Rollback-Free Recovery for a High Performance Dense Linear Solver With Reduced Memory Footprint

Daniela Loreti, Marcello Artioli, and Anna Ciampolini

***Abstract*—The scale of nowadays High Performance Computing (HPC) systems is the key element that determines the achievement of impressive performance, as well as the reason for their relatively limited reliability. Over the last decade, specific areas of the High Performance Computing (HPC) research field have addressed the issue at different levels, by enriching the infrastructure, the platforms, or the algorithms with fault tolerance features. In this work, we focus on the rather-pervasive task of computing the solution of a dense, unstructured linear system and we propose an algorithm-based technique to obtain fault tolerance to multiple anywhere-located faults during the parallel computation. We particularly study the ways to boost the performance of the rollback-free recovery, and we provide an extensive evaluation of our technique w.r.t. to other state-of-the-art algorithm-based methods.**

***Index Terms*—Rollback-free recovery, algorithm-Based fault tolerance, high performance computing, linear systems solver.**

I. Introduction

OVER the last decade, the computing capabilities of High Performance Computing (HPC) systems have impressively increased, heading beyond the Exaflops realm [1]. Unfortunately, their availability and reliability have not escalated at the same pace. The reason behind this is directly connected to the system's scale and the chip density that characterises HPC hardware [2]. Nowadays, despite an overall increase in chip's Mean Time Between Failures (MTBF), the availability of a relevant number of computing nodes frequently translates not only into the desirable higher degree of parallelism (which should improve the application's performance) but also into a significant drop of the infrastructure's MTBF [3], [4] (which in the end affects the application's capability to provide correct results). Clearly, this issue is particularly relevant for those applications that are significantly time-consuming despite being parallelly executed on many cores.

An example of such applications is the resolution of linear systems characterised by large-scale dense matrices: a task required in many scientific fields, from circuit simulation to aerodynamic design. Beside some system- and application-level solutions to improve the overall resilience of HPC systems [5], [6], [7], [8], [9], fault-tolerant algorithms have been developed over the last years, to precisely address matrix factorisation or linear algebra applications in general. These algorithm-specific approaches to resilience are often classified into the broad category of Algorithm Based Fault Tolerance (ABFT) [10].

Whether it is applied at the algorithm, application, or system level, the efficiency of a fault tolerance strategy is usually connected to two factors: the overhead introduced during computation *before* the failure occurs—e.g., to maintain an encoded replica of the computation's state; also known as *computation* or *failure-free overhead*—and the computational cost due to the recovery mechanism *after* the failure occurs—e.g. to reconstruct the state of the fallen processors and restart the computation; often referred to as *recovery overhead*.

A traditional, and still widely employed technique for fault-tolerance at the system/application level is Checkpoint/Restart (C/R), which envisages periodically saving the calculation state on a persistent or volatile storage system [11]. In case of error, before the application can continue, the faulty unit must be replaced and the state rolled back to the latest available checkpoint. All these operations entail potentially significant failure-free and recovery overheads. Albeit characterised by lower portability w.r.t the C/R approach, ABFT techniques usually show the key advantage of significantly reducing both the computation and the recovery overheads.

In this work, we focus on an algorithm-level strategy to obtain fault tolerance to multiple concurrent faults during the resolution of linear systems on HPC hardware. In the context of linear solvers for dense unstructured matrices, previous works [12], [13] have studied the limited fault-free overhead that can be obtained by applying ABFT to an existing direct linear systems solver, called Inhibition Method (IMe) [14], [15]. However, the approach shows poor performance in terms of memory occupation and disregards the impact of recovery overhead on the computation. The present paper addresses the limitations of the previous works by exploiting the structure of the IMe solver to provide a rollback-free technique that significantly

Manuscript received 25 September 2023; revised 16 March 2024; accepted 8 May 2024. Date of publication 13 May 2024; date of current version 31 May 2024. This work was supported by Daniela Loreti with a research contract co-financed by the European Union - PON Ricerca e Innovazione 2014-2020 ai sensi dell'art. 24, comma 3, lett. a), della Legge 30 dicembre 2010, n. 240 e s.m.i. e del D.M. 10 agosto 2021 n. 1062. Recommended for acceptance by M. Si. *(Corresponding author: Daniela Loreti.)*

Daniela Loreti and Anna Ciampolini are with the Department of Computer Science and Engineering, University of Bologna, 40126 Bologna, Italy (e-mail: daniela.loreti@unibo.it; anna.ciampolini@unibo.it).

Marcello Artioli is with the Italian National Agency for New Technologies, Energy and Sustainable Economic Development (ENEA), 40129 Bologna, Italy (e-mail: marcello.artioli@enea.it).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2024.3400365, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2024.3400365

limits memory occupation. At the same time, the enhancement improves the performance of the technique w.r.t state-of-the-art fault tolerant solvers.

II. Contribution

This work provides the following contributions.

- A rollback-free approach to fault tolerance for the resolution of linear systems with dense unstructured matrices. The technique allows recovery from multiple faults with reduced memory occupation.
- A demonstration of how the method can be used to solve multiple systems with the same matrix of coefficients but different constant terms (analogously to, e.g., LU factorization).
- A flexible parallel implementation of the above-mentioned approach, which enhances the performance by limiting the number of synchronisation points due to communication.
- An analysis of the advantages and shortcomings of centralised and distributed implementation strategies that can be adopted for the recovery process.
- An empirical evaluation of the proposed solution, comparing its overall performance (including scalability, memory occupation, and accuracy of the recovery task) with other, well-known fault-tolerant methods for dense linear system resolution.

III. Related Work

In the last decade, the decline of MTBF in HPC systems has fostered the study of novel solutions to identify and handle malfunctions affecting large-scale parallel computations. In literature, a widely adopted classification distinguishes the failures based on their effects: *hard errors* or *fail-stop* address malfunctions that interrupt the processor's computation; whereas *soft errors*, *fail-continue* or Silent Data Corruption (SDC) address those errors that do not halt any computing node but nonetheless induce alterations in the results. Known causes of soft errors are thermal drift or radiation [16]. Both hard and soft faults have the same power to invalidate the whole distributed computation but, as their origins and effects are different, they are usually addressed with different techniques. The approach described in this work is primarily devoted to recovery from hard errors.

C/R is the traditional way to tackle fail-stop errors in large clusters [17]: the computation flow is periodically interrupted to save the state of long-running programs on reliable storage. In the event of an error, the whole application state is rolled back to the most recent copy of the state, while the state of faulty processors is sent to newly provided computing units. In this way, the application can survive any number of faults because it can continue from the latest saved checkpoint instead of being started over.

