Noname manuscript No.
(will be inserted by the editor)

Legio: Fault Resiliency for Embarrassingly Parallel MPI Applications

Roberto Rocco · Davide Gadioli · Gianluca Palermo

Received: date / Accepted: date

Abstract Due to the increasing size of HPC machines, dealing with faults is becoming mandatory due to their high frequency. Natively, MPI cannot handle faults and it stops the execution prematurely when it finds one. With the introduction of ULFM (User Level Fault Mitigation), it is possible to continue the execution, but it requires complex integration with the application. In this paper we propose Legio, a framework that introduces fault resiliency in embarrassingly parallel MPI applications. Legio exposes its features to the application transparently, removing any integration difficulty. After a fault, the execution continues only with the non-failed processes. We also propose a hierarchical alternative, which features lower repair costs on large communicators. We evaluated our solutions on the Marconi100 cluster at CINECA with benchmarks and real-world applications, showing that the overhead introduced by the library is negligible and it does not limit the scalability properties of MPI.

Keywords MPI · ULFM · Fault Tolerance · HPC

1 Introduction

The high demands of computational science applications are leading the evolution of the current high-performance systems, increasing 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 [1, 2]. This evolution introduces 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.

Modern HPC architectures are featuring millions of cores and components, and the probability that at least one of them is the victim of a fault rises with

Roberto Rocco, Davide Gadioli and G. Palermo, are with Dipartimento di Elettronica, Infomazione e Bioingegneria, Politecnico di Milano, Italy. {name.lastname}@polimi.it

these numbers. The mean time between failures of current systems is measured in days [3], and probably in the future systems will be measured in minutes [4]. With this high frequency of faults, the MTBF of the system can be lower than the application run-time. Without any explicit management, an application would have to be restarted several times up to when it is capable to reach the end of the computation without any problem. Most applications based on MPI [5], the de-facto standard for inter-process communication, lack reliability management since the standard assumes that the application executes in a controlled environment, where all the system components work properly. This implies that applications must feature some sort of reliability management to reach the end of the execution.

This problem has been solved mainly by leveraging Checkpoint-and-Restart (C/R) techniques, but with the reduction of the MTBF new solutions are needed, because the time needed for the checkpoint can easily exceed the MTBF value [6]. To avoid relying purely on C/R, during the years several MPI implementations featuring reliability methodologies has been developed, such as MPICH-V [7], rMPI [8], or FT-MPI [9]. These efforts try to introduce reliability methodologies directly in MPI, creating new functionalities in the existing standard. While remarkable, they received only limited support and did not solve entirely and efficiently the problem. The last effort among those is the User-Level Fault Mitigation (ULFM) [10] MPI extension: it's a collection of functions that allow the user to repair and continue its MPI execution. This work is receiving a lot of attention, mainly due to the focus on the integration in the MPI standard: the next version of MPI (4.0) will focus on reliability, and ULFM is one of the candidates to be introduced in the standard.

Various efforts (such as Fenix [11], CPPC [12], LFLR [13]) have been developed on top of ULFM since it provides an interface to handle a fault and to repair the related data structures. These frameworks couple ULFM with a method to restore the execution (typically C/R based) and create an all-in-one tool improving the reliability of an MPI application. 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 it 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 present Legio[1], a framework that introduces fault resiliency in MPI applications without requiring any integration effort from the application developers, in terms of lines of code to be changed. The main difference between fault resiliency and C/R solutions provided in other efforts (called fault tolerance) is that in the former there is no focus toward the recovery of a consistent state but the application continues without recovering.

[1] The name Legio comes from the 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.

This means that, upon noticing an error, the failed processes are discarded and the execution continues only with the non-failed ones. This approach is faster compared to the standard C/R proposed in the other frameworks, but impacts the correctness of the application result: an acceptable trade-off for applications producing an approximate result, like for example Monte Carlo solvers [14], or high-throughput in-silico virtual screening applications [15].

We can achieve this goal since we target embarrassingly parallel MPI applications, a very common and scalable type of parallel program that reduces to the minimum the interactions between the processes. Embarrassingly parallel applications are also envisioned to be among the first ones capable to fully exploit an exascale system. Typically, they use MPI I/O to maximize the data transfer between computation nodes and the file-system, while avoiding as much as possible explicit synchronization between them.

Legio supports most used MPI calls in embarrassingly parallel applications together with one-sided communication and file support, features not yet included in ULFM. 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, reducing the time to repair in larger communicators. We evaluate Legio on the Marconi100 cluster at CINECA [16] 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 paper 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;

The remainder of the paper is organized as follows. Section 2 analyzes the previous works that tried to solve the problem and introduces some definitions and knowledge useful for the following sections. Section 3 covers the initial exploration of the ULFM behaviour in presence of faults. Section 4 exposes the design process of the Legio framework. Section 5 analyzes the hierarchical alternative of the framework. Section 6 goes through the experimental evaluation of our work by showing the overhead at the MPI call and application-level. Section 7 discuses some potential improvements of the produced implementations. Lastly, Section 8 concludes the paper.

2 Background and related work

When an MPI process detects a failure in another process, e.g. a segmentation fault, the default behaviour is to propagate this information and to stop all

the processes that compose the application. However, if we are willing to react to a failure, we can proceed in two main directions. On the main hand, we can adapt and continue to execute with fewer processes, i.e. fault resiliency. On the other hand, we can try to replace the faulty process with a new one and continue the elaboration, i.e. fault recovery.

