten years. This is contradictory if compared to the attention rise that happened on the field in the last years, but can be explained by the presence of the User-Level Fault Mitigation (ULFM) effort.

ULFM was born as an evolution of FT-MPI, an effort that introduced the ability to repair communicators in MPI. Among the weak points of FT-MPI, the absence of support for communicators different from `MPI_COMM_WORLD` stopped its widespread adoption.

The main goal of ULFM is to define a minimal set of functions that can introduce fault tolerance in an MPI application. It should pose no constraints on the programmer but should only enable new functionalities. It reused the work of FT-MPI concerning communicator reparation, but does not specify a way to recover the execution: it just provides methods for handling the fault and repair the involved structures. This flexibility and transparency made ULFM very appealing among the programmers dealing with fault tolerance problems, and it quickly received a lot of attention. The attention led to more development and, as of now, ULFM is in synchrony with the evolution of the MPI implementation it is based on, OpenMPI.

Currently, ULFM is one of the main focuses of the MPI Forum's Fault Tolerance Working Group, a development team that aims at introducing fault tolerance in the MPI standard. Many applications started using it and obtained fault tolerance with negligible overhead.

2.3.1 ULFM standard

ULFM bases on the principle that no MPI call can block indefinitely after a fault, but must either succeed or raise an MPI error. The ULFM standard introduces three supplementary errors which are missing in the MPI standard that can represent the occurrence of a fault in the communication. Those errors are:

- `MPIX_ERR_PROC_FAILED`, raised when a process fault prevents the completion of an MPI operation;
- `MPIX_ERR_PROC_FAILED_PENDING`, raised when a process performs a wildcard receive but another process in the network has failed;
- `MPIX_ERR_REVOKED`, raised when performing operations on a revoked communicator.

These three errors help MPI to understand the status of the execution, enabling the user to use the newly added functions properly. The ULFM standard introduces 5 new

functions that cover not only the error propagation and notification parts but also the communicator repair phase. Those functions are:

- `MPIX_Comm_revoke`, which takes as a parameter a communicator and interrupts any communication pending on it at all ranks. It is the function that realizes error propagation. The communicator will be revoked and will return `MPIX_ERR_REVOKED` each time a process tries to use it. The function `MPIX_Comm_is_revoked` is also present in the latest version and it simply checks if the communicator passed as a parameter has been revoked;

- `MPIX_Comm_shrink`, which takes a communicator as a parameter and will create a new one with only the non-failed processes. It is the most important function for what concerns communication repair;

- `MPIX_Comm_agree`, which performs a consensus on a parameter over a communicator. The function will perform a bitwise or reduction of all the values, and the result will be available by all the processes;

- `MPIX_Comm_failure_get_acked`, which obtains the group of currently known failed processes;

- `MPIX_Comm_failure_ack`, which notifies the presence of faults in the communicator to let processes which issued a wildcard receive raise `MPI_ERR_PROC_FAILED_PENDING`.

To detect an error, ULFM needs a timer that will measure the maximum amount of time an MPI call can take. If the call is still waiting after the timer expires, the `MPIX_ERR_PROC_FAILED` error is raised and the repair phase can begin. This approach poses some problems due to the insurgence of spurious errors, especially in the case of programs where communication plays only a side role. ULFM provides also other ways to detect failures, which consists of an internal thread that will check the liveliness of all the processes. This solution introduces a slightly bigger overhead, but it is usually an acceptable trade-off.

Many other efforts inspired or even used ULFM to provide libraries and frameworks able to introduce fault tolerance in generic MPI applications. These efforts try to integrate the recovery policy with ULFM, creating an all-in-one solution for the introduction of fault tolerance. We will explore these solutions together with other approaches in the next section.

2.4 State of The Art

In this section, we will analyse the solutions adopted up to now, starting with a brief classification of the most important efforts. This classification is based on a few metrics that help to define similar characteristics. It also helps us defining areas not completely covered by the efforts, and that may require some further work. This classification will be covered in the first subsection, while the next one will focus more on the efforts that propose a similar solution to the one in this thesis.

		Recovery policy		
		Local recovery	Global recovery	None
Integration approach	Extension	[23] *[24]* [17] *[31]*	*[25] [16]* [30]	[21]
	Change		[6]	[15]
	None	*[12]*		[19]

Table 2.1: *Classification of the main related works. In italics efforts using ULFM.*

2.4.1 Classification

The first metrics to be analysed is the way the framework is introduced within the MPI standard: some approaches prefer not to modify it, creating an extension, others opt for changing it. Other efforts are more towards a single application and prefer not to integrate their approach with the MPI standard but directly change the code of their application.

Another metric defines what happens in case of failure: if all the processes have to restart it is possible to talk about global recovery, otherwise the analysed framework introduces local recovery. Other efforts tend to avoid recovering the execution, either by not restoring the failed process or solving the failure without rolling back.

Following these two metrics, it is possible to summarize all the efforts in a single table like the one that follows:

The distribution of the efforts in the table is not uniform: this is due to the different difficulty of the various approaches. In particular, extending the MPI standard is easier than modifying it since it must not face compatibility issues that may arise when introducing changing the standard. Additionally, it shall be remarked that local recovery solutions are developed usually from global ones since some parts can be reused.

2.4.2 Most relevant efforts

Many efforts have been built on top of ULFM functionalities by adding different recovery strategies. In particular, the integration of a C/R framework with ULFM provides an all-in-one framework to manage the insurgence of faults in a generic MPI application [16, 17, 25, 29, 31]. These solutions opted for the recovery of a consistent state: by loading a previous checkpoint, the execution restarts from a valid point. They usually provide a simple interface to the user but require changes in the application code. While obtaining a similar result to our proposed solutions, these frameworks do not pursue transparency and, rather than opting for fault resiliency, they recreate the failed processes. These characteristics provide the possibility to create a more flexible solution, working with any MPI application, at the cost of application intrusiveness and performance penalty in case of a fault.

A completely different perspective is the one presented when applying algorithm-based fault tolerance [14], which exploits the possibility to re-compute the data of a failed process using the information of the others. This solution is very application-specific since it leverages data redundancy to implement a resilient method with reduced overhead. Examples are shown in the context of matrix-multiplication and LU factorization kernels, but cannot be taken into consideration for a generic MPI program. In particular, ABFT should not be exploitable in embarrassingly parallel applications, like the one we are targeting with Legio, due to the high data independence across the

processes.

A method tackling transient fault has been presented in SLIM (session layer intermediary) [21]. The solution reduces the impact of transient faults by repeating the operations. Despite SLIM works for any MPI application, it cannot be considered a valid solution in case of permanent faults.

An effort that shares many concepts with the approach we are proposing has been presented in [27]. It uses the functionalities introduced by ULFM to manage the presence of faults in a Monte Carlo application, a typical embarrassingly parallel MPI application. The authors implemented the resiliency by removing the faulty processes from the execution and continuing only with the non-failed ones. The concept behind this solution is similar to the one proposed in this paper. However, it has been achieved by directly modifying the application code since the focus of the authors was on a specific application. With Legio, we are proposing to generalize this approach by implementing a transparent framework capable to tackle all the embarrassingly parallel applications.

A lot of efforts tried to solve the fault-tolerance problem with purely C/R. We will analyse those efforts in the following subsection.

2.4.3 Pure C/R solutions

During the years a lot of efforts tried to solve the problem by using pure C/R solutions: upon fault, the execution is stopped and will restart from the last saved point. It is important to analyze these solutions since they can be the base for a more complex approach, similar to what has been done in Fenix [16] and CPPC [25]. Before analyzing the efforts, it is better to define some characteristics we can use to better describe the solution produced.

The first distinction we are going to analyze is between application-level and system-level C/R: the first provides a service to the application, which can interact with it and choose what to save and when; the second acts from a lower level and the application does not recognise its presence. The distinction between application-level and system-level is the basis of the trade-off these solutions have to face: approaches that are more towards the application-level can produce smaller checkpoints at the cost of transparency; system-level solutions tend to be less efficient and may face consistency problems since they lack the information the other level can leverage.

Other classification criteria involve the eventual presence of coordination in the C/R routine: while the communication needed for coordination constitutes a cost in terms of performance, its absence may it make more difficult to find a correct spot to restart from. This difficulty comes from the presence of messages that may disrupt the status upon restart, leading to what is called the domino effect. The insurgence of the domino effect led to the preference towards coordinated C/R, a trend that can easily be seen in the literature by the disparity of the efforts number in the two fields.

Given these classification criteria, it is possible to effectively analyze the efforts present in literature. The first one that we are going to consider is Simple Checkpoint/Restart (SCR) [26], developed by a working group at the Lawrence Livermore National Library. The produced solution implements coordinated application-level C/R by providing a simple interface for the application. The interface defines functions that the application can call to start, coordinate, and write checkpoints. The application can choose what and where to save, giving maximum freedom to the user.

CRAFT [29] is another effort that develops the trend of application-level C/R. The usage is very similar to SRC, but it includes support for dynamic process recovery capabilities using ULFM. The two approaches provided can be used separately or combined, but both need changes in the application to be fully operative.

On the side of system-level C/R, Berkeley Lab Checkpoint/Restart (BLCR) [20] is one of the most remarkable efforts. It is completely transparent to the application since it works at the kernel level and its support requires only a recompilation. It is supported by many job schedulers, like SLURM, which interact directly with it and can fully benefit from the features it introduces. BLCR has been widely used in HPC, primarily due to the transparency of its introduction.

Another remarkable effort on the side of system-level C/R is the Distributed Multi-Threaded CheckPointing facility (DMTCP) [8], which achieves a similar result without needing access to the kernel level. The framework is designed to support feature integration with the use of plug-ins: among those, we can find one that allows the application to control the frequency of checkpoints. Its expandability and adaptability make it one of the most developed solutions up to now and its latest evolution, MANA [18], introduces further functionalities for MPI applications.

CHAPTER 3

The Legio framework design and architecture

THIS chapter will focus on Legio, the proposed solution that introduces fault resiliency in an embarrassingly parallel MPI application. The key features of the framework are the transparency towards the application and the focus towards fault resiliency. These concepts will make the framework more usable and less impactful than the others present in literature [16, 17, 25, 29, 31]. We will follow this structure: Section 3.1 will deeply analyse the requirements of the project, pointing out needed functionalities and desirable properties. The analysis will proceed in Section 3.2 with some preliminary analyses that will be the basis for the rest of the project. From Section 3.3 we will analyse all the implementing details of the project, pointing out the main problems and alternative solutions.

3.1 Requirements

It is possible to briefly summarize the key requirements of the Legio framework in the following list:

- The library should introduce fault resiliency in an MPI application;
- The library must need no code changes in the application to work properly (some changes may be optional to better understand the status of the communication or to allow additional configurability/tuning);
- The library must introduce a negligible overhead into the application;
- The library must keep the scalability properties of the application;
- The library should expose no against-standard behaviour in absence of faults;

- The library should be based on well-supported libraries (like ULFM);
- The library should support a reasonable set of MPI functions.

The name Legio comes from a Latin word that represents a military unit of the Roman army. The name was inspired by the fact that soldiers will keep fighting even after some of their fellows perish: analogously, the library aims to make MPI processes continue their execution, despite the failures of some of them.

3.2 Preliminary analyses

In this section, we will discuss some issues of the ULFM implementation of the MPI standard in presence of faults [17, 32]. The considerations taken in this section will be the basis for the design process done in the next chapters.

Before proceeding with the analysis, we want to provide some definitions of key terms for the remaining part of the paper:

- A *process notices a fault* when it receives the error code `MPIX_ERR_PROC_FAILED` after an MPI call;
- A *faulty communicator* is a communicator in which at least a participating process is failed, but no process noticed it yet;
- A *failed communicator* is a communicator in which (at least) a participating process noticed the fault;
- A *revoked communicator* is a communicator in which a process called `MPIX_Comm_revoke`.

All the operations not working on a faulty communicator will not work on a failed one. All the operations working on a failed communicator will work on a faulty one. No operation will work on a revoked communicator, except for all the ULFM calls.

Using these definitions, we sum up our considerations on the MPI standard in points to better refer to them in the next sections.

P.1 Some MPI functions work in faulty and failed communicators. Some remarkable functions that expose this behaviour are `MPI_Comm_rank` and `MPI_Comm_size`, but also many operations that deal with `MPI_Groups`. These operations are labelled as local in the MPI standard and do not require communication to complete successfully.

P.2 Point-to-point communication works in a faulty communicator, as long as the processes involved in it are not failed. They will not work in a failed communicator.

P.3 Collective communications will not work in a failed communicator but may expose strange behaviour in a faulty communicator. This behaviour comes from the fact that not all the processes may notice the fault. The `MPI_Bcast` operation exposes this strange behaviour, while others like `MPI_Reduce`, `MPI_Barrier`, and `MPI_AllReduce` do not. This behaviour will be called the "Broadcast Notification Problem" (**BNP**) from now on.

P.4 File and remote memory access operations are not supported by ULFM and are likely to fail in a faulty environment (rather than raising an error, they throw a segmentation fault making the execution impossible to recover).

P.5 Communicator management functions like `MPI_Comm_dup` or `MPI_Comm_split` will not work in a faulty communicator. This includes also all the Inter-communicator related ones.

P.6 The `MPIX_Comm_revoke` function introduced by ULFM has no instant effect and cannot be used to fix the BNP. The effect will be seen by all the processes, but some of them may see it after the next MPI operation.

P.7 The `MPIX_Comm_shrink` function will keep the processes ranks in the same order as in the original communicator.

P.8 The `MPIX_Comm_agree` function will work on faulty and failed communicators and can be used to solve the BNP.

An additional explanation should be provided for the BNP. Collective operations can be realized following a virtual topology, causing a loss of symmetry between the processes. The broadcast operation, for example, follows a tree topology in the ULFM implementation: the root sends the information to another process, then the two forward to the other two, then the four to the other four, and so on. If the root is failed, all the processes will notice the fault, but this is not true for a process that does not forward: only the process that would have sent it the data recognises the error, creating the BNP.