A fault-tolerant mechanism can be implemented at system-level, by modifying the OS kernel or the hardware [6], [18], [19]; user-level, by linking the program to fault-tolerant libraries [20], [21], [22], [23]; or application-level, by injecting the resilient code directly into the application (i.e., relaying on the programmer's domain knowledge or by means of a pre-processor) [8], [9]. Despite some recent relevant attempts [24] to combine system- and user-level checkpointing to minimise the failure overhead, I/O bottleneck remains the main concern of C/R techniques. Diskless checkpointing [11] (and its following enhancements [25], [26]) helped to contain this problem, by encoding and saving the state of the computation into the internal memory of redundant computing nodes, instead of reliable storage. This solution provides a considerable reduction of checkpoint overhead but comes at the price of the application no longer being able to survive a whole-system failure. Furthermore, when the program iteratively modifies large memory regions (as in matrix factorisation), diskless checkpointing might still exhibit relevant overheads [27]. Obviously, the naive solution of increasing the checkpoint interval has the drawback of a longer recovery overhead (on average), because the system will probably need to roll back and restart the computation from an older state when the fault occurs. To overcome these issues, an algorithm-specific solution can be applied.

ABFT was first introduced by Huang et al. [10] to tackle the problem of soft errors. This approach is based on the observation that in various matrix operations, the checksum relationship can be kept invariant during the course of the algorithm. Therefore, a miscalculation can be detected by verifying if the final results still maintain the checksum relationship. ABFT has been later extended [28], [29], [30] and applied also to hard errors [31], [32].

The advantage of some ABFT techniques w.r.t a traditional—disk or diskless—checkpointing is twofold. On one hand, the failure-free overhead is reduced because there is often no need to stop the computation to encode and save the state: the checksum is calculated once at the beginning, and then maintained by applying some operations as it is done on the data. On the other hand, these ABFT approaches do not need to perform rollback when the fault occurs because the state is always up-to-date.

The works [27], [33], [34], [35], [36], [37], [38], [39], [40] proved that a checksum invariant can be maintained during matrix-matrix multiplication, and various linear system resolution methods. In particular, Davies et al. [33] focus on High Performance Linpack (HPL) and demonstrate that the right-looking LU factorisation can be performed while preserving a checksum at each step of the computation. Such a strategy protects the U matrix from the occurrence of a single anywhere-located fault, whereas the L matrix is not protected. In [36], the authors propose a dynamic fault-tolerant mechanism for Communication-Avoiding LU (CALU) grounded on redundancy: at the beginning, the algorithm cannot tolerate any fault; then at each step, an exchange of data between the computing nodes increases the number of tolerable faults. Kang et al. [39], [40] propose an ABFT linear system solver for sparse matrices based on the Conjugate Gradient method, which tolerates up to k faults by augmenting the input matrix with a suitable set of coded rows and columns. To contain the—otherwise relevant—computation overhead, they use a sparse coding scheme so that the algorithm is no longer able to recover from *any possible* k faults, but tolerates the faults with a given probability. Bouteiller et al. [27] extended a previous work [35] to deal with multiple faults. They propose a general ABFT framework for matrix factorisation, which can be applied not only to HPL and Cholesky but also to

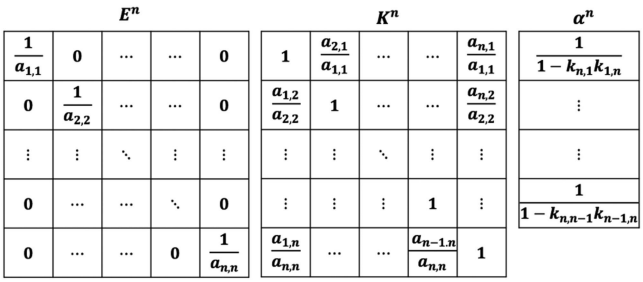

Fig. 1. Initialization of IMe data structures.

LU and QR. However, the protection of the left factor is achieved through a C/R scheme.

Differently from all the cited works, the technique described in this work is devoted to the recovery from multiple anywhere-located faults during a different solving approach, grounded on IMe [14], [15]. This strategy enables a pure algorithm-based approach to fault tolerance that avoids any C/R and greatly simplifies the management of the checksums by exploiting the structure of IMe itself. Other remarkable advantages are the reduced memory overhead to maintain the checksum and the intrinsically load-balanced parallel implementation.

Since IMe is scarcely referred to in literature, we dedicate the following section to the description of its original formulation devoted to the linear system resolution.

IV. Background

IMe was initially proposed in 1963 [41] to perform the analysis of complex electric circuits and later extended to linear system resolution and matrix inversion [14], [15]. The method envisages an exact direct solver for systems in the form $\boldsymbol{Ax} = \boldsymbol{b}$, where the matrix $\boldsymbol{A}$ of coefficients is a squared $n \times n$, dense, unstructured, and invertible matrix, and $\boldsymbol{b}$ is the vector of constant terms.

The original formulation starts by computing two $n \times n$ matrices $\boldsymbol{E}$ (the matrix of the *effects*) and $\boldsymbol{K}$ (the matrix of the *inhibition quantities*), and an auxiliary vector $\boldsymbol{\alpha}$ (with $n-1$ elements) out of the initial matrix $\boldsymbol{A}$, as shown in Fig. 1.

$\boldsymbol{E}$, $\boldsymbol{K}$ and $\boldsymbol{\alpha}$ are modified in n subsequent iterations, which were originally called *levels* and named by decreasing order (so that level $l = n$ actually addresses the first iteration). At each level $l = n \ldots 2$ all the elements of the two matrices are modified according to the following *fundamental formula*.

$$\begin{aligned} e_{ij}^{(l-1)} &= \left[e_{ij}^{(l)} - e_{lj}^{(l)} \cdot k_{il}^{(l)}\right] \cdot \alpha_i^{(l)}, \quad i = 1 \ldots l-1, j = 1 \ldots n \\ k_{ij}^{(l-1)} &= \left[k_{ij}^{(l)} - k_{lj}^{(l)} \cdot k_{il}^{(l)}\right] \cdot \alpha_i^{(l)}, \qquad\qquad i, j = 1 \ldots l-1 \end{aligned} \tag{1}$$

where $e_{ij}^{(l)}$ and $k_{ij}^{(l)}$ denote the elements on row i and column j at level l of the matrices $\boldsymbol{E}$ and $\boldsymbol{K}$, respectively. The vector $\boldsymbol{\alpha}$ too is recomputed at each level as follows.

$$\alpha_i^{(l)} = \frac{1}{1 - k_{li}^{(l)} \cdot k_{il}^{(l)}}, \forall l = n \ldots 2,\ i = 1 \ldots l-1. \tag{2}$$

As shown in Fig. 2, the algorithm reduces $\boldsymbol{E}$ by a row at each level, whereas $\boldsymbol{K}$ loses both a row and a column. This process

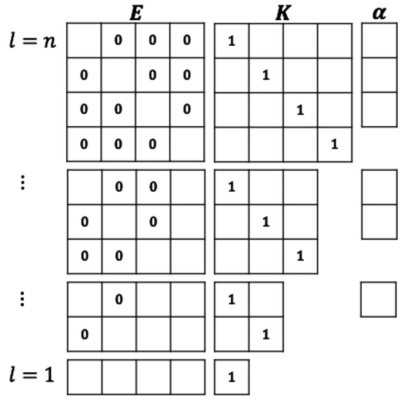

Fig. 2. Evolution of IMe data structures.