ULFM [10] is one of the most relevant efforts in the field since it allows to continue the execution past the detection of a fault. Indeed, it specifies a set of functions to enable fault tolerance in MPI applications. The main ULFM features that we use in our approach are the following: (a) the possibility to set a communicator as out of order (revoked), (b) the possibility to remove failed processes from a communicator and obtaining a working one, (c) the possibility to agree on a result even in presence of faults, and (d) the possibility to identify failed processes.

Many frameworks 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 [11, 13, 17, 18]. 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 usually do not pursue transparency and, rather than opting for fault resiliency, they recreate the failed processes. Among those efforts, the ones presented in [12,19] do not need code changes in the application: those adaptations are made automatically by the framework using a heuristic analysis. While their solution achieves transparency, they are different from the proposed approach since they do not opt for fault resiliency. All these frameworks achieve a solution that can work with any MPI application, but our proposed approach can obtain better results in terms of performance overhead for embarrassingly parallel applications.

A completely different perspective is the one presented when applying algorithm-based fault tolerance (ABFT) [20], which exploits the possibility to obtain 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 approach that does not involve the recreation of the failed processes has been explored in two previous work [22, 23] that propose a solution similar to Legio. For example, [22] discussed in detail the need for rank mapping

between the communicators pre- and post-failure, while [23] adopted a network topology that is very similar to the one discussed in Section 5, with the only difference being the presence of reliable nodes. However, both of them tackle a very specific problem and it is not trivial to generalize their approach.

An effort that shares many concepts with the approach we are proposing has been presented in [14]. 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 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 more in-depth analysis of the current state-of-the-art solutions leveraging ULFM can be found in [24].

3 Preliminary analyses

In this section, we will discuss some issues of the ULFM implementation of the MPI standard in presence of faults [18, 25]. 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 presence of a fault;

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_Group`s. 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 do not work in a failed communicator.

P.3 Collective communications will not work in a failed communicator but may partially work in a faulty communicator. This behaviour comes from the fact that only some of the processes may notice the fault, while the others can complete without problems. In particular, the `MPI_Bcast` operation exposes this behaviour, unlike `MPI_Reduce`, `MPI_Barrier`, and `MPI_AllReduce`,

since those may need a feedback from the receiver on the correct reception of the message. 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.

These points are used in the next Sections to justify some of the choices done while designing the proposed framework.

4 The Legio framework design and architecture

The basic idea behind the Legio framework is that it has to provide fault resiliency functionalities in embarrassing parallel applications without code intrusiveness. To achieve our purpose, we designed a library that behaves like an intermediary between the application and the MPI implementation by exploiting the MPI profiling interface (`PMPI`), which is in the standard and it allows us to intercept 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.

The proposed solution consists of the substitution of the MPI structures used (and created) by the application with others managed by Legio. In this way, when a fault happens, it affects only the Legio structures, making the repair process easier and controllable by the framework. The MPI operations performed by Legio are the ones called by the applications, but with different MPI structures and ranks of the involved processes. In particular, the MPI structures that are involved in the Legio repair process are communicators, windows, and files. The structure substitution introduces many problems that must be addressed, all referring to the possible differences between the original and the substitute. For what concerns communicator substitution, the ranks of the processes may raise some problems: the application is expecting its rank not to change during the execution, but we may have to change the communicator due to faults and, as a consequence, ranks. Our solution must be able to transparently map ranks from the original structure to the substitute one.

Another problem that arises in this situation is the fact that faults may heavily affect the correctness of the application result. While we expect an accuracy loss as a consequence of a fault, the impact of such a loss depends on the role of the failed process within the application. Working transparently at the application level implies that Legio has no way to know the importance of a process within the application, and neither the application has any way to tell Legio that information. Legio infers the importance of a process from the

communication patterns observed and adapts its behaviour based on these considerations. In particular, processes that are not the root of a collective call are assumed less important than the root and their fault does not alter the completion of the operation: after repairing the communicators, the calls are repeated. On the other side, when a failed process is involved in the communication, either by being the root of a collective call or by participating in a point-to-point operation, there are two possible courses of action. Legio can ignore the failure, for example when the failed process was gathering data from the others, or it can stop the application execution, for example when the failed process is spreading important data. The choice is done at Legio compile-time and we provided ways to the user to configure this behaviour to better fit the application.

The presence of a fault is checked after the execution of the operation with the substitute structures: if it is confirmed, then the structures must be repaired and the operation must be repeated. Since ULFM supports communicator repair only if all the processes participate in the procedure, the error checking routine is not performed in non-collective calls. The error checking routine suffers from the **BNP** (property **P.3**): since all the processes need to participate, the fact that only some processes notice the fault can block the repair process, resulting in a deadlock. To avoid this problem we perform an agreement operation that combines the results obtained by all the processes into a single one equal for all, so that either all the processes notice the fault (and can proceed with the repair procedure) or none, avoiding deadlocks.

While communicators management is enough to support many MPI functions, there are many more that do not base on them. All the operations referred to in property **P.1** are left unchanged, while others need some additional structures. File operations and one-sided communication ones, in particular, leverage other structures not yet supported by ULFM so any fault may cause the program to behave indefinitely (property **P.4**). Any operation that uses one of these structures must be sure of the absence of faults because we cannot repair those structures and the execution would stop. The solution adopted up to now faces the problem of having to ensure that the substitute structure is fault-free before executing the operation. To achieve this requirement, we added a call to a barrier operation before the actual function: in this way the eventual presence of a fault will be recognised by the barrier and it will be possible to proceed with the repair.

These solutions allow us to support most of the MPI calls, but other functions like the gather and scatter operations rely on the value of the rank to provide correct behaviour, and simply running them on a substitute communicator would produce a wrong result. We decided to implement those functions as a combination of others that do not suffer from the same problem.