The revoke operation is useful to propagate the fault to all the processes communicating directly with the caller but won't stop the communication entirely since it needs to be propagated. Eventually, all the processes will notice the error, but at the end of the operation causing the BNP only a subset of those will. Having some processes that stop before a collective and others after can create problems when resuming the execution: after the reparation of the communicators, some processes will have to repeat the failed collective, but cannot do so since other completed it and are proceeding with their execution.

The agree operation behaves differently and allows us to avoid the problem: it is a collective call, so it can be used to check the result of the problematic operations and perform the reparation even if it is completed without problems. The agree operation is a bit more costly than the revoke one and ULFM standard suggests to reduce its use and prefer the revoke, but it is the only way to solve the BNP effectively.

3.3 The barrier and Broadcast support

The proposed solution consists of the substitution of the MPI structures used (and created) by the application with others managed by Legio. This way, when a fault happens, it affects only the Legio structures, making the repair process easier and controllable by the framework. The MPI structures that are involved in the Legio repair process are communicators, windows, and files.

To achieve our purpose, we designed a library that behaves like an intermediary between the application and the MPI implementation. To achieve that position we exploit the profiling interface provided by the MPI standard (`PMPI`), which allows us to

redefine most of the MPI calls without affecting the application. In the MPI standard, PMPI is the profiling interface that allows intercepting every MPI call made in the parallel program. Originally thought for profiling, it can be used to inject code of different types around the target MPI call. In our work, we used PMPI to introduce fault resiliency using ad-hoc code and ULFM methods.

Implementation started considering a single operation and only the MPI_COMM_WORLD communicator. The first operation considered was the MPI_Barrier, due to its simplicity. The key idea was to control if the used communicator was MPI_COMM_WORLD and if so, substitute it with a duplicate created within the MPI_Init. If the operation on the substitute raises an error, it is possible to perform an MPIX_Comm_shrink on the duplicate to obtain a new one with only the survivor processes. After shrinking, the operation is repeated: the execution will exit the MPI_Barrier only upon complete and correct execution.

The following code block contains the implementation described above. It received changes with the introduction of other features, but we will discuss them later.

```
int MPI_Barrier(MPI_Comm comm) {
  while(1) {
    int rc;
    if(comm == MPI_COMM_WORLD)
      rc = PMPI_Barrier(cur_comm);
    else
      rc = PMPI_Barrier(comm);
    //...
    if(rc == MPI_SUCCESS || comm != MPI_COMM_WORLD)
      return rc;
    else
      replace_comm(&cur_comm);
  }
}
```

The replace_comm function is used to call the MPIX_Comm_shrink and after that change the error handler of the newly created communicator to MPI_ERRORS_RETURN. The variable cur_comm is a global MPI_Comm that contains the current duplicate of MPI_COMM_WORLD. This first code shows some patterns that will be reused for the rest of the code. In this case, the substitution happens only if the communicator used for the operation is MPI_COMM_WORLD, if not the operation will complete without changes. The MPI_Barrier implementation is certainly useful to show the basics of how Legio works, but it is not enough representative of the other function complexity. From now on, we will call the communicator used for actual communication instead of MPI_COMM_WORLD as the substitute.

The MPI_Bcast requires four parameters more, so it is natural to expect it to be more complex. Analysing them more, it is possible to see that three of them do not require changes: the buffer address, its dimension and it is type shall not be touched by the library, since they will not cause any communication error. Legio will solve only errors raised due to faults and cannot deal with bugs, so we are expecting not to receive problems from those three parameters. The root parameter is a bit more tricky since it is a rank.

The problem with the ranks is that the substitution made may associate different ranks for each process. While this is not true in absence of faults (the duplication with MPI_Comm_dup preserves the ranks), it happens after any fault since the MPIX_Comm_shrink creates a smaller communicator. So, for example, if the node

Original rank	0	1	2	3	4	5	6
Rank in substitute	0	1	2	3	4	5	6
Rank in substitute after 3 failed	0	1	2	X	3	4	5
Rank in substitute after 6 failed	0	1	2	X	3	4	X
Rank in substitute after 0 failed	X	0	1	X	2	3	X

Figure 3.1: *The figure shows the evolution of ranks in the substitute communicator with the insurgence of faults. The X represents the* `MPI_UNDEFINED` *value.*

with rank 0 fails, all the nodes will have the rank in the substitute equal to the one in alias minus one because of property **P.7**. It is mandatory to perform some operation to translate the ranks from the alias communicator to the substitute. This functionality is performed by the `translate_ranks` operation.

This operation takes as a parameter the rank in alias and the substitute communicator and will perform the translation. The function is a wrapper of the `MPI_Group_translate_ranks` function which extracts the groups from the communicators and performs the translation. One of the problems of the translation is that it can produce `MPI_UNDEFINED` when a failed process is part of the alias communicator but is not in the substitute communicator. This occurrence has a different meaning in each function and will be considered singularly.

The `MPI_Bcast` support works as follows: firstly, the communicator is compared to `MPI_COMM_WORLD` to see if the function shall or should not be modified; if it is equal, then the root is translated and it is possible to perform the call with the substitute communicator and the translated rank. Special care must be given when the translation returns `MPI_UNDEFINED`: it happens when the process that would have to share the information with all the others is failed. During the development of Legio, we decided to leave some choices to the user, enabling some compile-time configuration that allows to stop the execution completely or ignore the operation. In this case, we are expecting that the application needs the information shared in the broadcast operation and will not be able to produce a useful result without it, so we opted for the termination of the program as a default option. The user may decide to switch to ignoring the operation and continuing the execution, but this may make the code use some uninitialized variables (the receive buffer), so additional care must be taken.

After the execution of the call with the substitute and the translated rank, the system checks the presence of errors. Due to property **P.3**, we have to consider the BNP eventuality: the `MPIX_Comm_shrink` is a collective operation and will be called only if an error is present, so it is necessary to provide some way to have a unique view on the status of the execution. Property **P.8** suggests the use of the function `MPIX_Comm_agree`, and that is exactly the solution we chose. After the agreement, it is possible to check the presence of errors and eventually replace the communicator like in the `MPI_Bcast`. The following code summarizes all the concepts described above:

```
1 int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
      {
2   while(1) {
```

```
    int rc;
    if(comm == MPI_COMM_WORLD) {
      int root_rank;
      translate_ranks(root, cur_comm, &root_rank);
      if(root_rank == MPI_UNDEFINED) {
        //handle missing root, user configured
      }
      rc = PMPI_Bcast(buffer, count, datatype, root_rank, cur_comm);
    }
    else
      rc = PMPI_Bcast(buffer, count, datatype, root, comm);
    //...
    if(comm == MPI_COMM_WORLD) {
      int flag = (MPI_SUCCESS==rc);
      MPIX_Comm_agree(cur_comm, &flag);
      if(!flag && rc == MPI_SUCCESS)
        rc = MPIX_ERR_PROC_FAILED;
      if(rc == MPI_SUCCESS)
        return rc;
      else
        replace_comm(&cur_comm);
    }
    else
      return rc;
  }
}
```

These were the first functions supported by Legio and show some other important remarks:

- Execution of an MPI function on `MPI_COMM_WORLD` is repeated until it is correct.
- `MPIX_Comm_revoke` is a powerful function to propagate the errors within a communicator eventually, but cannot be used reliably to propagate before the end of the call that generated the error. This is why it is not used and the function `MPIX_Comm_agree` is used instead.
- These two functions have a rather similar pattern that will be shared with many other functions: in particular, they consist of
  - an outer loop;
  - the control over the communicator to check if it is `MPI_COMM_WORLD`;
  - the eventual translation of the ranks specified in the function;
  - the execution of the operation;
  - the error checking part, with the eventual agreement phase and culminating with the call to `replace_comm` in case of faults.

  These points will help us define other functions faster since we will base on the presence/absence of these procedures.

3.4 Point-to-point operations and other collectives

As stated in properties **P.2** and **P.3**, the main difference for what regards error notification between point-to-point operations and collective ones is their behaviour in

a faulty communicator: the first will not notice the error if it is outside from the involved processes, while the second will notice it (after solving the Broadcast Notification Problem). Also, point-to-point operations do not involve all the processes within a communicator, so detecting an error cannot cause a `replace_comm` call since the shrink operation is collective. Because of this, point-to-point operation should not perform the error checking part and, as a consequence, do not need a loop to keep trying until successful execution. Both the `MPI_Send` and the `MPI_Recv` have to translate the ranks and may have to deal with the `MPI_UNDEFINED` value: in the sending operation, the default reaction is to ignore the call, while in the receive the execution is stopped. Analogously with the broadcast operation (and in all other operations from now on that have to deal with the `MPI_UNDEFINED` problem), the user can change those values in the pre-compilation phase.

There are a few other notes on the structure of point-to-point operations that should be considered. Send operations, while not having a loop that iterates until the execution encounters no problems, have a cycle that will repeat the execution for a limited number of tries. This solution tries to limit the impact of transient faults, which are the ones that rise but then vanish a few instants later. By repeating the execution a few times, it is possible to go further than the length of the fault and achieve communication despite the problems. Another remark involves the receive operation, which has been modified to check if the rank passed as the source parameter is `MPI_ANY_SOURCE`: in that case, the procedure for fault management must be slightly different, since it involves the function `MPIX_Comm_failure_ack`.

Aside from `MPI_Bcast` and `MPI_Barrier`, it is possible to easily add the support for many other MPI collective calls. The `MPI_Allreduce` operation, for example, is implemented very similarly to the `MPI_Barrier` since it requires no agreement and no rank manipulation. The `MPI_Reduce` function, on the other hand, behaves very similarly to the `MPI_Bcast`, needing both the agreement and the rank translation. In the `MPI_Reduce`, upon failed translation of the root rank, the default option is to ignore the operation and proceed with the execution. The `MPI_Scan` operation is supported similarly: it needs no rank translation but requires the agreement.

The case of the `MPI_Scan` operation is simpler than it could have been. The main difference between all of the operations above and the `MPI_Scan` is that in the latter the rank of all the processes heavily influences the output produced, while in the firsts it does not. The point is, if we run the same `MPI_Scan` operation over two communicators having the same processes in different rank order, we get results that are very different. On the other hand, if we shuffle the ranks in any of the operations above, the result will not change (as long as we translate eventual root ranks correctly). This would have raised a lot of problems if it was not for property **P.7**: by keeping the order, we are sure that each process will have as its predecessors only non-failed predecessors from the pre-shrink communicator. So the operation completes without further problems.

3.5 The scatter and gather support

The functions `MPI_Scatter` and `MPI_Gather` proved non-trivial to support since they use not only the rank order but also the rank number. The value of each process rank specifies the part of the buffer assigned to each process, in terms of offset from the

Original rank	0	1	2	3	4	5	6
Rank in substitute after 3 failed	0	1	2	X	3	4	5
Scatter result if no faults	A	B	C	D	E	F	G
Scatter result using substitute	A	B	C	X	D	E	F
Scatter result correct	A	B	C	X	E	F	G

Figure 3.2: *The figure shows that may rise with the function* `MPI_Scatter` *in presence of faults.*

beginning of the buffer. Working on a shrank communicator means having fewer ranks, so the last position in the buffer cannot be accessible. To keep output consistency, these functions have to be realized as combinations of others.

This is the first time we had to realize a function that is not based on its standard implementation but follows a new structure. In this case, we noted that point-to-point operations, while managing to put the data in the correct position, do not scale well with the communicator dimension since the root of the operation has to perform one send (or receive in case of `MPI_Gather`) for each process in the communicator. Those operations cannot be performed in parallel, so the serialization would worsen the performance too much. On the other hand, basing on one-sided communications would have solved the problem since every process would have to perform a single operation.

Converting the scatter operation in a set of one-sided communication operations has not been so difficult. It was possible to define the relationships between the various parameters between `MPI_Scatter`, `MPI_Win_create`, and `MPI_Get`: each process computes the offset in the buffer that should be distributed with the `MPI_Scatter` and obtains the data it needs with the `MPI_Get` operation. The buffer is visible to all the processes within the communicator thanks to the `MPI_Win_create` function, removing the need for other calls. The `MPI_Gather` is realized analogously, with the function `MPI_Put` instead of the `MPI_Get`.

In conclusion, the structure of the functions is similar to an `MPI_Bcast` since both the rank translation and the agreement at the end are needed, but the operations are different from the MPI standard ones. Both the `MPI_Gather` and the `MPI_Scatter` have to deal with the eventuality of the failed root and solve it by ignoring and aborting by default respectively.

3.6 One-Sided Communication

The functions `MPI_Scatter` and `MPI_Gather` required the support for one-sided communication. This functionality needs dedicated management of the `MPI_Win` object since the simple management of `MPI_Comm` is not enough for functions like `MPI_Get` and `MPI_Put`. It is mandatory to keep track of the processes that have access to the window since a fault among them may make the execution stop as stated in property **P.4**.

The solution we followed was the creation of a collection of objects that contains each window created using `MPI_COMM_WORLD`. This collection is stored within the `ComplexComm` class, that will be used to keep the substitute communicator and all

the windows created from MPI_COMM_WORLD. The first subsection will cover that class, while the second will focus on the support for the various functions.

3.6.1 The ComplexComm class

The code snippet that follows contains the structure of the first version of the ComplexComm class:

```
class ComplexComm {
  public:
    void replace_comm(MPI_Comm);
    MPI_Comm get_comm();

    void add_window(void*, MPI_Aint, int, MPI_Info, MPI_Win);
    void remove_window(MPI_Win);
    MPI_Win translate_win(MPI_Win);
    void check_global(MPI_Win, int*);

    ComplexComm(MPI_Comm);

  private:
    MPI_Comm cur_comm;
    std::unordered_map<int, FullWindow> opened_windows;

  //...
};
```

From the declaration, it is possible to split the functions into two groups: on one side we have operations that deal with communicators (replace_comm and get_comm) and on the other the ones working with windows (add_window, remove_window, translate_win, check_global). The first versions are very similar to some functionalities already introduced before (functions replace_comm and the cur_comm global variable): the fact that windows are very related to communicators made them move into the class. While the substitute communicator was a global variable before, with the introduction of this class it becomes a private member of a global ComplexComm object. All the operations that dealt with it previously (access and replacement) have to be done through methods provided by the class (get_comm and replace_comm respectively).