continues until $\boldsymbol{E}$ is reduced to a row vector of n elements and $\boldsymbol{K}$ contains only an element with value 1. The original method prescribes using the matrices obtained from each level to modify the constant terms vector $\boldsymbol{b}$ and, at the same time, compute the solution $\boldsymbol{x}$. That is, for all levels l

$$b_i^{(l-1)} = b_i^{(l)} - k_{l-1,i}^{(l-1)} \cdot b_l^{(l)}, \qquad i = 1 \ldots l-2 \tag{3}$$

$$x_i^{(l-1)} = x_i^{(l)} + e_{l-1,i}^{(l-1)} \cdot b_l^{(l)}, \qquad i = l-1 \ldots n \tag{4}$$

Thanks to its structure, IMe is particularly suitable for a parallel implementation.

A previous work [12] proposed IMeFT, an HPC implementation of IMe, which employed a 1D-block distribution along the matrix columns. The paper demonstrated that by computing row-wise checksums of $[\boldsymbol{E}|\boldsymbol{K}]$, and then applying the fundamental formula (1) on them as on any other element of $[\boldsymbol{E}|\boldsymbol{K}]$, the additional columns continue to hold the checksums for any following level. In other words, the checksum relation is kept invariant during the whole computation. Based on this observation, the work [12] concluded that it is possible to obtain fault tolerance from a single, anywhere-located, hard error at the price of a small computation overhead: an additional computing processor must be employed during the whole algorithm execution.

In this work, we adopt the definition of *failure period* of Bouteillers et al. [27] i.e., the time between the moment when the failure is detected, and the moment when the data have been completely recovered, so that the normal computation is ready to continue. Theoretically, the strategy of IMeFT [12], like the one of Bouteillers et al. [27], allows surviving any number of faults, as long as their failure periods do not overlap. In other words, if two faults occur during the same computation, the second one can be recovered only if it shows up after the first one has been recovered.

However, it is important to consider that this circumstance is not frequent in nowadays HPC systems: as they are composed of (several) multiprocessor machines, a hard fault is unlikely to involve a single core at a time. More probably, it will regard a processor socket with multiple cores or the whole machine, thus making more pressing the need for algorithms that can survive multiple faults within the same failure period. For this reason, the work [13] demonstrated that—by adopting a simple column-wise parallelisation and using a weighted sum instead of a simple

sum—IMeFT can be extended to tackle also multiple hard errors i.e., faults involving up to a certain number of anywhere-located processors, and occurring during the same failure period.

The advantage of the fault-tolerant approaches described in [12], [13] w.r.t a traditional Gaussian solver with diskless checkpointing is in the reduced failure-free overhead, whereas the memory occupation remains an issue: the IMe-based approach can work in-place at every level by overwriting the elements of $[\boldsymbol{E}|\boldsymbol{K}]$ for all l, but it employs a matrix $[\boldsymbol{E}|\boldsymbol{K}]$ that, initially, is twice the size of $\boldsymbol{A}$. Furthermore, a more flexible parallelisation scheme (i.e., not limited to a strict column-wise distribution) is needed to provide a method that can be successfully applied in practice in an HPC environment. Finally, [12], [13] do not deepen the topic of the recovery overhead: the performance of the recovery mechanisms needs to be investigated in order to compare the technique with existing state-of-the-art fault-tolerant methods for linear system resolution.

In this study, we address each of these issues: we start with the reduction of the memory occupation described in the following section; the column-wise distribution is enhanced to a 2D-block cyclic distribution as presented in Section V-A; and a new distributed recovery strategy is described in Section VII.

V. Enhancing Fault-Tolerant IMe With Reduced Memory Footage

Looking at the effects, depicted in Fig. 2, produced by the n subsequent applications of the fundamental formula, it is easy to see that the original algorithm maintains a considerable number of 1 and 0 entries at each level. Most of these elements keep their 0 or 1 value during the whole computation despite the algorithm applying the fundamental formula on them as on any other element of $[\boldsymbol{E}|\boldsymbol{K}]$. As we want to reduce the memory occupation of the algorithm, a natural enhancement is therefore to eliminate these 1 and 0 entries and overwrite them with more meaningful values. To this end, we must first assess and demonstrate that the position of 1 and 0 values is indeed the one illustrated in Fig. 2 throughout the computation. Then we can use this information to propose a compression of $[\boldsymbol{E}|\boldsymbol{K}]$.

Theorem V.1. At any level $l = n \dots 1$ the matrix $[\boldsymbol{E}|\boldsymbol{K}]^{(l)}$ contains l ones (on the diagonal of $\boldsymbol{K}^{(l)}$) and $l^2 - l$ zeros (on all non-diagonal elements of $\boldsymbol{E}^{(l)}$).

The proof of Theorem 5.1 can be found in the Appendix. We can therefore overwrite the 0 and 1 entries when compressing $\boldsymbol{E}^{(l)}$ and $\boldsymbol{K}^{(l)}$ into a single compact matrix $\boldsymbol{V}^{(l)}$ as shown in Fig. 3. At the initialisation step, $\boldsymbol{V}^{(n)}$ is actually the original matrix $\boldsymbol{K}^{(n)}$ with the 1-entries overwritten by the diagonal elements of $\boldsymbol{E}^{(n)}$. Then, for each level, the rows of $\boldsymbol{V}^{(l)}$ have the same content of $\boldsymbol{E}^{(l)}$, except for the 0-entries in the first l columns, which are overwritten by the corresponding elements of $\boldsymbol{K}^{(l)}$. Thanks to this strategy the memory occupation is reduced by a half.

According to this change, we initialise each $v_{ij}^{(n)}$ element of $\boldsymbol{V}^{(n)}$ as follows.

$$v_{ij}^{(n)} = \begin{cases} \frac{1}{a_{ij}}, & \text{if } j = i \\ \frac{a_{ji}}{a_{ii}}, & \text{otherwise} \end{cases}$$

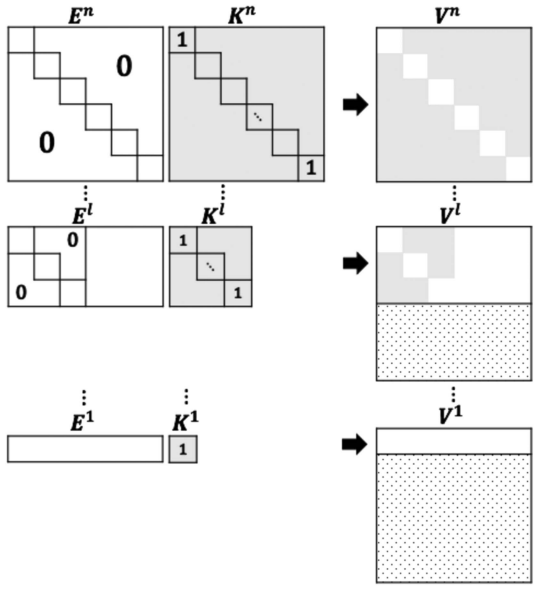

Fig. 3. Evolution of IMe data structures when compressing $\boldsymbol{E}$ and $\boldsymbol{K}$ into a single matrix $\boldsymbol{V}$. Here, $\boldsymbol{V}$ is reused at each level. The dotted portions correspond to cells that are not modified at that level.