By following these concepts, we managed to create an implementation of our library that features support for many MPI operations[2]. This solution is

[2] The source code of the Legio framework can be found here, together with a list of all the MPI calls currently supported: `https://github.com/Robyroc/Legio`.

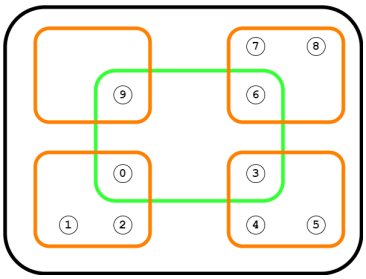

Fig. 1: An abstraction of a MPI application. Processes are depicted as small circles containing their rank in the target communicator. Each rounded square represents a communicator. The black one is the target communicator. The orange ones are the *local_comm*s, while the green one is the *global_comm*.

transparent: the application needs no strict code change to support the library because it is integrated only in the linking phase. However, the application developers must be aware that an MPI operation may be skipped, due to the rank translation problem. Therefore, they must perform all the operations required to avoid undefined behaviour, such as buffer initialization. We evaluated our solution to measure the overhead it introduces and its ability to handle faults: we will present the results in a later section.

5 The Hierarchical Extension

The ULFM standard requires that all the repair procedures involve all the processes, limiting the development of local recovery solutions where each process could repair itself independently [11,18]. This is a well-known issue, analyzed by the same authors, that leads to worse than linear scaling when we increase the number of nodes involved in the computation. Given the modern trend to increase the experiment size, the impact of this limit increases as well.

To solve this issue, we propose an alternative and novel solution that avoids the `MPIX_Comm_shrink` usage on the entire communicator. In particular, we developed a hierarchical approach. At first, we split the target communicator into a set of disjoint sub-communicators (*local_comm*s). Then, we create a new communicator (*global_comm*) that contains one process (named *master*) per sub-communicator. The *master* process of a sub-communicator is the one with the lowest rank. Figure 1 shows the topology of the hierarchical approach.

This solution has some major properties: *(a)* the number of communicators created scales linearly with the number of processes; *(b)* each process can reach anyone else in the network (if not directly, via forwarding), and *(c)* there is only one path from a process to another one that crosses the minimum amount of nodes. The new communicator resembles a star topology, avoiding any communication across different *local_comm*s outside the *global_comm*. On the main hand, this feature reduces the impact of a fault: only the processes

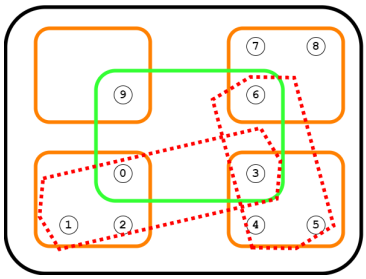

Fig. 2: Example of *POV*s communicators in the MPI application depicted in Figure 1. Each POV is represented as a dashed exagon. For simplicity we depict only the *POV* of which the process with rank 3 is part.

directly communicating with the failed one will have to participate in the recovery, while the others can continue their execution seamlessly. On the other hand, it complicates the repair procedure which depends on the role of the process.

When the faulty node is not a *master*, then the repair procedure is bounded within the *local_comm*. Otherwise, the framework needs to assign the *master* role to a new process and include it in the *global_comm*. In particular, every time the user creates a communicator (of size s), Legio creates *local_comm*s of max size k. The framework assigns each process to a *local_comm* according to its rank r i.e. a process will be assigned to the i-th *local_comm* (*local_comm_i*) if $i = r/k$. Moreover, we define *local_comm_(i+1)* as the successor of *local_comm_(i)*, while we define *local_comm_(i-1)* as its predecessor. We consider the last *local_comm* the predecessor of the first *local_comm*. The assignment of a process to a *local_comm* is final.

Due to property **P.5**, when Legio manipulates a communicator, it must be fault-free. Therefore, the framework needs to create additional communicators to complete the repair procedure, named *POV*s (short for Partially OVerlapped). Each *POV* includes all the processes of a *local_comm* and the *master* of the successor. Thus, Legio creates a *POV* for each *local_comm*. Legio uses these communicators only for the repairing procedure. Figure 2 highlights two *POV* communicators in the example depicted in Figure 1.

Figure 3 summarizes the required steps when we repair a failure on a *master*. The failure is noticed only by the processes in its *local_comm* and by the ones in the *global_comm* (Figure 3a). However, the failed process belongs to four different communicators and all of them must exclude the failed process to proceed. The *local_comm*, its *POV*, and the *global_comm* can shrink to exclude the failed *master* process. However, the *master* of the predecessor needs to notify the processes in its *POV* before shrinking, since they were unable to notice it directly. In this phase of the repair procedure, the processes in the *local_comm* of the failed *master* can communicate with the other processes only by using their *POV* through the *master* of the successor. Legio uses this connection to include the new *master* node, i.e. the process with lower rank