Window operations need to work on non-faulty structures, so all the processes involved must be correctly running. Since the replace_comm operation is called only in case of a fault, it is possible to use it as a way to know that some windows may contain failed processes. Those windows have to be recreated on the new substitute communicator so that no failed process is involved. To enable the recreation, the system must be able to repeat the call that generated the window, so it is mandatory to keep track of all the parameters used to create it.

During the execution, a program is not restricted to the usage of a single MPI_Win: it may create many of them, so we have to keep track and distinguish them all. The class contains an unordered_map that serves this purpose and associates with each window created a unique identifier, used to characterize the element within the collection. These identifiers are connected to MPI_Win structures with the functions MPI_Win_set_attr and MPI_Win_get_attr, taken directly from the MPI standard.

Using the functions add_window and remove_window, it is possible to introduce and remove respectively a window from the map. The first function will

also deal with the identifier management, assigning to each window a unique one that will be shared only with its substitutes. The functions `check_global` and `translate_win` leverage the existence of those identifiers, using them to check if the window has been created with `MPI_COMM_WORLD` and, eventually, returning its substitute. These functions will be the core of the one-sided communication functions that will be explored in the next subsection.

3.6.2 One-Sided Communication functions

The first function we are going to analyse is the `MPI_Win_create`: the function creates a window handler. It needs to be modified to notify the window creation to our `ComplexComm` class. The structure of the implemented function is similar to the others discussed in the previous sections: it contains a loop, the conversion of the communicator, the execution of the actual operation, and ends with the error checking and the eventual substitute replacement. The only difference is that, in case of correct execution, a call to the `add_window` function within the `ComplexComm` global object is performed, and this notifies the creation of the new `MPI_Win`.

It shall be noted also that the `MPI_Win_create` calls the `MPI_Barrier` before executing the standard MPI call: this is done to remove faults from the network and avoid problems (recall property **P.4**). By calling an `MPI_Barrier`, all the faults that happened before are solved since the operation would perform a `replace_comm` call if it finds any error. The `replace_comm` call also fixes the windows since it will recreate them all. This way we can perform calls safely, circumventing the problems shown by property **P.4**. The `MPI_Barrier` call will be present in all the collective one-sided communication functions.

The functions `MPI_Get` and `MPI_Put` are very similar: the first step is checking if the window has a substitute with the function `check_global`. If the answer is affirmative, it is possible to proceed with the substitution (calling the function `translate_win`), the translation of the target rank and the operation using these new values. All the error detection is missing since those functions are treated analogously to point-to-point calls: the `MPI_UNDEFINED` problem is also handled similarly to point-to-point operations.

The last two functions supported are the `MPI_Win_free` and the `MPI_Win_fence`: the first is simple and needs only to call the `remove_window` function from the `ComplexComm` global object before destroying the window, while the second works like an `MPI_Get` or `MPI_Put` without the rank translation part and with the use of the `MPI_Barrier`.

3.7 File operations

The introduction of one-sided communication support shares many concepts with the one of files, so the first step we followed was the refactoring of the code to introduce some generalizations that could help us to reduce the duplicate code. In particular, the part of `ComplexComm` that deals with windows would have been duplicated for files, so it got moved in a template class called `StructureHandler`. The `StructureHandler` class contains five public functions that are similar to all the functions that were present in `ComplexComm` (the only exception is `get_comm`). The

main difference with the ComplexComm version is the extensive use of functional programming: rather than saving the parameters for the recreation, it stores a function that serves the same purpose. This approach allows for the support of multiple ways to create structures. Also, the operations that deal with identifiers are turned into lambda functions since they are strictly related to the structure.

With the introduction of the StructureHandler, the job of the class ComplexComm becomes simpler: it will have to deal with the communicator management mainly. It will contain two StructureHandler objects, one for files and the other for windows: ComplexComm will have to forward all the MPI_Win and MPI_File calls to their respective StructureHandler.

While files and windows share a lot of underlying concepts, there is a remarkable difference: while in MPI_Win there are caching functions like MPI_Win_set_attr, those are not present in files. We had to use a different approach for identifiers management and, after a few tries, we opted for f2c and c2f operations. Those calls are not intended for this purpose since they are designed to allow inter-operability between handles designed for c and Fortran. Luckily for us, Fortran handles structures like communicators, windows, files, etc. differently: rather than having a type, it refers to them with an integer. This integer is unique and associated with each object: two communicators will not have the same integer, even if they are duplicate.

The functions MPI_File_c2f will return the integer given the MPI_File, while MPI_File_f2c will do the other way around. The main difference in using this method compared to the one used for files is that the second allowed to choose the value of the identifier, while in this case is given by default. Nonetheless, this is the way we used to solve the problem and could prove useful eventually also for windows and communicators.

We decided to support a lot of file operations to give the user more flexibility in its choices. We will not go through all the calls, but we will describe the common structure and we will focus more on the functions that feature important differences.

The first function we are going to analyse is MPI_File_read_at since its structure is very basic and can be seen as a starting point for the others. The first step is to translate the MPI_File passed as a parameter and, after that, perform the operation on the translated structure. No error management is needed since the operation involves a single process. Most non-collective functions work analogously, repeating the same steps. Collective calls like MPI_File_read_at_all require the use of the MPI_Barrier before the operation, similar to one-sided communication collectives. They also have the section for error management that will perform the agreement and eventually replace the substitute communicator. These patterns are shared by most of the calls supported, with few exceptions that we will cover in the following lines.

The first function that differs from the pattern is MPI_File_open. It behaves like a normal collective operation (it needs the MPI_Barrier and error management parts), but has to do more: its amode parameter must be changed to remove some flags that could block some important operations introduced for fault tolerance. In particular, files should never be deleted upon close since we may close them earlier than necessary due to the presence of a fault. Moreover, the access should not be exclusive since the system may introduce substitutes for filehandles. Besides modifications to the parameters, the function must add the newly created file to the StructureHandler within

the shared ComplexComm object if the creation is successful and the communicator used was MPI_COMM_WORLD. Also, the MPI_File_close is different since it has to remove the file from the StructureHandler.

The MPI_File_seek_shared adds to the collective pattern exposed above another security measure: the seek operation may not be idempotent, so repeating it twice may give a different result from executing it only once. To make it idempotent, the status of the shared file pointer is saved before the execution and is restored in presence of faults before the next iteration of the loop.

The MPI_File_set_view function is handled like a normal collective file operation, but its support forces us to handle file views. The user can set a file view and expects it to stay the same until further changes. In presence of faults, file handlers are recreated, and the new ones do not directly feature the characteristics of the old ones. A solution consists of the extraction of valuable information from files before they get closed: that information includes file views, pointer positions, and other values that will be set on the new file after creation. This ensures a seamless substitution and will provide no inconsistency to the user.

3.8 Multiple communicators support

Up to now, all the calls supported introduced fault tolerance in calls based on MPI_COMM_WORLD. MPI applications usually create and use other communicators a lot, so it is mandatory to include the support for all of them. An important remark is that new communicators are created from existing ones, so by supporting the calls that generate them it is possible to intercept all the new MPI_Comm structures. Before doing so, it is better to define the classes that will help us by storing all the information related to the created communicators, and how those changes affected the already supported calls.

This Section is defined as follows: subsection 3.8.1 will analyse the current status of the class ComplexComm, pointing out how it could be generalized to be useful for any communicator; Subsection 3.8.2 will focus on the Multicomm class that deals with the presence of multiple ComplexComm objects and will show the changes that are needed on the already supported function calls; Subsection 3.8.3 will cover the communicator management MPI calls supported.

3.8.1 ComplexComm evolution

Up to the point described in section 3.7, the ComplexComm class has been designed to have a single instance that would have been globally accessible. To add the support for multiple communicators, it should handle the fact that it could not be the substitute of MPI_COMM_WORLD only, but of any other communicator as well. Each ComplexComm object should keep track of the communicator it is providing a substitute to, and should expose a function to make it accessible from the outside: this last functionality resides in the get_alias function introduced within the class. We will call the original communicator alias from now on.

Another function introduced is get_group, which returns the MPI_Group of the alias: this information is particularly important during rank translations, and used to be defined globally as the group of MPI_COMM_WORLD. With the presence of multiple

communicators with `ComplexComm`, having groups defined globally is not a good choice anymore.

The last change is just the rename of a function: the `check_global` function name was a reference to the fact that the `ComplexComm` alias was `MPI_COMM_WORLD` and it included all the processes. Removing the reference implies changing the name, to avoid confusion: the new name of the function will be `check_served`.

3.8.2 The `Multicomm` class

The problem of having more than one communicator supported is similar to the one faced when introducing one-sided communication support: for each structure generated, we have to find a way to link it to an object that contains its substitute. In one-sided communication, the linking was performed using identifiers set with the caching functions provided by MPI: these functions are available also for communicators, but we opted for `f2c` functions like for file handlers.

The key idea is to put `ComplexComm` objects within an `unordered_map` and use the integer associated with the alias as an identifier. We will provide methods to translate a communicator into its related `ComplexComm` object so that the functions can use it like they are already doing with the one of `MPI_COMM_WORLD`.

The structure of the class reflects this core idea but adds many functions that are needed to fully realise it. In the code block that follows it is possible to see its definition.

```
class Multicomm {
  public:
    bool add_comm(MPI_Comm);
    ComplexComm* translate_into_complex(MPI_Comm);
    void remove(MPI_Comm, std::function<int(MPI_Comm*)>);
    void part_of(MPI_Comm, int*);
    void change_comm(ComplexComm*, MPI_Comm);

    Multicomm();

    bool add_file(ComplexComm*, MPI_File, std::function<int(MPI_Comm, MPI_File *)>);
    bool add_window(ComplexComm*, MPI_Win, std::function<int(MPI_Comm, MPI_Win *)>);
    void remove_window(MPI_Win*);
    void remove_file(MPI_File*);
    ComplexComm* get_complex_from_win(MPI_Win);
    ComplexComm* get_complex_from_file(MPI_File);
  private:
    std::unordered_map<int, ComplexComm> comms;
    std::unordered_map<int, int> window_map;
    std::unordered_map<int, int> file_map;
};
```

The definition shows the presence of an `unordered_map` containing `ComplexComm` objects (`comms`) and functions used to insert, remove, and check the presence of communicators within it (`add_comm`, `remove`, and `part_of` respectively). The function `change_comm` is just a wrapper of the `replace_comm` function present in the `ComplexComm` class. The function `translate_into_complex` can be used to get the `ComplexComm` object connected to the `MPI_Comm` passed as a parameter. The other functions are needed to manage the correct translation of files and windows since it gets more complicated than previously.

To translate a window or a file it is necessary to know the `ComplexComm` that was used to generate it. This is mandatory since the `StructureHandlers` objects are contained within `ComplexComms` ones. The main problem of this approach is that,

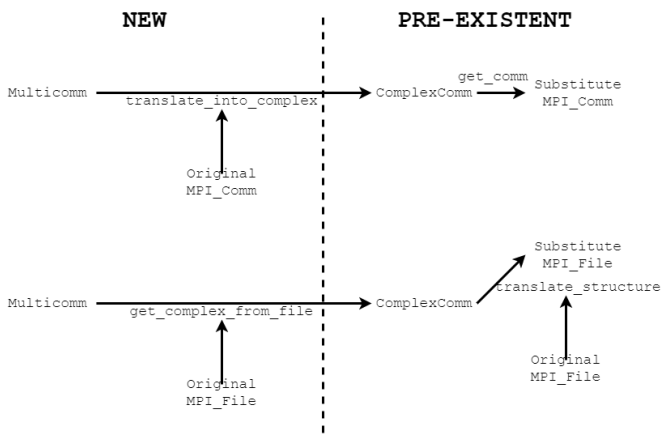

Figure 3.3: *The figure shows the chages to be introduced to support multiple communicators. Changes with the window operations are omitted since they are analogous to the file ones.*

while before there was a single `ComplexComm` object only and it was possible to check the presence of the window (or file) directly, now there are multiple `ComplexComm` and we need a way to find the correct one. To achieve this, we need to obtain information about all the calls that create a window or file and use them to create an association between the `ComplexComm` object used and the structure to be created.

The functions `add_file` and `add_window` serve this purpose: they act as wrappers of the calls in `ComplexComm` that perform the insertion of files and windows respectively. While forwarding the call, they keep track of the association between the structure and the communicator: these associations are stored in the `window_map` and `file_map` structures. After saving the association, it is possible to go from a window (or file) to the `ComplexComm` that generated it with the function `get_complex_from_window` (or `get_complex_from_file` respectively). Association will be removed when windows (or files) are destroyed with the functions `remove_window` (or `remove_file` respectively).

Figure 3.3 summarizes the procedure needed to obtain the conversion of the three main structures considered: `MPI_Comm`, `MPI_File`, and `MPI_Win`. From those procedures, it is possible to proceed with adapting all the already supported calls to the newly added class.

All the added calls are needed to retrieve the correct `ComplexComm` to operate on. All the functions follow the patterns in the figure 3.3, without remarkable differences.

All the functions that operated directly on `ComplexComm` functions have to change their calls and use the `Multicomm` wrappers instead. The `MPI_Init` needs some changes too since the creation of the `ComplexComm` of `MPI_COMM_WORLD` must be done within the `Multicomm` class. The `MPI_Init` function will also create the `ComplexComm` of the `MPI_COMM_SELF` communicator, to ensure that all the possible sources of new communicators are served.

3.8.3 Communicator management operations

Besides supporting the communicators created within the `MPI_Init`, our solution must provide support also for any communicator created starting from a supported one. As a conclusion, we must include support for many operations that produce new com-

municators. The first one that we are going to analyse is the `MPI_Comm_dup`, which produces a duplicate of the given communicator.

Property **P.5** exposes one of the problems we are going to face when dealing with communicator management operations: it is impossible to create a communicator starting from a faulty one. In the `MPI_Comm_dup`, we have two options: either we use the communicator provided as a parameter (which is the alias of its `ComplexComm`) or we use its substitute. Property **P.5** forces us to take the second option since the parameter may provide a faulty communicator, but this means that we may be creating a duplicate of a different communicator (since the substitute does not contain all the failed processes). This is reflected in the assignment of ranks, so the rank in the duplicate communicator may be different from the one in the starting one (also the size may differ).