with $i, j = 1 \dots n$. The fundamental formula becomes:

$$v_{ij}^{(l-1)} = \alpha_i^{(l)} \cdot \begin{cases} v_{ii}^{(l)}, & \text{if } j = i \\ -v_{il}^{(l)} v_{ll}^{(l)}, & \text{if } j = l \\ v_{ij}^{(l)} - v_{il}^{(l)} v_{lj}^{(l)}, & \text{otherwise} \end{cases} \tag{5}$$

with $l = n \dots 2, i = 1 \dots l-1$, and $j = 1 \dots n$. To compute the solution we apply the following for all levels $l = n \dots 2$:

$$b_i^{(l-1)} = b_i^{(l)} - v_{l-1,i}^{(l-1)} \cdot b_l^{(l)}, \qquad i = 1 \dots l-2 \tag{6}$$

$$x_i^{(l-1)} = x_i^{(l)} + v_{l-1,i}^{(l-1)} \cdot b_l^{(l)}, \qquad i = l-1 \dots n \tag{7}$$

The matrix $\boldsymbol{V}$ obtained with the described compression strategy shows a derirable characteristic:

Theorem V.2. Given a linear system $\boldsymbol{Ax} = \boldsymbol{b}$, the iterative application of (5) produces a matrix $\boldsymbol{V}^{(1)}$ that can be used not only to compute $\boldsymbol{x}$, but also $\boldsymbol{x'}$ solution of $\boldsymbol{Ax'} = \boldsymbol{b'}$, i.e., the solution of any other linear system characterised by the same matrix $\boldsymbol{A}$ but a different constant term $\boldsymbol{b'}$.

In other words, Theorem 5.2 (proof in the Appendix) ensures that, similarly to the LU factorisation, once the matrix $\boldsymbol{A}$ has undergone all the n levels producing $\boldsymbol{V}^{(1)}$, the latter can be used to solve systems with different right-hand sides. More precisely, since (6) and (7) entail two floating point operations (flops) for $i = 1 \dots n$ in any of the n levels, the computation of the solution requires $2n^2$ flops (the same required to compute the solution from a LU-factorised matrix). The theoretical cost of computing $\boldsymbol{V}^{(1)}$, instead, is $O(\frac{3}{2}n^3)$ [12], slightly higher than LU factorisation: $O(\frac{2}{3}n^3)$.

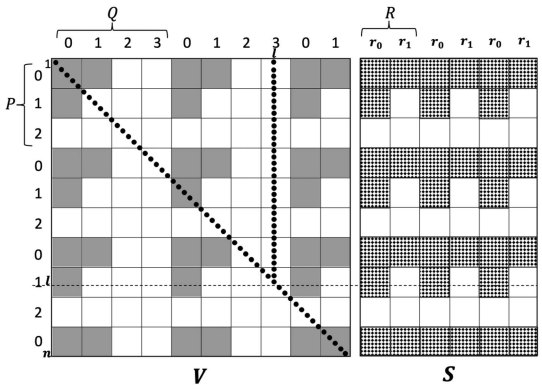

Fig. 4. 2D block-cyclic distribution of V and S on a 3×4 and 3×2 processor grid, respectively. Generally, with this scheme, the algorithm can recover from up to $R = 2$ anywhere-located faults. In this case, even if the fault involves three processors, (0, 0)(0, 1) and (1, 0) creating the grey holes in V, they can be recovered anyway using the dotted blocks on the checksum processors (0, r_0)(0, r_1) and (1, r_0). This is possible because the third faulty processor (1,0) is not on the same processor row as the others.

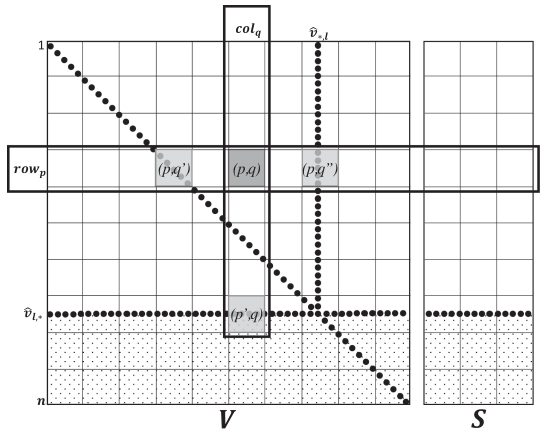

Fig. 5. The blocks involved in the update of a specific block on the processor (p, q) are those illustrated in light gray beloging to processors (p, q'), (p, q'') and (p', q).

A. Checksum Protection Against Multiple Failures

The use of the compact matrix $\boldsymbol{V}$ turns IMe into an algorithm that can work in-place with the same memory occupation as LU factorisation. It is also crucial to demonstrate that these changes do not affect the checksum invariant. To this aim, we first need to consider how IMe can be parallelly executed on a HPC environment with distributed memory.

The previous work [13] assumed a contiguous allocation of the matrix columns on the computing processors. In order to obtain a flexible algorithm (i.e., an algorithm that can be applied to any number of interconnected nodes preserving the load balancing feature during the execution), we hereby introduce a 2D-block cyclic distribution of the initial matrix $\boldsymbol{V}^{(n)}$. According to 2D-block cyclic distribution, $\boldsymbol{V}^{(n)}$ is divided into $N \times N$ blocks of data, each one of size $n_b \times n_b$. The blocks are distributed along a process grid of $P \times Q$ processors using a cartesian virtual topology. Therefore, each processor receives at most $K = \lceil \frac{N}{P} \rceil \times \lceil \frac{N}{Q} \rceil$ blocks. Similarly to the notation of [27], we denote with $V^{(l)}_{k,pq}$ the kth block of columns held by processor (p, q), with $0 \leq p < P$, $0 \leq q < Q$, and $1 \leq k \leq K$. Also, we denote with $\mathcal{Q}$ the set of the indexes $\{0, \ldots, Q-1\}$ on a row of the processor grid.

A 2D-block cyclic layout brings several advantages w.r.t a simple column distribution. It helps to balance the computational load because, in each step of the algorithm, many computing nodes can be concurrently engaged in computations. Also, as we show in the following, this choice contributes to reducing the synchronisation points due to data communication because broadcast messages are limited to processor rows and columns instead of involving all computing nodes.