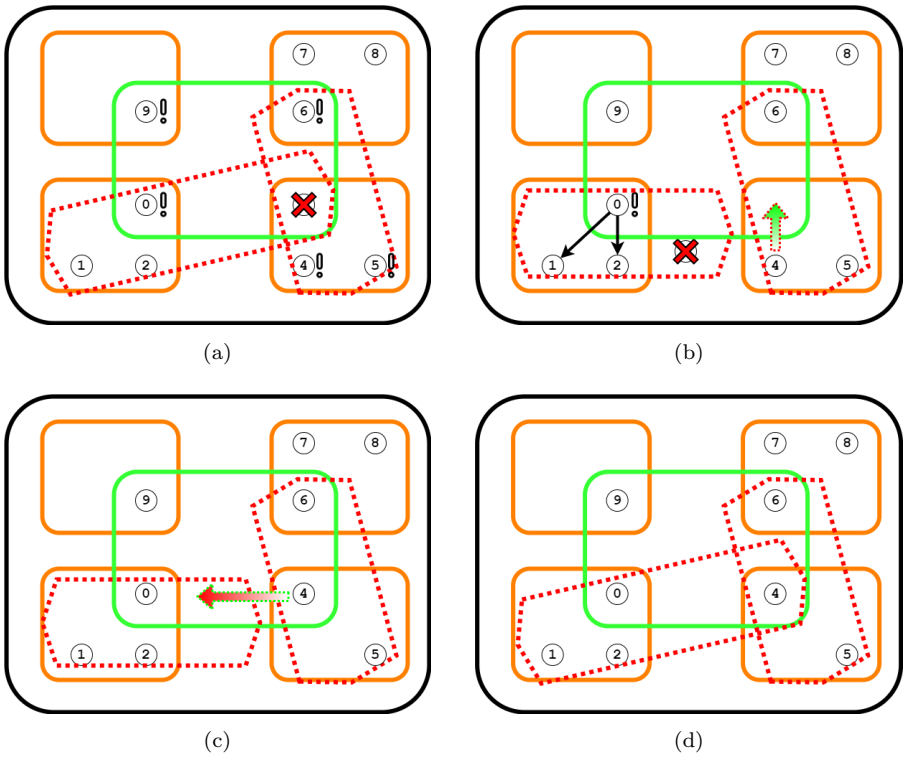

Fig. 3: Overview of the repair procedure when a *master* fails. The communicators and processes follows the notation rules of the previous images. The red cross highlights the failed node. The exclamation marks highlight the nodes that notice the failure. The arrows that originate from a process represent 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.

among the ones in the *local_comm* of the failed *master*, to the *global_comm* (Figure 3b). Then, it can use the *global_comm* to update also the predecessor POV (Figure 3c) and complete the repair procedure (Figure 3d).

Even if this procedure is composed of several steps, it reduces the cost of the repair operations because it lowers its complexity. If we refer to $S(x)$ as the computational cost of the shrinking operation over x processes, we can define the shrink complexity as follows:

$$R_H(s,k) = \begin{cases} S(k) + 2S(k+1) + S(s/k) & \text{if failed master} \\ S(k) & \text{otherwise} \end{cases} \quad (1)$$

where s is the size of the entire communicator and we assume for simplicity that it is multiple of the maximum size of the *local_comm*s k. For the master

fault case, the three terms refer to the shrinking of the *local_comm* ($S(k)$), the two POVs ($2S(k+1)$), and the *global_comm* ($S(s/k)$). The complexity depends on the role of the process, as described previously, and on the value s. When s increases, the complexity of the hierarchical approach improves with respect to $S(s)$, i.e. shrinking the entire communicator. In particular, there might be a minimum value of s such that the hierarchical approach will be less expensive than the normal one (for some value of k). Formally:

$$\exists s_0(\forall s > s_0(\exists k | R_H(s,k) < S(s))) \tag{2}$$

To answer this question we need the complexity of S. Even if we do not have a formal definition, the authors of Fenix [11, 18] have empirically estimated a more than linear complexity. Under the assumption that all the processes have an equal probability of failure, it is possible to continue the analysis by combining the two parts of the Equation 1. In particular, given s as the size of the entire communicator and k as the size of the local communicators, we can state that the probability of a process to be master is $\frac{1}{k}$ (one process per local) and, as a consequence, the probability of being non-master is $\frac{k-1}{k}$. From this, we can obtain:

$$R_H(s,k) = \frac{1}{k}(S(k) + 2S(k+1) + S(\frac{n}{k})) + \frac{k-1}{k}S(k) = \\ = S(k) + \frac{2}{k}S(k+1) + \frac{S(s/k)}{k} \tag{3}$$

Equation 4 and Equation 5 provide the relationship between the communicator size and the value of k that minimizes the overall repair complexity for the linear ($S(x) = cx$) and quadratic ($S(x) = cx^2$) case respectively. The two equations can be obtained by deriving Equation 3 with respect to k, putting the result equal to 0 and by substituting $S(x)$ with the chosen hypothesis. The actual relationship lies between the bound highlighted by the two equations.

$$s = \frac{k(k^2-2)}{2} \tag{4}$$

$$s = \sqrt{\frac{2k^2(2k^2-1)}{3}} \tag{5}$$

Even if we consider the linear case when $s > 11$ the hierarchical approach has a lower complexity. However, the split nature of the network introduces communication overheads since not all the processes are directly connected. This forced us to rethink the way each operation is performed, eventually splitting the execution across the smaller communicators. In particular, we divided the supported operations into various classes, that share the same data movement characteristics:

- **One-to-one** operations are the simplest ones since they involve only two processes. Following property **P.2** and the fact that they do not need the error-checking part, we decided to run them on the entire communicator.