Aside from the coherency problems, the support is implemented similarly to all the other functions: the first step is obtaining the `ComplexComm` of the communicator passed as a parameter with the function `translate_into_complex`, then we obtain the substitute with the function `get_comm` and we perform the `PMPI_Comm_dup` on it. After that we manage the errors that may arise and, if there are none, we add the newly created communicator to the ones supported with the call `add_comm`: the `Multicomm` object will take care of the creation of its `ComplexComm` and its storage.

The `MPI_Comm_split` will behave analogously to `MPI_Comm_dup`, while the `MPI_Comm_group` created a few more problems, so we decided to base its implementation on `MPI_Comm_split`. We also had to introduce support for the `MPI_Comm_free` and `MPI_Comm_disconnect` functions, that will have to remove the communicator from the `Multicomm` object.

CHAPTER 4

The Hierarchical Extension

THIS chapter explores the first evolution of the Legio library, which introduces a new network layout to reduce the time needed to repair the communicators. We will proceed as follows: Section 4.1 will analyse the reasons that caused this evolution and the previous solutions evaluated; Section 4.2 will make an overview of our solution, pointing out how it affects the execution of all the functions supported; Section 4.3 will cover the reparation procedure; Section 4.4 will expose the preliminary code refactoring steps needed before the integration; Section 4.5 will discuss the actual implementation, focusing a lot on the steps used to repair; lastly, Section 4.6 will perform a computational complexity analysis, proving the theoretical efficiency of the solution.

4.1 Analysis of the problem

The version of our Legio library shown in the previous chapter was useful to show the capabilities of ULFM and to prove that our fault resiliency approach is viable. The use of ULFM proved to be easier than expected, mainly due to the reduced number of functions it introduces. The choice of ULFM was based on the support it is receiving and the extensive usage by many similar applications. Other applications, however, decided to move away from ULFM, pointing out some scalability issues. One of the efforts that deeply analyzed these issues is [17], which clearly defined two weak points of ULFM: the first one is that all the processes within a communicator have to collaborate to repair it; the second one concerns the scalability of the `MPIX_Comm_shrink` operation. Quoting their work:

> *In our experimentation, we found that the cost of ULFM's communicator shrink increased worse than linearly with the system size, and took a non*

> *trivial amount of time to complete. As a result we explored two options to avoid using it.*

Their effort analysed two possible solutions to the scalability problem issued above. The first idea they tried to pursuit was to avoid repairing communicators and continue working with failed ones. Our tests proved that execution can get problematic without repairing communicators since the status becomes undefined. After a few tries, the effort came to the same conclusion:

> *We found out that MPI operations on a non-repaired failed communicator were unpredictable.*

The other approach they tried to pursuit was to reduce the communicator to a set of smaller ones with size equal to two. This reduces the dimension of communicators but greatly increases their number. The problem with this approach is the handling of collective operations, which have to be split across the communicators. Also, the recovery of the execution is difficult to properly synchronize. The effort commented on their results with the following words:

> *The main problem with this design is that the application has to understand the specific details of the recovery procedure, and all possibilities in terms of the order of execution upon failure have to be covered.*

While they managed to obtain some results with this second approach for 1d stencil applications, they decided to move away from ULFM since the difficulty of their approach was rapidly escalating. The idea of moving away from ULFM, while it may simplify the achievement of non-collective repair, was not adopted by us since we think that the effort of the MPI Forum's Fault Tolerance Working Group can lead to its integration within the MPI standard.

4.2 Our hierarchical solution

In this thesis, we decided adopt the communicator size reduction proposed in [17], but with a smaller number of communicators. Our idea bases on splitting the starting communicator into a set of smaller disjoint sub-communicators (we will call them *local_comm* from now on). Each process will be part of a *local_comm* and cannot be part of another for the rest of the execution. Within each *local_comm*, one process will be elected *master*: all the *master*s are part of a communicator, called *global_comm*. Figure 4.1 shows the structure of these communicators.

It shall be noted that there always exists a path from a process to any other: if they are not in the same *local_comm*, the path will go through the *global_comm* with the help of the *master*s of the involved processes. This remark shows the changes that will have to be made within the library: data may need to be forwarded, introducing a new level of complexity. Our solution aims as always to the transparency of the result, making the challenge even harder.

To better analyze the calls supported up to now, we divided them into 9 groups based on their structure. We will cover those types, their meaning, and the algorithm needed to support them in the next subsections.

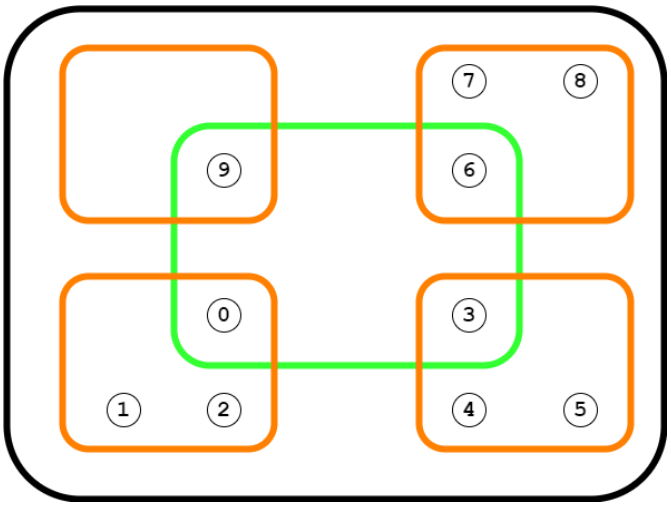

Figure 4.1: *The figure shows the structure of our hierarchical solution. Processes are depicted as small circles containing their rank in the entire communicator. Each rounded square represents a communicator. The black one is the entire communicator. The orange ones are the local_comms, while the green one is the global_comm.*

4.2.1 *one-to-one* operations

We define *one-to-one* operations as the functions that move data from a single process to another without the use of any structure besides communicators. In this group, there are mainly point-to-point operations. The algorithm followed to support them leverages property **P.2** to work on a communicator that is not split (the entire communicator in figure 4.1): upon fault, the communicator is not repaired. This solution avoids us to deal with the forwarding of the messages, which could have introduced unnecessary overheads.

4.2.2 *one-to-all* operations

We define *one-to-all* operations as those collectives that move data from a process to all of them within a communicator: an example of those is the `MPI_Bcast` function. These operations are more difficult to integrate than *one-to-one* since we cannot leverage property **P.2** anymore. Any call issued in a *local_comm* would not reach all the processes within a communicator, and it is better not to use the entire communicator since repair operations may be needed.

The solution can be seen if we analyze the paths that go from the root node to all the others: some of them will only go through the *local_comm* of the root, others will cross the *global_comm* and end up in all the other *local_comm*s. We can see that many paths share some parts and the patterns in each communicator can be seen as a *one-to-all*: all the *master*s in *global_comm* need to receive the data from the *master* of the root's *local_comm*, and all the processes part of a *local_comm* different from the root's one must receive the data from their *master*. Using these considerations, it is possible to define an algorithm that emulates the behaviour of one-to-all functions on our network structure:

1. The first step is the recognition of its role in the communication, made by iden-

tifying the position of the root within the network. The roles from which each process has to choose are:

- Non-*master* member of root's *local_comm* (role 1);
- *Master*s of root's *local_comm* (role 2);
- Non-*master* member of another *local_comm* (role 3);
- *Master* of another *local_comm* (role 4).

2. All the processes must act according to their role:
   - The processes with role 1 have to perform the *one-to-all* call within their *local_comm*, with the real root acting as root;
   - The process with role 2 has to behave like the processes with role 1 at first, but it has also to perform the *one-to-all* call within the *global_comm*, with itself acting as root;
   - The processes with role 3 have to perform the *one-to-all* call within their *local_comm*, with their *master* acting as root;
   - The processes with role 4 have to perform the *one-to-all* call within *global_comm*, with the *master* of root's group acting as root, then they behave like processes with role 3.

The key idea is that the root will propagate the data to the other processes in its *local_comm*, then the *master* of that *local_comm* will propagate the data using *global_comm*, then all the other *master*s will spread the information to all the other nodes using the other *local_comm*s.

This solution allows us to avoid using the entire communicator, which needs a theoretically costly reparation in case of a fault, in favor of *local_comm*s and *global_comm*. This comes at the cost of having to perform many calls for each one done by the application: this fact may affect greatly the overheads, so we will deeply analyze its impact later.

4.2.3 *all-to-one* operations

We define *all-to-one* operations as those collectives that move data from all the processes within a communicator to a single one: an example of those is the `MPI_Reduce` function. *all-to-one* operations are very related to *one-to-all* since the paths are the same but in the inverse direction. The algorithm is analogous and keeps the same roles but changes the second point for processes of role 2 and 4: they will have to perform the two calls in the reverse order.

4.2.4 *all-to-all* operations

We define *all-to-all* operations as those collectives that move data from all the processes within a communicator back to all after performing modifications: an example of those is the `MPI_Allreduce` function. Our approach towards *all-to-all* operations is to split them into *all-to-one* and *one-to-all* and perform the two operations in succession. For each *all-to-all* operation, we have to find the two operations that, once combined, output the desired result.

4.2.5 File operations

File operations have their type, but they do not introduce communication between the processes that should be handled directly. So it is not necessary any forward mechanism, and all the processes can use their *local_comm* to open files. The problem with this solution is that the number of file handles per file is bigger and that can lead to overheads: we will have to check this solution in the performance evaluation part.

4.2.6 Window operations and collective-window operations

Window communication introduces communication between processes. There are two solutions to support them: either we make them work on the entire communicator, losing all the gains obtained by splitting the communicator, or we forward the calls through our network configuration. The latter proved to be very difficult without introducing a big overhead, so we opted not to support them for now, since their support is not mandatory for the applications we plan for our experimental evaluations.

4.2.7 *comm-creator* and *local-only* operations

All the operations that create communicators are within the *comm-creator* type. These operations need the entire communicator to work properly since they need to operate with all the processes within it at the same time. This means that we cannot exploit the features introduced with the hierarchical approach in this case.

local-only operations are the ones that do not involve direct communication between the nodes. Since no communication is needed, we decided to use *local_comm* for them since the eventual repair would be smaller than in the entire communicator.

4.2.8 Positional calls

In section 3.5 we analyzed the support of `MPI_Scatter` and `MPI_Gather`, pointing out that the value of each process rank is meaningful for the correct execution of these operations. While the `MPI_Scatter` follows a *one-to-all* pattern and the `MPI_Gather` an *all-to-one*, executing them like an `MPI_Bcast` and `MPI_Reduce` respectively would produce the wrong result due to the differences in the ranks between the entire communicator and the *local_comm*s. These calls will be referred to with the positional attribute. We decided not to support all these since it would need the use of the entire communicator, reducing the effectiveness of the features introduced.

4.3 Reparation procedure

The network structure described in Section 4.2 allows us to reduce the dimension of communicators used for most MPI calls, at the cost of having more of them. If we assume that the considerations taken from section 4.1 are correct, repairing smaller communicators will be more efficient than fixing the entire one.

To analyze the reparation procedure, we must first consider the communicators used by the operations. *local_comm*s and the *global_comm* will need a new algorithm for the reparation since we must keep the network structure, while the entire communicator can follow a similar pattern as the one proposed in the last chapter. We will not discuss

the reparation of the entire communicator further since it does not introduce anything new.

If we consider our network structure, it is possible to define two semantically different faults that can happen: either a non-*master* node fails or a *master* one. These faults affect the network in different ways: non-*master* nodes faults will be noticed only by the processes within the same *local_comm*, while *master* faults will influence both the *global_comm* and the *local_comm* of the failed process. In both cases, some processes will not notice the error and do not have to participate in the recovery part, so they may be able to proceed with their execution.

Faults of non-*master* nodes are easy to resolve since the only thing needed is to repair the *local_comm* and all the processes within it notice the fault, so no special solution is needed. When a *master* node fails, however, the recovery becomes much more difficult since we must elect a new *master* and put it in the place of the failed one. This substitution proved difficult to realize and we will discuss it in the next subsections.

This section is structured as follows: Subsection 4.3.1 will introduce some concepts, conventions, and conclusions useful for the rest of the analysis while Subsection 4.3.2 will explain the steps each process will have to complete to repair the structure.

4.3.1 Conventions and concepts

The repair procedure leverages a lot of concepts that are not already explained before. In this Subsection, we will cover them to provide a better understanding of the following analysis.

The first assumption is linked to the *master* election: the process should be very simple and all the processes should agree on the same *master*. We implemented this solution by making that the process with rank 0 in the *local_comm* is the *master*: if it fails, the *local_comm* is shrunk and a new process will cover rank 0. This simple implementation removes the need for a complex leader election algorithm.

Another assumption is that the subdivision of processes across *local_comm*s is not done at random, but each process can know which group any process is part of. This assumption is core also for the realization of the function calls exposed in Section 4.2 since all the processes must define their role but can do so only knowing the position of the root. Another important remark is that processes cannot change *local_comm* during the execution: the shrinking operation just removes failed processes from the *local_comm*, no process movement is allowed.

*local_comm*s have a fixed maximum dimension (called K from now on), which is specified in the code and can be tuned by the user at the pre-compile time. This is very useful in our implementation because it helps us define the *local_comm* of each process. To obtain the *local_comm* of a process, it is possible to compute r/K, where r is its rank in the entire communicator: this way we know that processes with ranks from 0 to K-1 will be in the first *local_comm*, from K to $2K$-1 in the second, and so on. This value helps us also to distinguish between *local_comm*s: we can refer to the one containing processes with ranks from 0 to K-1 with the name *local_comm_0*, from K to $2K$-1 with *local_comm_1*, and so on. These names allow us to define the concepts of next and previous: those are the *local_comm_x* where x is equal to the value of the own *local_comm* plus one or minus one respectively. To this definition, it should be included

the circularity of the concept: *local_comm_0* is the next of *local_comm_(s/K+1)*, where *s* is the size of the entire communicator; moreover, *local_comm_(s/K+1)* is the previous of *local_comm_0*.