Assuming a 2D-block cyclic distribution, if we want to tolerate up to R anywhere-located faults, we need $P \times R$ additional processors to store the checksum matrix $\boldsymbol{S}^{(n)}$. We will call them *recovery processors*. Actually, as shown in Fig. 4, $P \times R$ recovery processors allow tolerating a number of faults that can be even higher than R (up to $P \cdot R$), provided that we have at most R faults on the same processor row. We denote with $\mathcal{R}$ the set of the indexes $\{0, \ldots, R-1\}$ on a row of the recovery processor grid.

Furthermore, to be consistent with the original algorithm [13], the row-wise weighted sums computed during the initialisation must be performed on the matrix $\tilde{\boldsymbol{V}}^{(n)} = \boldsymbol{V}^{(n)} + \boldsymbol{I}$ (to take into account the 1-entries on the diagonal of $\boldsymbol{K}^{(n)}$ overwritten by the compression). So, $\boldsymbol{S}^{(n)}$ is partitioned into the following blocks:

$$S^{(n)}_{k,pr} = \sum_{q \in \mathcal{Q}} w_{qr} \tilde{V}^{(n)}_{k,pq} \tag{8}$$

where $S^{(n)}_{k,pr}$ is the kth block of $\boldsymbol{S}^{(n)}$ held by the recovery processor (p, r) (with $1 \leq k \leq K$ and $r \in \mathcal{R}$), and analogously $\tilde{V}^{(n)}_{k,pq}$ is the kth block of $\tilde{\boldsymbol{V}}^{(n)}$ that is stored on the computing processor (p, q). Also, w_{qr} is the entry of row q and column r of a weight matrix $\boldsymbol{W}$. Indeed, to recover multiple faults, the checksums must be weighted employing a $Q \times R$ matrix suitably conceived for the purpose (such that any square sub-matrix of $\boldsymbol{W}$ is non-singular [13], [26]). All blocks $S^{(n)}_{k,pr}$ have $n_b \times n_b$ elements and each recovery processor hosts at most K blocks (as any other computing processor).

IMe operates on the checksum blocks as if they were matrix blocks. Hence, we can imagine to iteratively apply the fundamental formula (5) to the matrix $[\boldsymbol{V}|\boldsymbol{S}]^{(n)}$.

Theorem V.3. The iterative application of (5) to $[\boldsymbol{V}|\boldsymbol{S}]^{(n)}$ preserves the checksum invariant i.e., at any level l, the submatrix $\boldsymbol{S}^{(l)}$ holds the checksum of $\boldsymbol{V}^{(l)}$.

Theorem 5.3 (proof in Appendix) ensures ABFT for IMe.

VI. The Parallel Implementation

A parallel message-passing implementation of the algorithm is shown in Algorithm 1. In the following, we will refer to it as Compressed Inhibition Method with Fault Tolerance (C-IMeFT).

Algorithm 1 shows the pseudocode of a generic (computing or recovery) processor (p, q), which receives as input a set $A_{*,pq}$ of blocks of the coefficient matrix $\boldsymbol{A}$, and the diagonal $d(\boldsymbol{A})$. These

Algorithm 1: Pseudocode of C-IMeFT on a Generic Processor (p, q), With $p \in [0, P), q \in \mathcal{Q} \cup \mathcal{R}$.

Input: $A_{*,pq}$, blocks of $\boldsymbol{A}$ assigned to processor (p, q), supposed empty if $q \in \mathcal{R}$; $d(\boldsymbol{A})$, diagonal of $\boldsymbol{A}$.
Output: $\widehat{V}^1_{*,pq}$ blocks of $\widehat{\boldsymbol{V}}^1$ assigned to processor (p, q).

1: **procedure** C-IMeFT($A_{*,pq}, d(\boldsymbol{A})$)
2: $\widehat{V}_{*,pq} \leftarrow$ IMeInit($A_{*,pq}, d(\boldsymbol{A})$)
3: $row_p \leftarrow$ set of processors on the same row of (p, q)
4: $col_q \leftarrow$ set of processors on the same column of (p, q)
5: $(p, q') \leftarrow$ processor holding portion of $d(\widehat{\boldsymbol{V}})$ on the same row of (p, q)
6: **for** $l \leftarrow n \ldots 2$ **do**
7: $(p', q) \leftarrow$ processor holding portion of $\hat{v}_{l,*}$ on the same column of (p, q)
8: $(p, q'') \leftarrow$ processor holding portion of $\hat{v}_{*,l}$ on the same row of (p, q)
9: Broadcast($\hat{v}_{*,l}, row_p$,root=(p, q''))
10: Broadcast($\hat{v}_{l,*}, col_q$,root=(p', q))
11: **if** $q == q'$ **then**
12: **for each** α_i **on** (p, q') **do**
13: $\alpha_i = 1 - \hat{v}_{i,l} * \hat{v}_{l,i}$
14: **end for**
15: **end if**
16: Broadcast($\boldsymbol{\alpha}_{pq'}, row_p$,root=$(p, q')$)
17: **for each** $i, j | i < l \wedge \hat{v}_{i,j} \in \widehat{V}_{*,pq}$ **do**
18: **if** $j == i$ **then**
19: $\hat{v}_{i,i} \leftarrow \hat{v}_{i,i}/\alpha_i$
20: **else if** $j == l$ **then**
21: $\hat{v}_{i,l} \leftarrow -\hat{v}_{i,l}\hat{v}_{l,l}/\alpha_i$
22: **else**
23: $\hat{v}_{i,j} \leftarrow (\hat{v}_{i,j} - \hat{v}_{i,l}\hat{v}_{l,j})/\alpha_i$
24: **end if**
25: **end for**
26: **end for**
27: **return** $\widehat{V}_{*,pq}$
28: **end procedure**

data are used for initialising the elements of $\widehat{\boldsymbol{V}} = [\boldsymbol{V}|\boldsymbol{S}]^{(n)}$ held by (p, q) (procedure IMeInit on line 2). That is, (p, q) receives the blocks $V_{*,pq}$ if it is a computing processor ($q \in \mathcal{Q}$), or $S_{*,pq}$ if it is a checksum processor ($q \in \mathcal{R}$).