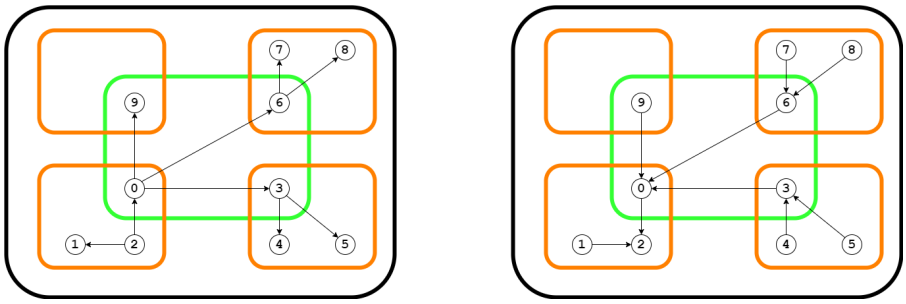

Fig. 4: The propagation steps in one-to-all and all-to-one operations. In both cases, the root process is the one with rank 2.

- **One-to-all** operations (like `MPI_Bcast`) involve all the processes and may cause repair. The data must go from a process to all the others, needing some sort of propagation. To execute the operation, we run it on the different parts in sequence: firstly in the *local_comm* of the root, then in the *global_comm*, and lastly in all the other *local_comm*s in parallel. Figure 4 shows the direction of the information within the network.
- **All-to-one** operations (like `MPI_Reduce`) are similar to one-to-all but the data travels in the opposite direction. We followed the same propagation plan as in one-to-all but in reverse order, as shown in Figure 4.
- **All-to-all** operations (like `MPI_Allreduce`) move data from and to all the processes within the network. We decided to represent them as a combination of an all-to-one and a one-to-all operation executed sequentially.
- **Comm-creator** operations generate new communicators. We cannot execute the operation on a *local_comm* or *global_comm* since there is the need for a unique communicator. These operations are executed on the entire communicator and may cause inefficient repairs. Nonetheless, the trade-off may be acceptable since their frequency is usually lower than the other operations.
- **File operations** do not involve data movement between processes directly: we can use this property to make each process execute the operation on their *local_comm* without the need for any propagation mechanism.
- **Local_only** operations are executed by a process on its structures: no data movement is needed, so it is possible to execute the operation on the *local_comm* as done in file operations.
- **Windows operations** involve data movement and all the windows must be accessible from all the processes within a communicator. These operations are executed on the entire communicator.

The implementation of this solution exposes to the user two knobs: the maximum size of the *local_comm*s and a threshold value for using the hierarchical communicator. Since this solution is an alternative to shrinking the entire communicator, we evaluated both solutions in the experimental campaign.

6 Experimental evaluation

To prove the validity of our solutions, we conducted some experiments using different benchmarks. The purpose of these experiments was to quantify and evaluate the impact of the Legio library usage on various applications. 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. The Legio library has been configured considering the maximum size of the *local_comm*s set to the closest optimal value following the relation obtained with the linear complexity hypothesis (Equation 4).

The experimental campaign aims to evaluate the execution overhead of an application using Legio in a fault-free scenario. This choice has been done since the problem solved by the application after a fault is different because it does not include the part handled by the failed process. Moreover, the survivor processes can complete their execution usually faster than before since there will be one less process competing for the resources. This means that computing the overhead in a faulty environment is not trivial, but can be simplified by considering that the operations performed in presence of a fault are almost the same done before its occurrence, the only difference is the repair procedure.

Our experiments evaluate the temporal overhead of the Legio introduction since the cost in terms of accuracy depends on the application, the problem and the rank of the failed process. 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 [26] 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 and to measure the time needed to repair the execution.

The experiments involving mpiBench were run on a 32 processes network and we analyzed the time needed to complete broadcast and reduce operations under increasing message sizes. The mpiBench application will repeat the calls 1000 times for each message size and for each of the three versions: at first, we linked the initial Legio implementation, then the hierarchical solution, and lastly we just compiled the application with ULFM without additional libraries. Figures 5, 6 and 7 show the average values of the execution times for each call. The overhead can be seen in the difference between the last configuration (an execution without fault management techniques) and the other two. This will also apply to all the tests featuring the "ULFM only" dataset.

It is possible to see how the three values share similar behaviours in terms of growths: this implies that our solutions do not damage the scalability of the MPI library with the increase of the message size.

The experiments involving the ad-hoc code have a different structure: we time each call and we compare it with the same call without the use of any

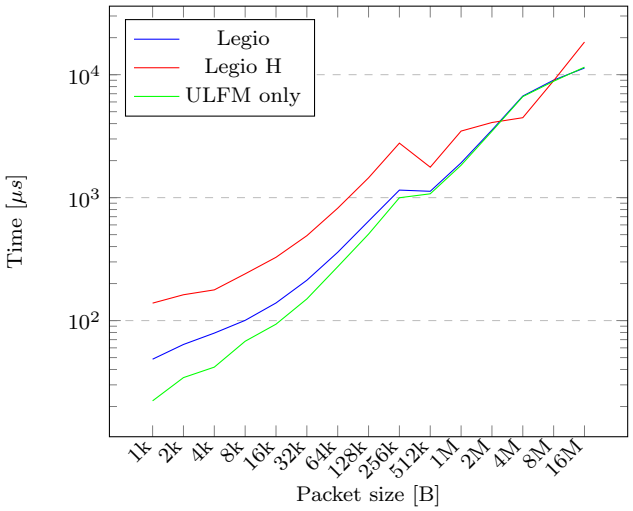

Fig. 5: Execution time to complete a `MPI_Bcast` by varying the message size. Each line represents a different MPI implementation.