Another remark defines the ranks of the processes within *local_comm*s and the *global_comm*: they follow the same order as the entire communicator. This property must be enforced since it is very useful for many operations done in the reparation phase (but also in the implementation of the operation as described in section 4.2).

4.3.2 Reparation procedure

The reparation phase begins when a process calls the `replace_comm` function: this happens after the agreement when needed, so we can assume that the BNP is solved. The absence of the BNP allows us to affirm that all the processes within the communicator used for the operation are in the same situation, ready to repair the structure. As stated before, the procedure depends on the role of the communicator.

In case of a fault during an operation in a *local_comm*, all the processes call the `MPIX_Comm_shrink` routine and produce a repaired communicator. Each process will also check if it has become the new *master* (if its previous rank was different from 0 but the new one is 0): if so, it will have to be introduced in the *global_comm*, otherwise the repair procedure is complete.

In case of a fault during an operation in *global_comm*, all the processes call the `MPIX_Comm_shrink` routine and obtain a repaired but smaller communicator. The processes within the group must identify the failed process to determine which *local_comm* has to produce a new *master*. All the operations described up to now are easy to obtain, but then we face a complex problem: the insertion of the new *master* within the repaired *global_comm* is impossible with the only structures defined up to now. The point is, to introduce a process within an already defined communicator we must use inter-communicator functions, in particular `MPI_Intercomm_create`. The function requires the presence of two groups of processes (all the ones within the repaired communicator and the new *master*) each containing a leader and a communicator that contains the two leaders (we will refer to this as the *medium*). If those were the only constraints, then there would be no problem since we can use the entire communicator as the *medium*. Property **P.5**, however, forces us to use a non-faulty communicator, and the entire communicator certainly is not non-faulty. We can try to use it anyway after shrinking it, but we face two other problems:

- The fault is not detected by all the processes within the entire communicator, so the shrink is not possible without some form of error propagation;
- Shrinking the entire communicator is exactly the operation we want to avoid with this approach because of the poor scalability of the `MPIX_Comm_shrink` operation stated in Section 4.1.

Solving the problem without using the entire communicator is impossible with the structures defined up to now because there is not another communicator containing the new *master* and at least a single process already part of the *global_comm*. We need to define a new communicator within the network structure that solves the problem.

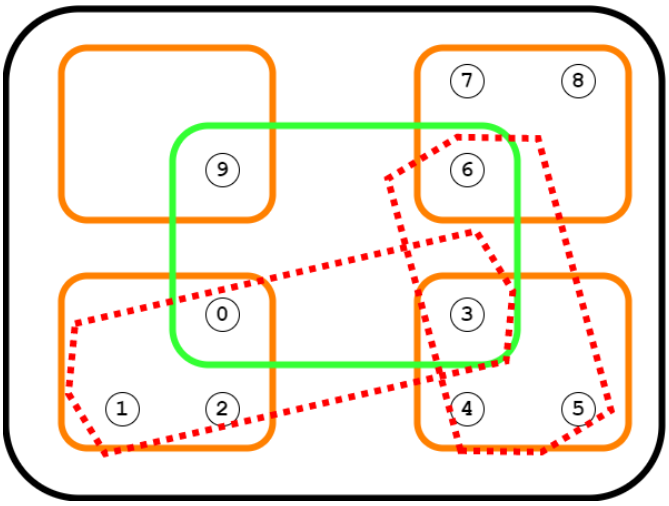

Figure 4.2: *Structure of the hierarchical solution with the addition of POV communicators, represented as the red dashed hexagons. For simplicity only the POV containing the process with rank 3 are displayed.*

Our solution proposed the introduction of *POV* (short for Partially OVerlapped) communicators, which contain all the processes within a *local_comm* plus the *master* of its next. All the non-*master* processes will be part of a *POV* communicator, while the *master* ones will be part of two: their own and the one of the previous *local_comm*. We can refer to *POV* communicators similarly to what we have done with *local_comms*: *POV_i* will contain all the processes within *local_comm_i* and the *master* from *local_comm_(i+1)*. *POV* communicators solve the problem of the absence of a *medium* communicator since it always contains the new *master* and at least a process from the *global_comm*. The fault of the i-th *master* process, however, affects now four communicators: aside from *local_comm_i* and *global_comm*, now also *POV_i* and *POV_i-1* contain the failed process and must be fixed.

The reparation process with the addition of *POV* communicators proceeds like described up to now, with an exception: processes within the *local_comm* of the failed *master* will call the `MPIX_Comm_shrink` also on the *POV* associated with the *local_comm*. The processes within the *global_comm* will be divided into three roles, depending on their position:

- Role 1: the failed *master* was in the previous *local_comm*;
- Role 2: the failed *master* was in the next *local_comm*;
- Role 3: the failed *master* was neither in the previous *local_comm* nor in the next.

There is always at maximum one process of role 1 and one of role 2. We will deal later with the case of a process being both roles 1 and 2 since it is a corner case.

The role 1 process is part of the failed *master local_comm*'s *POV*, so it proceeds with the shrink of that communicator. It checks the existence of another process within the shrank *POV*: if this condition does not apply, then all the processes within *local_comm* failed and there is no new *master*. It communicates the result of this control to all the processes within the *global_comm*, to have a unified view on whether to perform an

`MPI_Intercomm_create` and introduce a new process in the *global_comm* or just leave it shrank. The case when no master has to join is simpler, so we will discuss it later.

After communicating the presence of a new *master*, the process creates the inter-communicator using the *POV* as a *medium*. This procedure must be done by all the processes within *global_comm* and the new *master*: the process with role 1 is the only one able to directly communicate with the new *master*, so it will be the leader of its group (the new *master* will be the other leader). After the call to `MPI_Intercomm_create`, it is possible to obtain a communicator with all the needed processes with the function `MPI_Intercomm_merge`, called by all the participants in the inter-communicator. After that, the only thing left to do is to reorder the ranks in the communicator to keep the order property discussed in subsection 4.3.1: after a `MPI_Comm_shrink` call, the result is the new *global_comm*, repaired and working.

The only communicator left to fix is the *POV* of the *local_comm* previous of the new *master* one. That *POV* must include the new *master*, which will substitute the old failed one. The problem with this substitution is that only the process with role 2 will notice the fault among the ones in *POV*, all the other processes are not *master* and do not communicate with the failed process. To be able to operate on communicators, all the processes involved must collaborate, so we have to design some way to propagate the fault notification to all the non-*master* ones. This notification must be transparent to the processes involved, so we will use a daemon thread: its job will be to periodically check the presence of such notification and, if present, collaborate with the process with role 2. The process with role 2 will free its own *POV* communicator and it will call an `MPI_Intercomm_create`: the two groups will be its local (with itself as leader) and the new *master* (also leader). The *medium* communicator used for the operation will be the newly repaired *global_comm*, which is non-faulty and contains both the leaders. After the calls to `MPI_Intercomm_merge`, the new *POV* is created: here no rank reordering is needed because the merge function allows putting the new *master* at the end of the *POV*, in its expected position.

In case of the absence of a new leader, the same procedure applies but this time the process with role 1 must act as if it was the new leader.

The last part of this subsection is devoted to the analysis of some corner cases. The first one we are going to analyze is the presence of just a single *local_comm*: in this case, the *global_comm* will be a duplicate of the *master*'s `MPI_COMM_SELF` and the *POV* will be a duplicate of the *local_comm*. Upon recognising this structure, the new *master* will create the *global_comm* duplicating its `MPI_COMM_SELF`, simplifying the procedure a lot. The case in which there are two *local_comms* is a bit trickier: if a *master* fails, the one of the other *local_comm* must play accordingly to both roles 1 and 2. So it must behave accordingly to role 2 until the new *global_comm* is spawned, then it should switch to role 1 to fix its own *POV* communicator. All the other cases are already described in the procedure above.

One last remark must be given on the weak points of this algorithm: it is considering only single faults up to now, while in reality many of them may happen at the same time. The algorithm is robust if we consider at most a single *master* fault at a time, while it can be empowered to support non-contiguous *master* faults. Up to now, the current implementation does not support this last evolution.

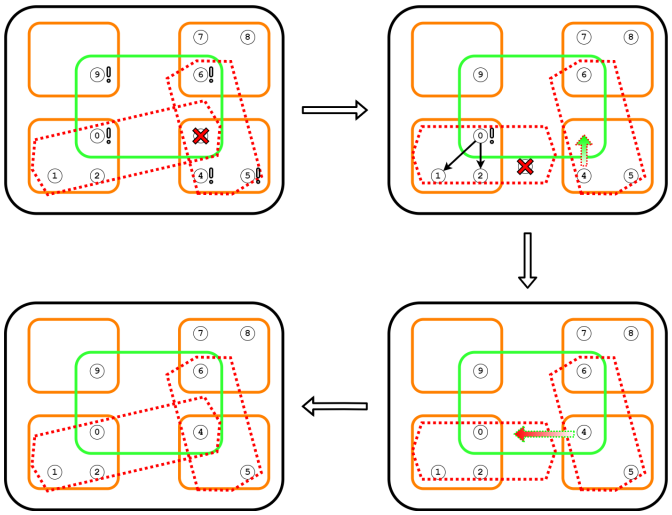

Figure 4.3: *Overview of the repair procedure when a master fails. The communicators and processes follows the notation rules of the previous images. The red cross highlights the failed node. The exclamation marks highlight the nodes that notice the failure. The arrows that originate from a process represent the inclusion of the process in a communicator. The arrow color represents the target communicator. The arrow border color represents the communicator used to perform the operation. The slim black arrow represents the propagation of the failure notification.*

4.4 Preliminary refactoring steps

All the steps of the algorithms explored in the last two sections need many changes in the structure of the library, thus requiring some refactoring. The first step in the refactoring process was the creation of the abstract class `AdvComm`, which substitutes `ComplexComm` as the class containing the substitute of a communicator. The `AdvComm` class will provide almost the same methods as the previous `ComplexComm` class, but will not specify them. It serves as an interface from which it is possible to define new ways to handle the communication. The old `ComplexComm` class is renamed to `SingleComm` and it is now an heir of `AdvComm`: most of the functionality of it will be the same, with some addictions due to other changes. The other heirs of `AdvComm` are `NoComm` and `HierarComm`: the last of those will be discussed in the next section. `NoComm` represents a communicator that is not mapped in the `Multicomm` class: an object of that type will be returned when looking for a non-present `AdvComm`. All the operations regarding `NoComm` will be done on the alias, so it is like that there is no substitution.

The structure of all the functions must change, because they must expose the data movement pattern: rather than extracting the communicator from an `AdvComm` object, they give to it an object containing the call to be performed and the parameters that can change. The function `perform_operation` of `AdvComm` is the one that allows the supported functions to provide the call to be performed: it is an overloaded call that supports many configurations, depending on the `Operation` object passed as the first parameter. The class `Operation` (in particular its heirs) specifies the dataflow and contains a lambda used to move the call to be performed. An `AdvComm` object, upon receiving an `Operation` object with the `perform_operation` function, will launch it according to the implemented network structure: for `SingleComm` it will

just call it on the substitute communicator, for `HierarComm` it will follow the paterns discussed above.

This refactoring part, while seeming useless since it does not introduce new features, defines precious structures that will be the core for the development of the `HierarComm` class, the one that implements the network structure illustrated in section 4.2. The distinction between the `Operations` heirs allows us to define different execution methods, that can vary from call to call: this allows us to run all the supported calls as planned in section 4.2 while keeping a layer of encapsulation.

4.5 The HierarComm implementation

To introduce the network structure discussed in the previous sections and all the algorithms involved we implemented the `HierarComm` class. Its usage is very similar to the one of a generic `AdvComm` object since it is an heir of that class. We will not explore in detail the implementation of the class since many details are present in the previous sections, but we will focus on some aspects that were not the core of the previous analysis.

Before digging into the implementation, it is useful to point out some differences in the nomenclature between the code and the thesis: *local_comm*s are called `local` in the code, *global_comm* is called `global`, the entire communicator substitute is called `full_network`, *POV* communicators are called `partially_overlapped_own` and `partially_overlapped_other` depending if the *POV* is associated to the own *local_comm* or the previous one. The maximum size of a *local_comm* is provided by the macro `DIMENSION`.

The first analysis will cover the constructor of the class, which has to create the entire network infrastructure and initialize the objects contained within the HierarComm instances. The following code snippet explains its functioning.

```
HierarComm::HierarComm(MPI_Comm comm): AdvComm(comm) {
  int rank;       //Rank in alias
  int group;        //Group of which the process is part
  int local_rank;     //Rank in local
  int size;       //Size of alias

  MPI_Comm_rank(comm, &rank);
  group = extract_group(rank);
  local_rank = rank\%DIMENSION;
  MPI_Comm_size(comm, &size);

  //Construction of the needed communicators

  PMPI_Comm_dup(comm, &full_network); //Creation of full_network
  PMPI_Comm_split(comm, group, local_rank, &local); //Creation of local
  PMPI_Comm_split(comm, local_rank == 0, rank, &global); //Creation of global
  if(local_rank != 0)
    PMPI_Comm_free(&global);

  if(rank == 0) {
    //Creation of partially overlapped other, then own
    //...
  }
  else {
    //Creation of partially overlapped own, then other
  }

  //Creation of notifier thread
```

```
  shrink_check = new std::thread(&HierarComm::change_even_if_unnotified, this, group);

  //Creation of other structures, like StructureHandlers
  //...
}
```

From the code, it is possible to see the initialization phases: at first, all the communicators used are created, followed by the *POV*. Here we need a special way to obtain the communicator since they are partially overlapped: all the processes will do the creation of their own *POV* first and eventually will create the *POV* of the previous *local_comm* later. If all the processes would do the same we would get a deadlock since all of them would be waiting for the *master* of the next *local_comm* to participate. By making the process with rank 0 (*master* of *local_comm_0*) do the creations in reverse order, it is possible to solve the deadlock and complete the creation. After the creation of *POV* comes the creation of the notification thread that we will discuss next and the initialization of the other structures that were part also of `ComplexComm`.