In order to apply the fundamental formula, at each level l the processor (p, q) needs a portion of the $\boldsymbol{\alpha}$ vector, and a portion of the lth column and lth row of $\widehat{\boldsymbol{V}}$. Therefore, at each iteration, the algorithm enforces a set of broadcast communications. Fig. 5 highlights the blocks containing relevant information for the computation carried out on (p, q). In particular, the processor (p, q'') on the same processor row of (p, q) holding a portion of the lth column of $\widehat{\boldsymbol{V}}$, sends it to the whole processor row (broadcast of $\hat{v}_{*,l}$ to row_p on line 9); the processor (p', q) on the same column of (p, q) holding a portion of the lth row of $\widehat{\boldsymbol{V}}$, sends it to the whole processor column (broadcast of $\hat{v}_{l,*}$ to col_q on line 10). After these exchanges, the processors holding the main diagonal of $\widehat{\boldsymbol{V}}$ have all that is needed to compute a portion of the $\boldsymbol{\alpha}$ vector (line 13). The portion is then broadcasted to all the processors on the same row (line 16). Then, all processors can apply the fundamental formula (lines 19–23) on all the elements of their blocks.[1]

Thanks to the adopted 2D block-cyclic distribution, the three collective communications in Algorithm 1 involve only processors on the same row or column. As the number of processors engaged in each collective communication is reduced w.r.t a broadcast involving all the nodes (which would be required in column-wise repartition), this strategy helps to reduce the synchronisation points. Furthermore, while the collective operation in line 16 needs to be performed after the computation in line 13 is completed, the first two broadcasts (lines 9 and 10) can be executed concurrently because they regard different elements of the matrix that are only accessed but not modified at this stage. The concurrency of these communications contributes to the containment of the computation time.

Our ABFT technique shows two advantages w.r.t. diskless checkpointing: *i)* there is no need to periodically interrupt the computation to save the state of the system, but an encoded version of it is constantly updated on the recovery processors; *ii)* the encoded state is always up-to-date, so if a failure occurs, there is no need to perform a rollback of the whole matrix to the latest available copy of the state.

When compared with the hybrid ABFT-checkpointing technique by Bouteillers et al. [27], C-IMeFT shows two advantages: *i)* a uniform fault-tolerant technique to protect the whole matrix because resilience is obtained with a pure ABFT approach; *ii)* the memory overhead to host the checksums is half the size required by [27]. Indeed, to obtain resilience from up to R concurrent faults during the elaboration, C-IMeFT needs $P \times R$ additional processors and a memory overhead of $PRKn_b^2$. Bouteillers et al. [27] suggest a mechanism to allow storing the checksums on the same processors used for the factorisation, but the additional memory occupation is $2PRKn_b^2$.

As explained in Section V, the flops required by the compressed version of IMe are $O(\frac{3}{2}n^3)$. Regarding the computational overhead introduced by our fault-tolerant strategy, it can be calculated starting from the observation that each recovery processor is subject to the same computing load as any computing processor. Indeed, the $P \times R$ additional processors operate on their checksum blocks as if they were part of the coefficient matrix, and each recovery processor hosts at most K blocks (as any of the other $P \times Q$ computing processors). Hence, fault tolerance makes the flops increase to $\frac{R}{Q} \cdot O(\frac{3}{2}n^3)$.

VII. Rollback-Free Distributed Recovery

When one or more hard faults occur, all the computing and recovery nodes still alive need to be notified of the event. Different solutions have been proposed to orchestrate this process [42], [43]. For example, Hydra[2]—a process manager used by

[1] Differently from the original description of the IMe, here α_i is computed without the reciprocal in line 13, and its value is used as the divisor in the fundamental formula. This allows for avoiding useless flops at each iteration.

[2] [Online]. Available: https://github.com/pmodels/mpich/blob/main/doc/wiki/how_to/Using_the_Hydra_Process_Manager.md

default in famous Message Passing Interface (MPI) implementations like MPICH[3] and IntelMPI[4]—provides a mechanism to automatically send a signal to all running processes, including those suspended on communication primitives, whenever a fault is detected.

In this section, we assume a lower-level mechanism (like the one provided by Hydra) is in charge of detecting the faults and notifying all the other working processors. Given this assumption, we focus on how the rollback-free recovery algorithm of C-IMeFT can be implemented in a distributed fashion. Indeed, a parallel implementation of this step helps in various ways:

1) it avoids a single point of failure;
2) it improves the performance by leveraging the resources of multiple nodes during the computation; and
3) it allows overcoming the limitations related to the size of the memory of a single node.

If IMe computation is carried out on $P \times Q$ computing processors and we have $P \times R$ additional recovery processors, we can face up to R anywhere-located hard faults, involving either computing or recovery nodes. If the fault actually involves both computing and recovery nodes, the algorithm must first focus on reconstructing the state of the computing processors. Then, the state of the recovery nodes can be determined through a straightforward recomputation of the checksums (as stated by (8)) involving the survived and newly-reconstructed computing nodes. Therefore, in the following, we focus on the more interesting case of F faults (with $F \leq R$) involving only computing processors. In this case, in order to replace the faulty nodes, a subset of available recovery processors (all of them if $F = R$) must be selected. We will call them *activated* recovery processors. The purpose of the recovery algorithm is to turn the activated recovery processors into up-to-date computing nodes. We denote with $\mathcal{F}$ the set of column indexes of the faulty nodes (therefore, $\mathcal{F} \subseteq \mathcal{Q}$ and $|\mathcal{F}| = F$), and with $\mathcal{R}'$ the set of the column indexes of the activated recovery processors (obviously, $\mathcal{R}' \subseteq \mathcal{R}$).

In general, to recover from F faults we need to solve a batch of linear systems, all characterised by the same coefficient matrix $\boldsymbol{W'}$. $\boldsymbol{W'}$ is an $F \times F$ matrix composed of a subset of the weights in $\boldsymbol{W}$: all rows of $\boldsymbol{W}$ corresponding to the faulty computing processors, and all columns corresponding to the activated recovery nodes going to take their place.

First, the algorithm needs to recompute the row-wise checksums of the survived processors on the activated recovery nodes, i.e.,

$$S_{k,pr} = S_{k,pr} - \sum_{q \in \mathcal{Q} \setminus \mathcal{F}} w_{qr} \tilde{V}^{(n)}_{k,pq} \quad \forall r \in \mathcal{R}' \tag{9}$$

Let us now denote with $\boldsymbol{s}_r$ the row vector obtained by collapsing on one dimension all elements of the blocks $S_{*,pr}$ held by the activated recovery processor (p, r)—with $r \in \mathcal{R}'$.

Let us also denote with $\tilde{\boldsymbol{v}}_f$ the row vector obtained by collapsing on one dimension all elements of the blocks $\tilde{V}_{*,pf}$ held by the faulty processor (p, f), with $f \in \mathcal{F}$. Each vector $\tilde{\boldsymbol{v}}_f$ has Kn_b^2 elements. These are the unknowns to be recovered.

[3][Online]. Available: https://www.mpich.org/about/overview/
[4][Online]. Available: https://software.intel.com/en-us/mpi-library

The recovery algorithm must solve:

$$\boldsymbol{W'} \cdot \begin{bmatrix} \tilde{\boldsymbol{v}}_{f_1} \\ \vdots \\ \tilde{\boldsymbol{v}}_{f_F} \end{bmatrix} = \begin{bmatrix} \boldsymbol{s}_{r_1} \\ \vdots \\ \boldsymbol{s}_{r_F} \end{bmatrix} \tag{10}$$

that is, a batch of Kn_b^2 small linear systems, all characterised by the same $F \times F$ coefficient matrix $\boldsymbol{W'}$. The memory overhead introduced by the recovery is ascribable to the matrix $\boldsymbol{W'}$, as it is the only additional structure that needs to be maintained. In general, the dimension of such a matrix should be small ($F \ll n$), entailing not only a limited additional memory occupation but also the fact that $\boldsymbol{W'}$ can be factorised (or inverted) with a very small computational effort; then, solving the batch of linear systems requires a relatively limited number of flops ($2F^2 \cdot Kn_b^2$).Nonetheless, depending on how we implement the recovery process, such a solving step may cause a relevant memory occupation.