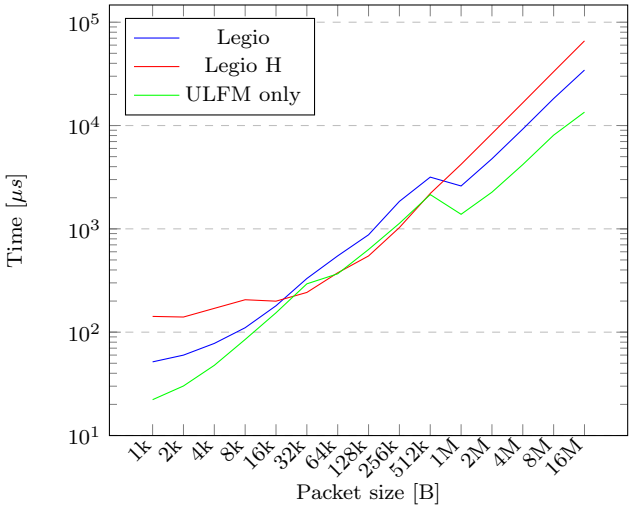

Fig. 6: Execution time to complete a `MPI_Reduce` by varying the message size. Each line represents a different MPI implementation.

Legio feature. The calls have been done using small messages (1 char) to show the overhead in the worst case when the time needed to complete the operation is the lowest. Each call is repeated 100 times, to reduce the impact of measurement noise. Figures 8, 9, and 10 show the results obtained. We also evaluated the cost of the repair procedure by injecting a fault and completing

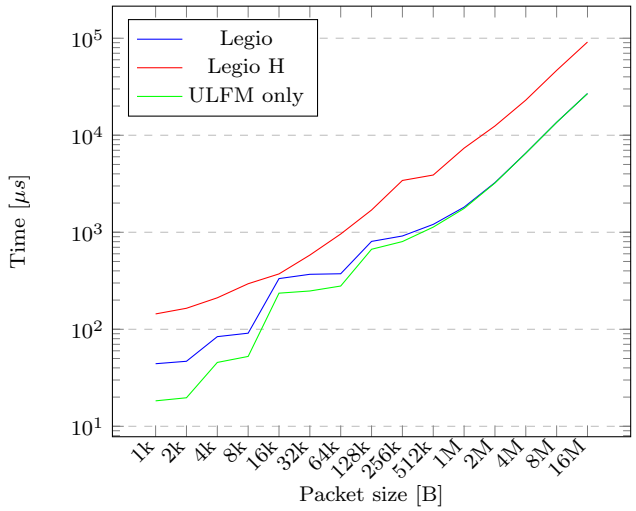

Fig. 7: Execution time to complete a `MPI_Allreduce` by varying the message size. Each line represents a different MPI implementation.

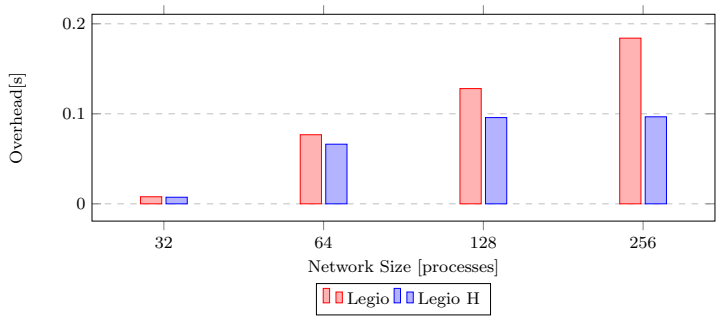

Fig. 8: `MPI_Bcast` overhead by varying the network size. Each measure accumulate 100 repetitions of the operation.

an operation. Figure 11 shows the results of this latter analysis: from that, it's possible to see that the non-linearity of the shrink theorized by [18] is not present in our tests. Despite this fact, the average time to repair on a 256 core machine is lower in the hierarchical case, since the probability for a master node to fail is contained (1/8). Using the ad-hoc code we checked also the overhead for file operations: running those tests in the same configurations as the previous ones, we noticed that the execution time of a single call is heavily influenced by the load of the file-system rather than from other aspects. The overhead measured was affected by the load too and, despite being contained, it cannot be considered meaningful.

The second group contains experiments run on two embarrassingly parallel applications. The first application is part of the NAS parallel bench-

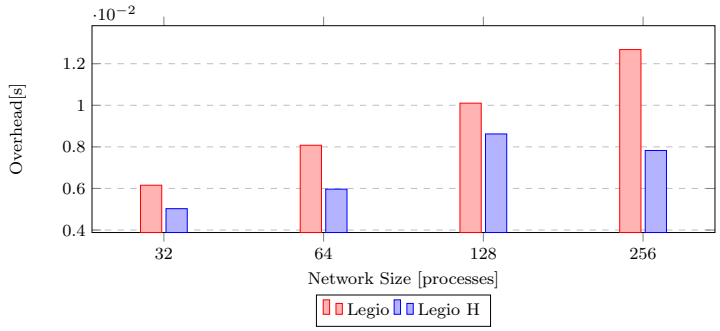

Fig. 9: `MPI_Reduce` overhead by varying the network size. Each measure accumulate 100 repetitions of the operation.

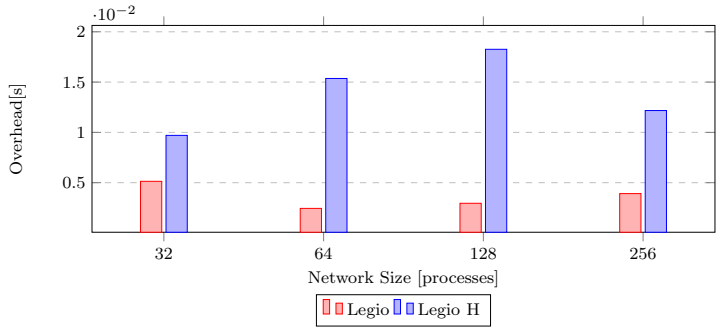

Fig. 10: `MPI_Barrier` overhead by varying the network size. Each measure accumulate 100 repetitions of the operation.