The notification thread solves the need expressed in section 4.3, where we exposed the need for asynchronous fault propagation across the *local_comm* that is previous to the one containing the failed *master*. Using this thread, the processes could perform the needed operations to repair their *POV* communicator and keep the status consistent without affecting the execution directly. The function ran by the thread is the following:

```
void HierarComm::change_even_if_unnotified(int rank_group) {
  int rank;
  MPI_Comm_rank(get_alias(), &rank);
  while(1) {
    std::unique_lock<std::mutex> lock(mtx);
    bool timeout = false;
    const auto res = kill_condition.wait_for(lock, std::chrono::seconds(PERIOD));
    timeout = res == std::cv_status::timeout;

    if(timeout) {
      int flag = 0, flag_self = 0, buf;
      if(local != MPI_COMM_NULL) {
        int local_rank;
        MPI_Comm_rank(local, &local_rank);
        MPI_Iprobe(0, 77, local, &flag, MPI_STATUS_IGNORE);
      }
      if(flag) {
        PMPI_Recv(&buf, 1, MPI_INT, 0, 77, local, MPI_STATUS_IGNORE);
        MPI_Comm icomm, temp;
        MPIX_Comm_shrink(partially_overlapped_own, &temp);
        PMPI_Comm_free(&partially_overlapped_own);
        MPI_Intercomm_create(temp, 0, MPI_COMM_NULL, 0, 50 + rank_group, &icomm);
        MPI_Intercomm_merge(icomm, 0, &partially_overlapped_own);
        PMPI_Comm_free(&icomm);
        PMPI_Comm_free(&temp);
      }
    }
    else return;
  }
}
```

The main part of the code snippet above can be addressed within the `MPI_Iprobe` function, which checks the presence of messages that are not received yet. If the presence of un-received messages is confirmed, the execution proceeds with the shrink of the *POV* communicator and the creation of its substitute. The algorithm followed is slightly different from the one exposed in section 4.3 because this one needs to shrink the *POV*, while the other would base the creation of the new *POV* on *local_comm*. This

design choice introduces a new shrink operation but solves problems arising from the unnoticed presence of faults within *local_comm*. Another note can be made on the parameters of the `MPI_Intercomm_create` operation: the first two define the local group and the leader and are correctly set to point to the *master* of *local_comm*; the following two have wrong values, but it is acceptable since the *master* will perform a different call and will provide correct values.

Among all the algorithms discussed before, the only one that received some changes is the realization of the *all-to-all* operations, which encountered a problem that was not addressed before. The code snippet below follows its functioning:

```
int HierarComm::perform_operation(AllToAll op) {
  OneToAll half_barrier([] (int root, MPI_Comm comm_t, AdvComm* adv) -> int {
    int rc;
    rc = PMPI_Barrier(comm_t);
    if(rc != MPI_SUCCESS)
      replace_comm(adv, comm_t);
    return rc;
  }, false);

  perform_operation(half_barrier, 0);

  int local_rank;       //Rank in local
  MPI_Group global_group;   //MPI_Group of global
  int pivot_rank;       //Lowest rank alive

  MPI_Comm_rank(local, &local_rank);
  if(local_rank == 0) {
    int source = 0;                 //Lowest rank alive
    MPI_Comm_group(global, &global_group);
    MPI_Group_translate_ranks(global_group, 1, &source, get_group(), &pivot_rank);
    PMPI_Bcast(&pivot_rank, 1, MPI_INT, 0, local);
  }
  else
    PMPI_Bcast(&pivot_rank, 1, MPI_INT, 0, local);

  int rc = perform_operation(op.decomp().first, pivot_rank);
  rc |= perform_operation(op.decomp().second, pivot_rank);
  return rc;
}
```

The algorithm discussed in section 4.2.4 is used to perform the operation, but with a difference: the algorithm proposed to split the operation in an *all-to-one* plus a *one-to-all* but did not specify the way to choose the root of the two parts. The choice of the root is a very important point since providing a failed node may result in the impossibility to complete the execution (`MPI_UNDEFINED` problem). To be sure that the choice is well made, we decided to introduce a *half barrier* that would fix the communicator before the execution of the choice, so that it is impossible to choose a failed node. The difference between a *half barrier* and an `MPI_Barrier` is the constraints it introduces: the first one ensures that all the processes entered the function, the second adds also that no process may exit from the call before all entered. The choice towards a *half barrier* is for simplicity and avoid a circular reference (the `MPI_Barrier` is implemented as an *all-to-all* operation).

After the execution of the *half barrier* operation, there is the choice of the root process. We decided to choose the lowest rank alive since this way we would need no leader election protocol. The lowest rank alive will always be the process with rank 0 within the *global_comm*: this can be deduced from the considerations that the lowest rank in each *local_comm* will be part of *global_comm* and that the ranks within

global_comm are ordered like in the original communicator. After that, the operation `MPI_Group_translate_ranks` finds which is the rank the application would provide if it was using that process as root, and that value is sent to all the processes within each *local_comm*. After these last calls, all the processes in the network have all the information to perform the *all-to-one* and *one-to-all* calls, so they are performed with the next two functions. The return values of the two parts will be combined and provided as output.

The instances of the class `HierarComm` are created in another place, where there is the function that has to decide how to support a newly created communicator. The function is called `add_comm` and is overloaded to support the various heirs of `AdvComm`. The code snippet below shows the three versions:

```
bool add_comm(MPI_Comm comm, NoComm* source) {
  MPI_Comm alias = source->get_alias();
  int size;
  MPI_Comm_size(alias, &size);
  if(size > BORDER_SIZE)
    return cur_comms->add_comm<HierarComm>(comm);
  else
    return cur_comms->add_comm<SingleComm>(comm);
}

bool add_comm(MPI_Comm comm, SingleComm* source) {
  return cur_comms->add_comm<SingleComm>(comm);
}

bool add_comm(MPI_Comm comm, HierarComm* source) {
  return cur_comms->add_comm<HierarComm>(comm);
}
```

The ending goal of these functions is to add a communicator in the global `Multicomm` object (`cur_comms`): the function `add_comm` of `cur_comms` called at the end of all the functions achieves exactly this purpose. `add_comm` is a template function since it accepts the type of `AdvComm` to add within the `Multicomm` object. The `add_comm` implemented in the code snippet shows how the decision is made: if the communicator that originated the new one is part of a `SingleComm`, then the new one will be a `SingleComm` and analogously for `HierarComm`. In the case of `NoComms`, however, a different decision must be taken since `NoComm` does not support fault tolerance: we decided to opt for `HierarComm` if the size of the new communicator is above a certain threshold specified in the `BORDER_SIZE` macro. This decision comes from the fact that `HierarComm` has been designed to tackle the poor scalability of the shrink operation as described in section 4.1, so it does its best with big communicators. Smaller communicators would suffer from the complexity of the operations and do not gain much from the shrinking of smaller communicators, so it is better to opt for `SingleComm` for them. The user is free to configure the `BORDER_SIZE` macro to fit the configuration used.

4.6 Complexity analysis

In this section, we will cover the theoretical evaluation of our approach, showing how the computational complexity of the algorithm followed during the repair is better than the reparation of the entire communicator. In our analysis we will use the notation $S(x)$ to express the computational cost of the shrinking operation over x nodes: from

section 4.1, we can assume that this function scales worse than linearly, but we do not know how effectively it scales. We will also refer to N as the size of the entire communicator and K as the maximum dimension of the *local_comm*s. We also suppose for simplicity that N is a multiple of K so that all the *local_comm*s will have the same dimension on creation.

This analysis will focus mainly on the impact of the shrink operation, and so it will ignore the cost due to other calls: since those are taken directly from the MPI standard, we can expect their impact to be optimized and, as a consequence, negligible.

The first version of the Legio library needed the whole communicator to shrink upon failure. We can express the cost of the repair operation as $S(N)$. On the hierarchical version of Legio, however, things are different: the complexity changes depending on whether the faulty process was a *master* or not. In the first case, as discussed in section 4.3, we will have to shrink a *local_comm* (size K), two *POV* comms (size $K+1$), and the *global_comm* (size N/K). In the second case, a shrink of the *local_comm* is enough. We can summarize this analysis in the following formula:

$$R_H(N,K) = \begin{cases} S(K) + 2S(K+1) + S(N/K) & \text{if failed node is master} \\ S(K) & \text{otherwise} \end{cases} \tag{4.1}$$

The hierarchical approach is viable if the following statement is valid:

$$\exists N_0(\forall N > N_0(\exists K | R_H(N,K) < S(N))) \tag{4.2}$$

To go further in the analysis, we need a way to combine the two cases discussed above. If we state that faults have the same probability to happen in all the processes involved then it is easy to obtain the combination: if there is a fault, the probability that the failed node will be a master is $(N/K)/N = 1/K$, so we can assume that the other case has probability $(K-1)/K$. This way it is possible to combine the two values in a single expression:

$$\begin{aligned} R_H(N,K) = \frac{1}{K}(S(K) + 2S(K+1) + S(\frac{N}{K})) + \frac{K-1}{K}S(K) = \\ = S(K) + \frac{2}{K}S(K+1) + \frac{S(N/K)}{K} \end{aligned} \tag{4.3}$$

This analysis cannot go further without posing some hypotheses for the complexity of the shrink operation. In the next two subsections, we will cover two hypotheses, and results will be summarized in the third subsection

4.6.1 Linear-complexity case

This section will cover the case in which $S(x) = cx$ for some c. The idea is to find the value of K that minimizes the function as a function of N, and then check the trend of the function analyzed where K is optimal. To obtain the optimal K value in this configuration, we have to derive the function over the variable K and obtain when the derivate is equal to 0. The result is obtained with the following steps (with the assumptions that N and K are strictly positive):

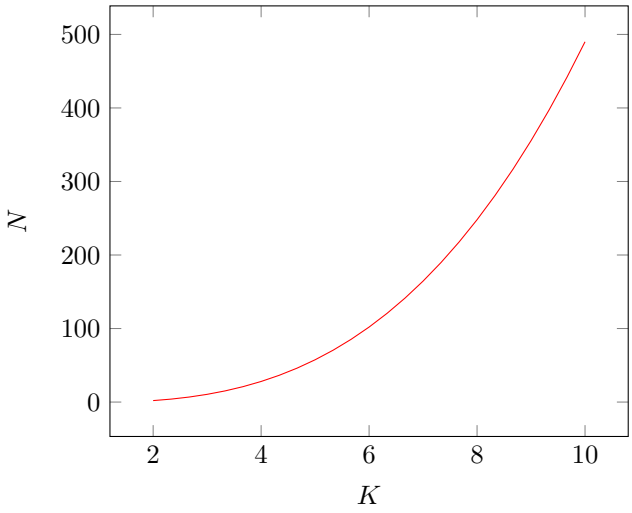

Figure 4.4: *The plot represents the relation between N and K under the linear hypothesis.*

$$\frac{\delta R_H(N,K)}{\delta K} = 0 \rightarrow \frac{\delta(S(K) + \frac{2}{K}S(k+1) + \frac{S(N/K)}{K}}{\delta K} = 0 \rightarrow$$
$$\rightarrow \frac{\delta(cK + \frac{2c(K+1)}{K} + \frac{cN}{K^2})}{\delta K} = 0 \rightarrow$$
$$\rightarrow c(1 - \frac{2}{K^2} - 2\frac{N}{K^3}) = 0 \rightarrow 2\frac{N}{K^3} = \frac{K^2-2}{K^2} \rightarrow N = \frac{K(K^2-2)}{2} \quad (4.4)$$

The result represents the relation between N and K in the optimal case, while it expresses it the other way around. To obtain K as a function of N it is sufficient to extract the inverse of that function (in the first quarter only), but this relation is enough for us. If we substitute it in the original formula we can check if exists a K that makes the assumption valid:

$$cK + \frac{2c(K+1)}{K} + \frac{cK(K^2-2)}{2K^2} < c\frac{K(K^2-2)}{2} \rightarrow$$
$$\rightarrow 2K^2 + 4K + 4 + K^2 - 2 < K^4 - 2K^2 \rightarrow$$
$$\rightarrow K^4 - 5K^2 - 4K - 2 > 0 \quad (4.5)$$

Which is true for $K > 3$. So by this computation, we can say that the optimal solution will lead to an improvement in the cost of repair if the optimal solution is greater than 3 which happens when N is greater than about 11.

4.6.2 Quadratic-complexity case

This subsection will assume that $S(x) = cx^2$ for some c. The procedure is the same as in the previous subsection, but the results are slightly different. The result of the derivation shows the following relation between N and K:

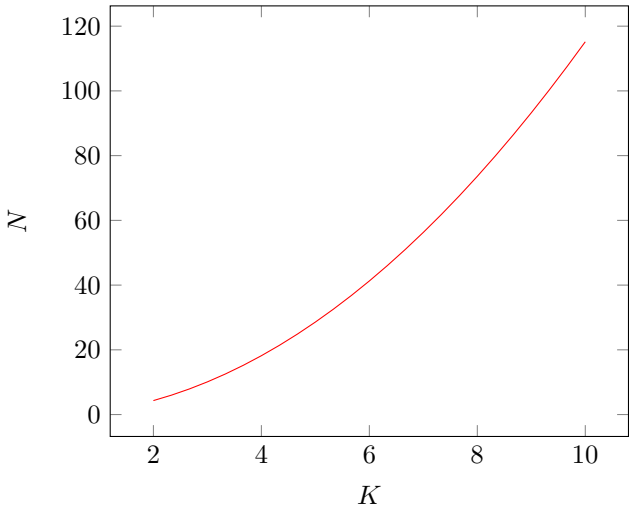

Figure 4.5: *The plot represents the relation between N and K under the quadratic hypothesis.*

$$N = \sqrt{\frac{2K^2(2K^2-1)}{3}} \tag{4.6}$$

After the substitution it is possible to check the statement:

$$\begin{aligned} cK^2 + \frac{2}{K}c(K+1)^2 + \frac{c}{K}(\frac{\sqrt{\frac{2K^2(2K^2-1)}{3}}}{K})^2 < c\frac{2K^2(2K^2-1)}{3} \rightarrow \\ \rightarrow K^2 + 2K + \frac{2}{K} + 4 + \frac{2(2K^2-1)}{3K} < \frac{2K^2(2K^2-1)}{3} \rightarrow \\ \rightarrow 3K^3 + 6K^2 + 6 + 12K + 4K^2 - 2 < \frac{4}{3}K^5 - \frac{2}{3}K^3 \rightarrow \\ \rightarrow \frac{4}{3}K^5 - \frac{11}{3}K^3 - 10K^2 - 12K - 4 > 0 \end{aligned} \tag{4.7}$$

Which is true for $K > 3$. Analogously with the computation of the previous subsection, we can state that the optimal solution will lead to an improvement in the cost of repair if the optimal solution is greater than 3, which happens when N is greater than about 11 also in this case.