For example, a naive "centralised" strategy to recover multiple faults, where a recovery node $r_0 \in \mathcal{R}'$ is elected coordinator, is shown in Fig. 6(b). Node r_0 must first perform a *reduce* operation to collect all the blocks from the other recovery nodes in $\mathcal{R}'$, solve all the systems in (10), and send the results back. Though flops should not be an issue, the coordinator node might not have enough memory to store the blocks from all other activated recovery processors. Also, communications may represent a bottleneck in such a centralised implementation. Alternatively, the coordinator could iteratively collect portions of blocks from the other nodes, solve the correspondent systems and send back the results. This idea reduces the memory requirement on the coordinator but increases the number of messages, and ultimately the synchronisation points.

To overcome the limitations of a centralised solution, we resort to a distributed recovery strategy. Algorithm 2 illustrates it through pseudo-code. The algorithm is executed on any survived computing or activated recovery processor (p, q).

In particular, for each activated recovery processor, all survived computing nodes pre-multiply their blocks for the weights (line 4) and perform a reduce operation to compute (9) in a distributed fashion (line 7). As a result, the constant terms in (10) are now scattered across the activated recovery nodes. Instead of collecting them on a single recovery coordinator for a centralised resolution and sending the results back (as in Fig. 6(b)), the algorithm adopts a distributed resolution strategy. Indeed, solving the linear systems in (10) is equivalent to solving the following:

$$\boldsymbol{Z} \cdot \begin{bmatrix} \boldsymbol{s}_1 \\ \vdots \\ \boldsymbol{s}_F \end{bmatrix} = \begin{bmatrix} \tilde{\boldsymbol{v}}_1 \\ \vdots \\ \tilde{\boldsymbol{v}}_F \end{bmatrix} \tag{11}$$

where $\boldsymbol{Z} = \boldsymbol{W'^{-1}}$. Therefore, the algorithm states that all activated recovery processors pre-multiply their $S_{*,pq}$ blocks for the corresponding coefficients of $\boldsymbol{Z}$ (line 12) and perform a reduce operation to compute (11) in a distributed fashion (line 13).

Fig. 6(c) shows how the checksum blocks are exchanged by the activated recovery processors. Obviously, the number of sent

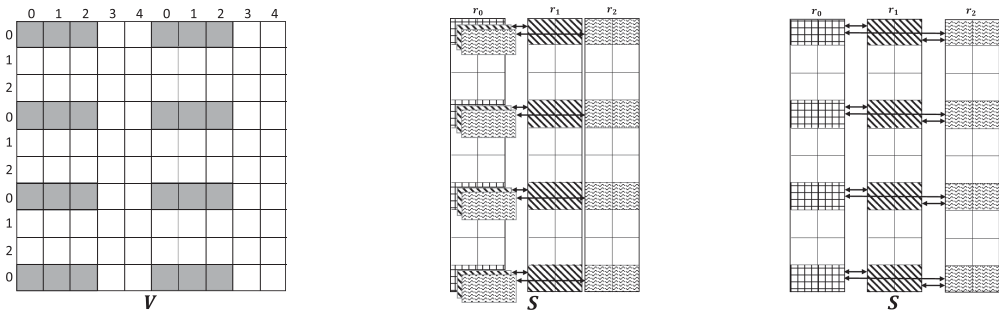

Fig. 6. Different strategies to recover from multiple faults involving the grey blocks in (a) centralised recovery approach (b), where the recovery node r_0 is elected coordinator and thus collects the blocks on the other nodes to reconstruct the state of the fallen ones), and the distributed strategy of Algorithm 2 (c).

Algorithm 2: Distributed Recovery Strategy Executed on a Generic Processor (p, q), With $p \in [0, P), q \in \mathcal{Q} \cup \mathcal{R}'$.

Input: $\boldsymbol{W'}$, weights matrix for the faulty processors; $\boldsymbol{Z} = \boldsymbol{W'^{-1}}$; $V_{*,pq}$, blocks of $\widehat{\boldsymbol{V}}$ assigned to processor (p, q); $\mathcal{R}'$, column indexes of activated recovery processors

Output: $V_{*,pr}$, blocks of $\widehat{\boldsymbol{V}}$ reconstructed after the fault and assigned to processor (p, r) with $p \in [0, P), r \in \mathcal{R}'$.;

1: **procedure** IMeRecover$\boldsymbol{W'}, V_{*,pq}, \ldots$
2: **for each** $(p, r)|r \in \mathcal{R}'$ **do**
3: **if** $q \in \mathcal{Q}$ **then**
4: $S_{*,pr} = -w_{qr} V_{*,pq}$
5: **end if**
6: $row_{(p,\mathcal{Q}+r)} \leftarrow$ group (p, r) with all survived computing processors on the same row of (p, q)
7: REDUCE($\boldsymbol{S}_{*,pr}, row_{(p,\mathcal{Q}+r)}$,root=$(p, r)$)
8: **end for**
9: **if** $q \in \mathcal{R}'$ **then**
10: $row_{(p,\mathcal{R}')} \leftarrow$ all activated recovery processors on the same row of (p, q)
11: **for each** $(p, r)|r \in \mathcal{R}'$ **do**
12: $V_{*,pq} = z_{rp} S_{*,pq}$
13: REDUCE($\boldsymbol{V}_{*,pq}, row_{(p,\mathcal{R}')}$,root=$(p, r)$)
14: **if** $V_{*,pq}$ is a diagonal block **then**
15: $V_{*,pq} = V_{*,pq} - I$
16: **end if**
17: **End for**
18: **end if**
19: **end procedure**

messages is higher w.r.t a centralised implementation, but the overhead of memory occupation is virtually zero.

VIII. Experimental Evaluation

A. Experimental Setup

We evaluate the performance of C-IMeFT and IMERECOVER algorithms on CRESCO6,[5] a HPC cluster built by ENEA, and composed of 434 nodes; each one with two sockets of 24 cores (2.10 GHz Intel Xeon Platinum 8160) and 192 GB RAM. Physical nodes are interconnected by an Intel Omni-Path 100 Gb/s network. Our implementation is included in a prototype library of IMe-based functions available on GitHub.[6] The whole library is written in C language and all communication/synchronization aspects are managed with MPI.