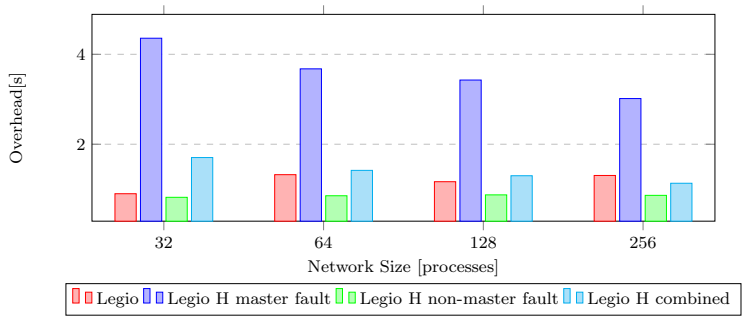

Fig. 11: Communicator repair time by varying the number of processes involved in the operation. The combined value summarizes both the values of the Hierarchical approach by assuming equal probability of faults across all nodes.

mark [27] and it generates independent Gaussian random variates using the Marsaglia polar method. The MPI calls performed by the applications are

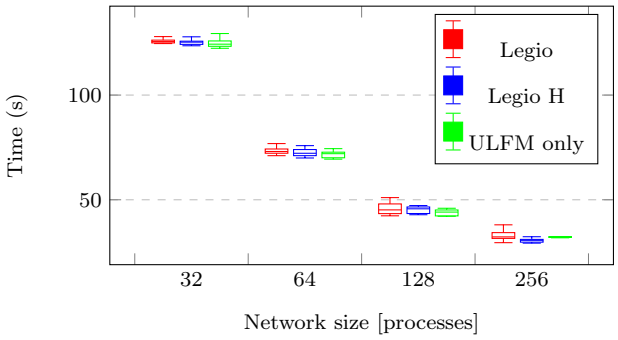

Fig. 12: Execution time distribution of the EP benchmark by varying the number of processes involved and the MPI implementation.

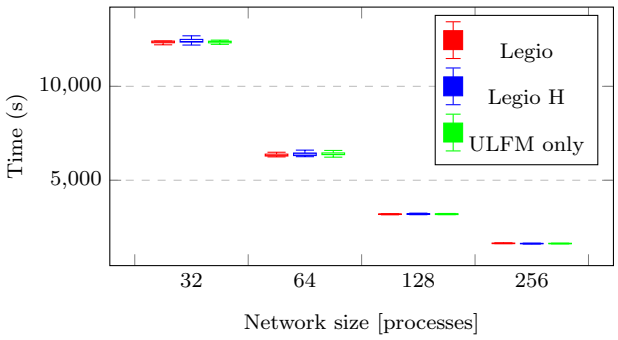

Fig. 13: Execution time distribution of the molecular docking application by varying the number of processes involved and the MPI implementation.

mainly `MPI_Allreduce` operations. We use the "C" size workload and the measurements refer to the successive execution of 7 runs. We ran the tests in various configurations in terms of the number of MPI processes and MPI implementation. In particular, we use 32, 64, 128, and 256 processes and choose between one of our implementations or only ULFM. For each configuration, we repeat the experiment 10 times, extracting all the execution times. The results can be seen in Figure 12.

The second experiment uses the skeleton of a molecular docking application, which estimates the strength of the interaction between two molecules. In this context, we have a target molecule and a database of smaller molecules that we need to evaluate to find the most promising ones. The application uses a wide range of MPI calls, from file operations to point-to-point and collective functions. As a workload, we use a database with 113K molecules. We ran the executions using the same configurations used in the previous experiment, repeating each one 10 times and extracting the execution times. The results can be seen in Figure 13.

From the experimental results, it is easy to notice how the overhead is negligible and the usage of Legio does not impact heavily the execution times in both cases. This is also related to the reduced use of communication in embarrassingly parallel applications. Nevertheless, those experiments validate our approach since they respect the requirements of low overhead introduction and transparency. Moreover, both the prototypes proved effective for the embarrassingly parallel applications tested and can continue the execution in presence of faults in a manner of seconds. Despite the plots being limited to 256 processes, we do not expect that the trends shown in figures 12 and 13 to change significantly, except for an increment of the overhead following the trend shown in figures 8, 9 and 10.

In both cases, the functional effect of a fault leads to an accuracy loss in the result computation. In particular, the accuracy loss can be estimated a-priori given the uniform data split across the n processes: if the system suffers from faults of f processes, the application will base its evaluation only on $\frac{n-f}{n}$ times the problem data. For the EP benchmark, this fraction represents the number of processes contributing to the final reduction, while in the molecular docking application it represents the lower bound of the overall molecules that will be screened.

7 Ongoing work on introducing the C/R feature

Not all embarrassing parallel applications relies on the fact that they can produce useful results even in presence of a failed process. In these cases, the current version of Legio cannot be employed. However, we already evaluated the possibility to introduce a C/R feature in the library thus obtaining the possibility to recover failed processes transparently.

As discussed in Section 2, many other efforts combined C/R frameworks with ULFM [11–13,17,18]. Most of them do not focus on transparency, asking for explicit application-level intrusiveness. Since Legio has in transparency one of its key features, we moved towards system-Level C/R frameworks that guarantee a transparent approach at the cost of a large overhead for both check-pointing (all the system status has to be saved) and restart phase.