4.6.3 Conclusions

The two subsections above showed the complexity effectiveness of the hierarchical approach developed in this chapter. We tested only those cases because we expect the behaviour of the real shrink operation to be somewhere in the middle of the two cases, but we cannot obtain it without further experiments.

It shall be remarked that this complexity analysis covers only the repair phase: the normal operations will take probably more time in the hierarchical approach than in the other one because of the needed propagation. Nonetheless, the reduction of the repair time may make the hierarchical approach viable, and as a consequence, it may be considered by the user when configuring Legio for its execution environment.

CHAPTER 5

Experimental evaluation

This chapter contains the explanation of all the experiments done to validate our solution. The chapter is structured as follows: Section 5.1 will show the purpose of the experiments, illustrating them and describing the HPC system that will host them. From Section 5.2 the results of the experiments will be shown.

5.1 Experimental setup

Solutions for the HPC field must be not only functionally working but also efficient. This is because the dimension of the systems will greatly increase the effect of inefficiencies. While the functional properties has been checked during the implementation of the libraries, the efficiency cannot be evaluated before the development is complete.

To check the efficiency of our solution we evaluated the increase of execution times when using our libraries. To better visualize these increases, we measured also the overhead per MPI-call under various execution configurations. Given the requirements of the Legio library presented in section 3.1, we want these overheads to be as much contained as possible, ideally negligible.

We conducted these experiments on the Marconi100 cluster at CINECA, featuring nodes with 2 x IBM POWER9 AC922 16 cores 3.1 GHz processors and 256 GB of RAM. In all the experiments done we adopted an MPI configuration featuring 32 processes per node, 1 process per physical core, to better exploit the features of the hardware but leaving space for the presence of additional threads spawned by the applications.

All the configuration parameters of the Legio prototypes were left to their default values, except for the maximum size of the *local_comm*s: we set it to the closest optimal value following the relation obtained with the linear complexity hypothesis (Equa-

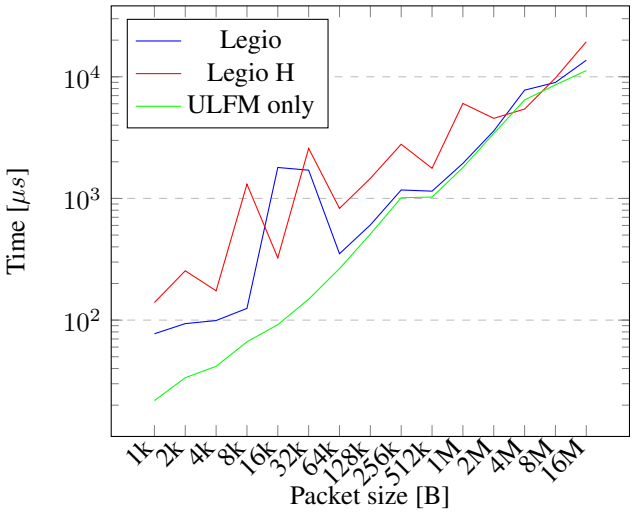

Figure 5.1: *Execution time to complete a MPI_Bcast by varying the message size. Each line represents a different MPI implementation.*

tion 4.4).

The experiments can be divided into two groups, different for their purpose and the information they produce: the first ones involve the per-operation measurement of the overhead introduced, while the second group consists of more general applications in which we will analyze the overall impact of the library. For the first group, we used mpiBench [4] to measure the overhead of the library when increasing the communication load and we used an ad-hoc code to evaluate the same parameters when increasing the network size. We will explore them in the next two Sections.

5.2 mpiBench experiments

mpiBench [4] is a program developed at the Lawrence Livermore National Library that measures the time needed to complete various MPI calls when increasing the dimension of the data exchanged. To do so it measures the time needed to complete a single call and collects the measurements done on each process, computing average, minimum and maximum. We used it to evaluate the impact of our libraries on the `MPI_Bcast` and `MPI_Reduce` operations. The experiments were run on a 32 processes network and in three configurations: 1) the Legio implementation, 2) the hierarchical solution, 3) the application compiled with ULFM without additional libraries.

The results shown in Figures 5.1 and 5.2 display the evolution of the execution times while increasing the packet size. Aside from some noise present in the tests with small packet size, it's possible to see the same behaviour in the late ones: this implies that our solutions do not damage the scalability of the MPI library with the increase of the message size.

5.3 Overhead measurement

To measure the overhead introduced by our solution in the MPI implementation we used some ad-hoc code. The following code snippet contains a part of that code, which

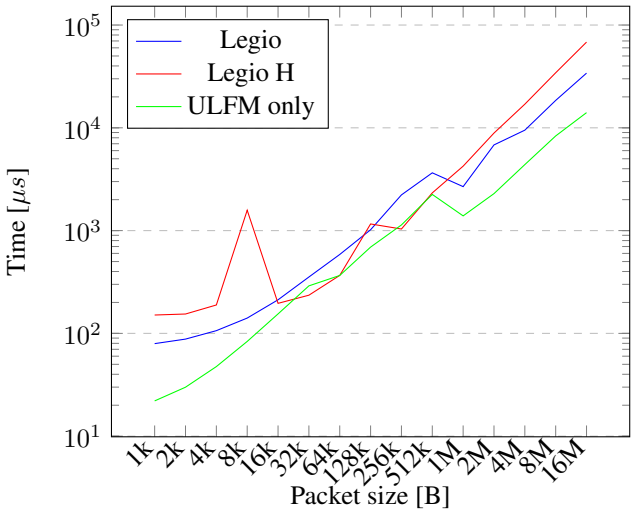

Figure 5.2: *Execution time to complete a MPI_Reduce by varying the message size. Each line represents a different MPI implementation.*

evaluates the overhead of the `MPI_Bcast` operation.

```
int value = rank;
double start = MPI_Wtime();
for(int i = 0; i < MULT; i++)
    MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);
double end = MPI_Wtime();

print_to_file(end-start, rank, size, file_p, "bcast");

value = rank;
start = MPI_Wtime();
for(int i = 0; i < MULT; i++)
    PMPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);
end = MPI_Wtime();

print_to_file(end-start, rank, size, file_p, "bcast original");
```

From the code snippet, it is possible to see that the structure of the code is simple: an MPI operation is repeated many times, and then results are extracted. The extraction is done in the `print_to_file` function, that will compute the average of the results and will write it into a csv file. The same operation is then repeated again with the same parameters: the only difference is the use of the `PMPI` function, which circumvents all the code added by the library. By comparing the two times, it is possible to quantify the overhead introduced by our solution. We used this experiment also to measure the repair time in case of fault. The following code snippet shows the part that measures the repair time.

```
//...
PMPI_Barrier(MPI_COMM_WORLD);

if(rank == 0)
    raise(SIGINT);

start = MPI_Wtime();
MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();

print_to_file(end-start, rank, size-1, file_p, "repair");
```

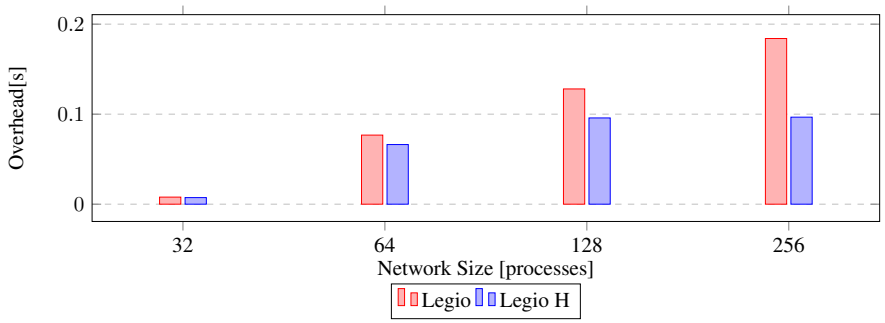

Figure 5.3: *`MPI_Bcast` overhead by varying the network size. Each measure accumulate 100 repetitions of the operation.*

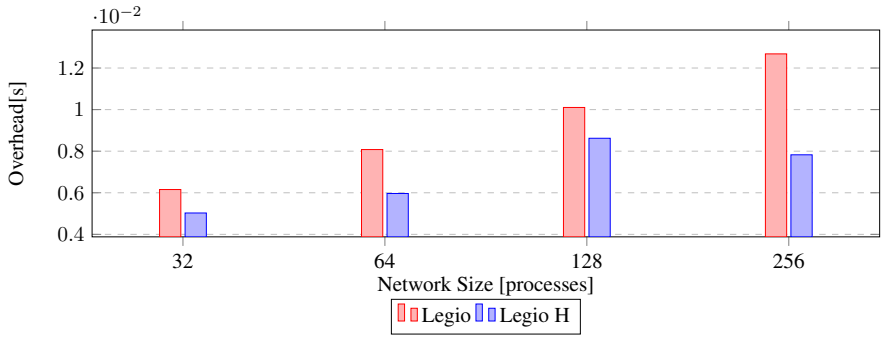

Figure 5.4: *`MPI_Reduce` overhead by varying the network size. Each measure accumulate 100 repetitions of the operation.*

```
12
13 if(rank == 2)
14     raise(SIGINT);
15
16 start = MPI_Wtime();
17 MPI_Barrier(MPI_COMM_WORLD);
18 end = MPI_Wtime();
19
20 print_to_file(end-start, rank, size-2, file_p, "repair again");
```

The `SIGINT` allows us to inject a fault in the execution: by not defining a signal handler, the execution just stops. The following `MPI_Barrier` will find a faulty communicator and will repair it. By measuring the time needed to complete the operation we can quantify the time needed to repair the communicator. The whole measurement is repeated: this way it is possible to measure both a *master* and non-*master* fail in the hierarchical configuration.

Figures 5.3, 5.4, and 5.5 show the results obtained with these experiments, showing that the overheads are limited (values refer to 100 calls). Figure 5.6 shows the results of the repair time analysis: from that, it's possible to see that the non-linearity of the shrink theorized by [17] is not present in our tests, and this limits the testability of the hierarchical approach. Nonetheless, the repair times are in the order of seconds, making them acceptable. We checked also the overhead for file operations: those are more influenced by the load of the file-system rather than other aspects, so we omit those results for simplicity.

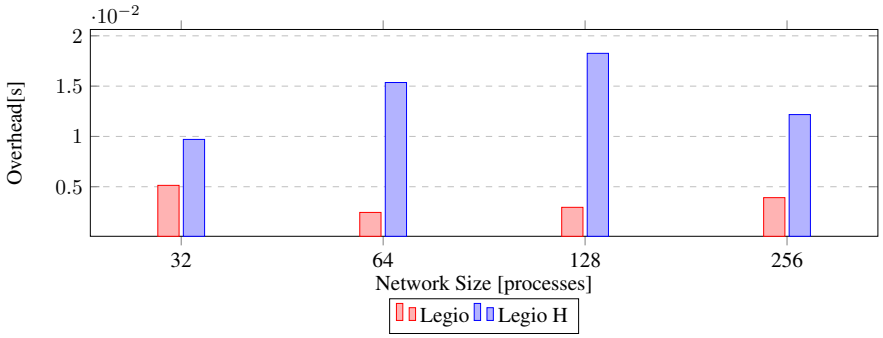

Figure 5.5: *`MPI_Barrier` overhead by varying the network size. Each measure accumulate 100 repetitions of the operation.*

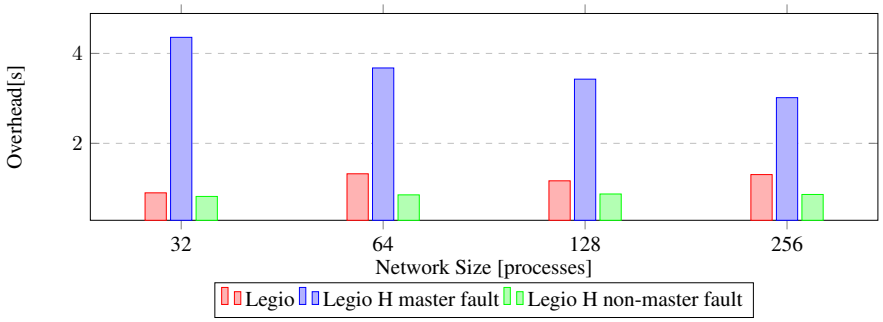

Figure 5.6: *Communicator repair time by varying the number of processes involved in the operation.*

5.4 Embarrassingly Parallel Applications

The second group contains experiments run on two embarrassingly parallel applications: EP from NAS parallel benchmarks [9] and a Molecular Docking miniapp [1].

The first experiment generates independent Gaussian random variates using the Marsaglia polar method. We measured the time needed to complete 40 consecutive executions with the "C" size workload. The second experiment has been executed on the skeleton of a molecular docking application [1], which estimates the strength of the interaction between two molecules. In this context, we have a target molecule and a database of 113K smaller molecules that we need to evaluate to find the most promising ones. Measurements are repeated 10 times and compared with the result obtained using only ULFM. The experiments were done on networks of different sizes, featuring 32, 64, 128, and 256 processes. The results of these executions can be seen in Figures 5.7 for the first experiment: the distributions displayed in the box-plots show that the overhead introduced by the library is negligible.

The second experiment has been run in the same configurations. The results can be seen in Figure 5.8, confirming that the impact of our solution is acceptable.