Considering the type of application we are targeting, the characteristic that we want to take out from a C/R framework is not to restart the entire application, but only the failed processes. The possibility to restart only a part of the network is not a common feature in system-level C/R frameworks. Usually, these frameworks are designed to consider the absence of fault mitigation mechanisms inside the application, so they assume that in case of fault all the processes must be restarted. Moreover, they tend not to split the checkpoint information of the various processes because they would lose significance without all the others. The restart part may also lead to problems: without knowing the details about the application, it may be difficult to load a system-level checkpoint on a process created by the application.

Among all the efforts produced in literature, recently we found in MANA [28] support in that direction. It provides system-level checkpointing (no application intrusiveness), the possibility to migrate processes (implying the division of the data per process), and flexibility on MPI versions upon restart. Our idea is to exploit the per-process data checkpointing offered by MANA to restore only the failed process. While everything seems ready for integration, MANA is still designed for global recovery and the steps towards local recovery are part of our ongoing work.

8 Conclusion

This paper 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 ULFM and the PMPI interface to wrap the MPI call and to implement all the required actions to manage failed processes. In the paper, an extension towards a hierarchical implementation has been also presented to reduce the overhead of the repair process in case of a large number of nodes involved. 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.

Declarations

- No funding was received for conducting this study.
- The authors have no conflicts of interest to declare that are relevant to the content of this article.
- All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

References

1. J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. 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*, vol. 23, no. 4, pp. 309–322, 2009.
2. S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill *et al.*, "Exascale software study: Software challenges in extreme scale systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, pp. 1–153, 2009.
3. G. Zheng, X. Ni, and L. V. Kalé, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. IEEE, 2012, pp. 1–6.

4. J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, "The international exascale software project roadmap," *The international journal of high performance computing applications*, vol. 25, no. 1, pp. 3–60, 2011.
5. L. Clarke, I. Glendinning, and R. Hempel, "The mpi message passing interface standard," in *Programming environments for massively parallel distributed systems*. Springer, 1994, pp. 213–218.
6. M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.
7. A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "Mpich-v project: A multiprotocol automatic fault-tolerant mpi," *The International Journal of High Performance Computing Applications*, vol. 20, no. 3, pp. 319–333, 2006.
8. K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, and R. Brightwell, "rmpi: increasing fault resiliency in a message-passing environment," *Sandia National Laboratories, Albuquerque, NM, Tech. Rep. SAND2011-2488*, 2011.
9. G. E. Fagg and J. J. Dongarra, "Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2000, pp. 346–353.
10. W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of mpi communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.
11. M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. 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*. IEEE, 2014, pp. 895–906.
12. N. Losada, I. Cores, M. J. Martín, and P. González, "Resilient mpi applications using an application-level checkpointing framework and ulfm," *The Journal of Supercomputing*, vol. 73, no. 1, pp. 100–113, 2017.
13. K. Teranishi and M. A. Heroux, "Toward local failure local recovery resilience model using mpi-ulfm," in *Proceedings of the 21st european mpi users' group meeting*, 2014, pp. 51–56.
14. S. Pauli, P. Arbenz, and C. Schwab, "Intrinsic fault tolerance of multilevel monte carlo methods," *Journal of Parallel and Distributed Computing*, vol. 84, pp. 24–36, 2015.
15. "Exscalate4cov - exascale smart platform against pathogens." [Online]. Available: http://www.exscalate4cov.eu/
16. "Marconi100, the new accelerated system." [Online]. Available: https://www.hpc.cineca.it/hardware/marconi100
17. F. Shahzad, J. Thies, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein, "Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 501–514, 2018.
18. M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. 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*. IEEE, 2015, pp. 1–12.
19. N. Losada, G. Bosilca, A. Bouteiller, P. González, and M. J. Martín, "Local rollback for resilient mpi applications with application-level checkpointing and message logging," *Future Generation Computer Systems*, vol. 91, pp. 450–464, 2019.
20. P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," *Acm sigplan notices*, vol. 47, no. 8, pp. 225–234, 2012.
21. U. Kalim, M. K. Gardner, and W. Feng, "A non-invasive approach for realizing resilience in mpi," in *Proceedings of the 2017 Workshop on Fault-Tolerance for HPC at Extreme Scale*, 2017, pp. 1–8.
22. P. E. Strazdins, M. M. Ali, and B. Debusschere, "Application fault tolerance for shrinking resources via the sparse grid combination technique," in *2016 IEEE International*

Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2016, pp. 1232–1238.
23. F. Rizzi, K. Morris, K. Sargsyan, P. Mycek, C. Safta, B. Debusschere, O. LeMaitre, and O. Knio, "Ulfm-mpi implementation of a resilient task-based partial differential equations preconditioner," in *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, 2016, pp. 19–26.
24. N. Losada, P. González, M. J. Martín, G. Bosilca, A. Bouteiller, and K. Teranishi, "Fault tolerance of mpi applications in exascale systems: The ulfm solution," *Future Generation Computer Systems*, vol. 106, pp. 467–481, 2020.
25. R. Thakur and W. Gropp, "Open issues in mpi implementation," in *Asia-Pacific Conference on Advances in Computer Systems Architecture*. Springer, 2007, pp. 327–338.
26. "mpibench: Mpi benchmark to test and measure collective performance." [Online]. Available: https://github.com/LLNL/mpiBench
27. D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.
28. R. Garg, G. Price, and G. Cooperman, "Mana for mpi: Mpi-agnostic network-agnostic transparent checkpointing," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, 2019, pp. 49–60.