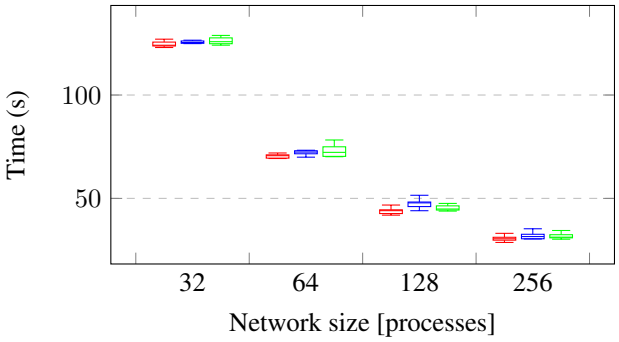

Figure 5.7: *Execution time distribution of the EP benchmark by varying the number of processes involved and the MPI implementation.*

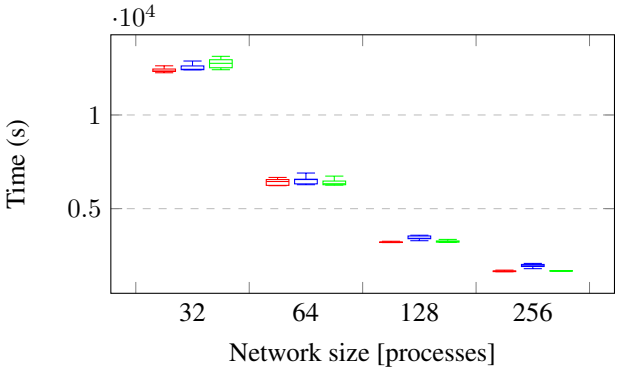

Figure 5.8: *Execution time distribution of the molecular docking application by varying the number of processes involved and the MPI implementation.*

CHAPTER 6

Future work

THIS chapter will discuss all the planned evolutions of the *Legio* library. Section 6.1 will cover the changes that are planned on the first two versions already developed, while section 6.2 will cover the study for a new evolution of the framework that will introduce C/R.

6.1 Legio evolutions

The effort we made on the first two solutions was meant to introduce some fundamental concepts that could be testable and show the potential of this solution. We did not focus on the support of all the MPI calls, but we limited our effort to the most used ones. We have also performed a study on all the functions left to be supported: some of them do not need any change and can be viewed as already supported, some require a new solution (like a way to handle `MPI_Request` structures for asynchronous calls), others are very complex to support. A future effort can introduce the support for those calls to make the framework compatible with even more applications.

Another effort could develop the interface exposed to the application: up to now it is very simple and contains only two functions that can tell to the application the number of failed processes and their rank in `MPI_COMM_WORLD`, but it could include more functions. The idea behind this interface is to allow new applications to provide and gather valuable information to and from the Legio framework: while the application is not forced to do so, it can gain a lot in terms of performance and knowledge of the status of the network. Among the possible introductions within the interface, we think that execution blocks are good features: they allow the user to specify a set of functions to be called as single, and error checking is performed only after the end of those. This way we avoid performing error checking too many times, obtaining an execution

speedup.

6.2 Legio with backline

This section discusses a bigger development planned for the library that would introduce C/R. The purpose of this evolution is to give the user the possibility to choose whether to perform fault resiliency or fault recovery: in some embarrassingly parallel applications, an approximate result is not enough so we need to recover from faults and ensure always correct completion.

In section 2.4.3 we discussed the main efforts that tried to introduce C/R in an MPI application. The distinction between application-level and system-level is very easy to spot and characterizes the efforts produced. While system-level C/R has been preferred during the years due to its simplicity and easy integration with the workload managers present, the rise of ULFM and the creation of all the frameworks that base on it shifted back the attention towards application-level C/R, due to its better efficiency. The main distinction between application-level and system-level C/R resides in the need for instructions: the first cannot obtain what to save and when on its own and must rely on the user to call its functions, the second can obtain the information by inferring them from the executing environment and, as a consequence, can work without big modifications in the source code.

Since all this thesis followed the transparency constraint, it is easy to see the direction we were planning to adopt: system-level C/R, a different approach from all the already developed frameworks. In theory, adapting our code to the C/R framework seems easy: the Legio library can already detect the presence of a fault and repair the communication, it just has to introduce a function to recreate the failed process from the last checkpoint. The only steps left are the choice of the C/R framework and its integration.

Before doing so, it is better to formalize the needed requirements of the C/R framework we will need, to be able to identify the perfect solution more easily. The first requirement is that it should operate at the system-level, as we discussed earlier. A second constraint is that it should save the data of each process separately: if this condition applies, then it is possible to continue the execution on all the non-faulty processes and restart only the failed one, achieving local recovery. Moreover, global recovery with C/R would not innovate since a solution with pure system-level C/R can obtain the same result: upon fault, the framework recognises the problem and restarts the execution, and the application does not need to do anything. To introduce innovation, the constraint on the separation of the processes data must be met.

Other constraints are less strict, such as the ability to initiate a checkpoint within the application and the avoidance of frameworks based on kernel-level code. These constraints are desirable but their absence can be accepted by adapting our solution.

After pointing out all these concepts, it is easy to see that there are not many frameworks that fit our constraints. The one that comes the closest is DMTCP [8], which introduces system-level C/R without the need for kernel-level code. It also supports a plug-in to perform checkpoint creation calls from the code of the application. The problem with DMTCP is the absence of support for the separation of the checkpoint data: the C/R framework is designed for various distributed applications, not only the

MPI-based ones. The way the framework supports distributed application is by keeping track of all the processes and threads generated by the starting program: to say that with the terms used by DMTCP developers, "DMTCP is contagious". This means that, rather than introducing support from MPI digging deep into its implementation, DMTCP can just checkpoint the `mpirun` application, which will create the processes that will become the MPI ones. DMTCP does not care about the division between the processes since it handles the execution as a unique homogeneous entity. This approach is problematic for us.

One of the latest evolution of the DMTCP framework is MANA, short for MPI Agnostic Network Agnostic [18]. The idea behind MANA is to improve DMTCP, especially for MPI-based applications, allowing them to be restarted using a different version of MPI and a different network topology. This new feature is very desirable since it allows us to stop the execution on a cluster and resume it on another, without compatibility problems.

Another interesting point of MANA is the way they achieve MPI agnosticism since interoperability across MPI implementations is an unexplored field. The lack of attention towards that comes from the fact that so many MPI implementations exist, and it would be hard to support them all. The solution MANA adopted is simple and yet very interesting: the framework does not save details about the MPI implementation but limits itself to keeping track of all the produced MPI structures (like `MPI_Comm`, `MPI_Group`, etc.). So they split the process space into two halves, where the topmost will contain the application and will be saved to the disk, while the second will hold the libraries and the frameworks needed and will be discarded. Upon restart, the second part is reconstructed and the first half is loaded, so the execution can proceed even with another MPI implementation.

To be able to migrate the processes from a network configuration to another, checkpoints are saved split: there will be one file per process. This is exactly the feature we need for our solution, and it is also the only effort based on system-level C/R to have it. Moreover, MANA has been developed as a DMTCP plug-in, so it will contain all the other features of that C/R framework.

We contacted the researchers that produced the MANA framework, and we discussed our idea of integration within Legio. From our conversation we could take some conclusions:

- MANA has been designed and tested for the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC). They worked mainly with two implementations of MPI, Intel MPI, and Cray MPICH. Our attempts to make it work with ULFM (based on OpenMPI) were not successful due to some compatibility problems. MANA is still relatively new and it is far from being production-ready, it will need some more development;
- The integration of MANA within our Legio framework is not seamless: MANA expects all the processes to restart in presence of faults, while our solution was planning to achieve local recovery, removing the need for all the processes to restart.
- Legio does not need all the features introduced by MANA, but just a subset of them: one of the biggest difficulties of the MANA implementation is being able

to recreate the process after a restart in a way that by loading the checkpoint the execution can continue. Legio does not need that part since the failed process checkpoint will be loaded on a node created with the `MPI_Comm_spawn` function.

- The development of our solution is feasible, but takes more time than expected since we will have to deal with low-level code: MANA digs deep into memory to obtain the data it needs to run (like the position of the blocks in memory, their access privileges, open files, etc.).

The innovations introduced with this last evolution could improve the scope of the Legio framework: it will be possible to support applications more complex than embarrassingly parallel ones since the errors will not cause other nodes to fail. This adaptation, however, increases the complexity of the evolution because it introduces more communication that must be handled at restart time.

CHAPTER 7

Conclusion

This thesis presents Legio, a framework designed to offer resiliency to embarrassing parallel MPI applications. The work makes the absence of intrusiveness in the target application one of the key elements. Indeed, the library makes use of the PMPI interface to wrap the MPI call and to implement all the required actions to manage failed processes. ULFM has been used as a base for the implementation of Legio.

In the thesis, an extension towards a hierarchical implementation has been done to reduce the overhead of the repair process in case of a large number of nodes involved. Within the document, also a theoretical analysis of when to apply the hierarchical version has been discussed.

The experimental evaluations considering both per-MPI-call and application-level evaluations demonstrate the efficiency of the implemented framework, proving how the solution can be used in embarrassingly parallel applications without affecting the overall performance.

Overall, the problem of dealing with faults in an MPI application is still complex and needs additional efforts. The proposed solution is very specific both in the target application and the recovery policy (fault resiliency), and it does not apply to a generic MPI application. Nonetheless, embarrassingly parallel applications are intrinsically scalable and are compatible with the future exascale architectures, and in those systems, fault tolerance will be even more important.

List of Figures

List of Tables

Bibliography

[1] Exscalate4cov - exascale smart platform against pathogens. http://www.exscalate4cov.eu/.

[2] Marconi100, the new accelerated system. https://www.hpc.cineca.it/hardware/marconi100.

[3] Mpi: A message-passing interface standard, version 3.1. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[4] mpibench: Mpi benchmark to test and measure collective performance. https://github.com/LLNL/mpiBench.

[5] Summit, oak ridge national laboratory's 200 petaflop supercomputer. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

[6] Julien Adam, Jean-Baptiste Besnard, Allen D Malony, Sameer Shende, Marc Pérache, Patrick Carribault, and Julien Jaeger. Transparent high-speed network checkpoint/restart in mpi. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–11, 2018.

[7] Saman Amarasinghe, Dan Campbell, William Carlson, Andrew Chien, William Dally, Elmootazbellah Elnohazy, Mary Hall, Robert Harrison, William Harrod, Kerry Hill, et al. Exascale software study: Software challenges in extreme scale systems. *DARPA IPTO, Air Force Research Labs, Tech. Rep*, pages 1–153, 2009.

[8] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

[9] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[10] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[11] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In *Programming environments for massively parallel distributed systems*, pages 213–218. Springer, 1994.

[12] Kiril Dichev, Kirk Cameron, and Dimitrios S Nikolopoulos. Energy-efficient localised rollback via data flow analysis and frequency scaling. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–11, 2018.

[13] Jack Dongarra, Pete Beckman, Patrick Aerts, Frank Cappello, Thomas Lippert, Satoshi Matsuoka, Paul Messina, Terry Moore, Rick Stevens, Anne Trefethen, et al. The international exascale software project: a call to cooperative action by the global high-performance community. *The International Journal of High Performance Computing Applications*, 23(4):309–322, 2009.

[14] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *Acm sigplan notices*, 47(8):225–234, 2012.

[15] Graham E Fagg, Antonin Bukovsky, and Jack J Dongarra. Harness and fault tolerant mpi. *Parallel Computing*, 27(11):1479–1495, 2001.

[16] Marc Gamell, Daniel S Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 895–906. IEEE, 2014.

[17] Marc Gamell, Keita Teranishi, Michael A Heroux, Jackson Mayo, Hemanth Kolla, Jacqueline Chen, and Manish Parashar. Local recovery and failure masking for stencil-based applications at extreme scales. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[18] Rohan Garg, Gregory Price, and Gene Cooperman. Mana for mpi: Mpi-agnostic network-agnostic transparent checkpointing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 49–60, 2019.

[19] William Gropp and Ewing Lusk. Fault tolerance in message passing interface programs. *The International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[20] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494, 2006.

[21] Umar Kalim, Mark K Gardner, and Wu Feng. A non-invasive approach for realizing resilience in mpi. In *Proceedings of the 2017 Workshop on Fault-Tolerance for HPC at Extreme Scale*, pages 1–8, 2017.

[22] M Farrukh Khan and Raymond A Paul. Pragmatic directions in engineering secure dependable systems. In *Advances in Computers*, volume 84, pages 141–167. Elsevier, 2012.

[23] Nuria Losada, Leonardo Bautista-Gomez, Kai Keller, and Osman Unsal. Towards ad hoc recovery for soft errors. In *2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, pages 1–10. IEEE, 2018.

[24] Nuria Losada, George Bosilca, Aurélien Bouteiller, Patricia González, and María J Martín. Local rollback for resilient mpi applications with application-level checkpointing and message logging. *Future Generation Computer Systems*, 91:450–464, 2019.

[25] Nuria Losada, Iván Cores, María J Martín, and Patricia González. Resilient mpi applications using an application-level checkpointing framework and ulfm. *The Journal of Supercomputing*, 73(1):100–113, 2017.

[26] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[27] Stefan Pauli, Peter Arbenz, and Christoph Schwab. Intrinsic fault tolerance of multilevel monte carlo methods. *Journal of Parallel and Distributed Computing*, 84:24–36, 2015.

[28] Rodrigo Schmidt, Islene C Garcia, Fernando Pedone, and Luiz Eduardo Buzato. Optimal asynchronous garbage collection for rdt checkpointing protocols. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 167–176. IEEE, 2005.

[29] Faisal Shahzad, Jonas Thies, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard 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.

[30] Nawrin Sultana, Anthony Skjellum, Ignacio Laguna, Matthew Shane Farmer, Kathryn Mohror, and Murali Emani. Mpi stages: Checkpointing mpi state for bulk synchronous applications. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–11, 2018.

[31] Keita Teranishi and Michael 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.

[32] Rajeev Thakur and William Gropp. Open issues in mpi implementation. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 327–338. Springer, 2007.