In order to evaluate the performance of our approach, we compare it with two existing fault-tolerant linear system solvers:

- The PDGESV routine of Scalable LAPACK (ScaLAPACK),[7] which implements a Gauss-Jordan solver optimised for the execution on HPC environments; we conveniently modified the function to guarantee fault tolerance through periodic diskless C/R [11]. In each experiment, the checkpointing frequency is calculated as prescribed by the foundational work of Young [44]. In the following, we refer to this solution by using the acronym (SPK+C/R).
- The Fault Tolerant Linear Algebra (FT-LA) library,[8] which implements the ABFT linear solver described in [27]. This library is composed of a subset of ScaLAPACK routines enhanced to intrinsically protect the right factor during the matrix factorisation. The resilience of the left factor is instead obtained through an efficient checkpointing mechanism.

B. Evaluation Approach

To clearly state the strengths and weaknesses of our method, we need to first evaluate the performance enhancements introduced by the compressed version of the algorithm. Then, both the failure-free and recovery overhead must be evaluated. In particular,we are not just interested in the speed-up that our parallel implementation can reach but also in the evaluation of its memory occupation. Also, it is important to investigate the entity of the reconstruction error, whenever the recovery mechanism has to intervene. For these reasons, we structure the tests into fourgroups.

Group 1. Performance enhancements . In order to clearly state the enhancements brought by the proposed technique, we compare C-IMeFT with its previous version IMeFT [13], which required to operate on a matrix that is initially twice the size of $\boldsymbol{A}$.

[5][Online]. Available: https://www.eneagrid.enea.it/CRESCOportal/

[6][Online]. Available: https://github.com/orgs/Reference-IMe

[7][Online]. Available: https://github.com/Reference-scaLAPACK

[8][Online]. Available: https://icl.utk.edu/ft-la/index.html

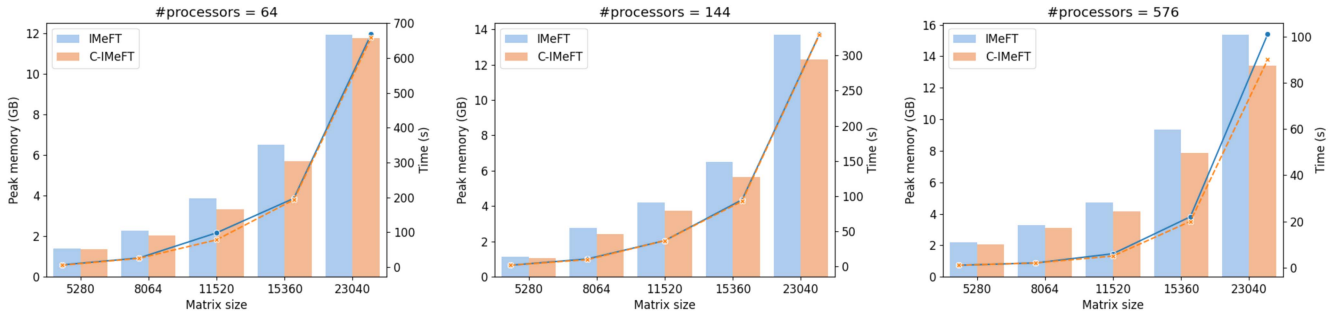

Fig. 7. Comparison of runtime (lines) and memory occupation (bars) of C-IMeFT with its previous version IMeFT.

The comparison is conducted in terms of both runtime and peak memory occupation when solving a set of randomly generated dense linear matrices with increasing sizes. For the measurement of the execution times, we employed the MPI_WTIME directive.[9]

Group 2. Scalability: We evaluate the performance (in terms of time to compute the solution) of C-IMeFT, SPK+C/R and FT-LA when solving a linear system of fixed dimension on an increasing number of computing processors, i.e., we perform a strong scalability test. Then, we also compare the weak scalability by keeping constant the load on each processor and progressively increasing both the matrix size and the number of computing cores. In this way, we evaluate the algorithms' ability to solve bigger systems in the same time whenever more computing nodes are available. In order to better understand the advantages and weaknesses of any approach, for both strong and weak scalability it is important to compare the fault-free performance, and then introduce an increasing number of faults to assess the trends of the recovery overhead. FT-LA implementation offers a performance test mechanism that simulates the occurrence of a fault. Likewise, our evaluation simulates faults for C-IMeFT and SPK+C/R. In case of multiple faults, we imagine them to happen contemporarily, as to simulate the case of malfunctions with overlapping failure periods. We test situations with an increasing number of faults from 0 (fault-free case) to 24. The latter corresponds to a hard fault that involves a whole computing socket of the considered infrastructure. As regards the spatial distribution of the faults, we randomly choose the row of processors involved in the malfunction. Analogously, we randomly choose the time instant in which the set of faults occurs.

Group 3. Memory occupation: We compare the peak memory footprints of the three approaches, for increasing number of computing processors and matrix dimension. These measurements are based on the values provided by the architecture's job scheduler, IBM Spectrum Load Sharing Facility (LSF).[10]

Group 4. Reconstruction Error: For the sake of completeness, we also analyse the error that is introduced in C-IMeFT computation when the occurrence of an error triggers the recovery procedure. We, therefore, compare the solution computed by a fault-free execution, with that obtained after the occurrence of a fault. In particular, we analyse the impact of the location in the matrix and the iteration at which the fault occurs on such a reconstruction error.

C. Results

The results of the four groups of tests are illustrated in the following.

Results of Group 1. Performance enhancements . Fig. 7 compares the peak memory occupation and runtime of C-IMeFT with those of its previous version, IMeFT, when solving matrices of increasing size. Three different execution cases are considered, involving 64, 144 and 576 processors.

As expected, the execution times (illustrated through lines in Fig. 7) of C-IMeFT and IMeFT are comparable. Indeed, the proposed performance enhancements do not reduce the number of flops but rather focus on reducing the dimension of the matrix on which the algorithm operates. Therefore, the key advantage of C-IMeFT stands precisely in the reduced memory footprint (shown through bars in Fig. 7). Given the superiority of C-IMeFT in all the considered cases, we refer to this enhanced version in all the following tests.

Results of Group 2. Scalability: Fig. 8 compares the strong scalability of C-IMeFT, SPK+C/R and FT-LA when increasing number of faults occur while solving dense linear systems with sizes $n = 11520, n = 23040, n = 46080$, corresponding to a memory occupation of around 500 MB, 2 GB and 8 GB, respectively. Each computation time in the graph is averaged over 5 different matrices for each size and 10 different runs.

It can be noted that for FT-LA and C-IMeFT some of the bars do not show the darker colours (corresponding to the executions with higher numbers of faults). This is due to different reasons for the two algorithms. For FT-LA the current implementation does not support multiple faults, so the graphs report only the computing time of the no-fault and the single-fault situations. For C-IMeFT instead, all the fault cases from 0 to 24 are reported but in some situations, it is not possible to appreciate the difference between the execution time of the 4, 8 and 24 fault cases. Indeed, the time to recover from 24 faults is always just slightly higher than the time to recover from one. This is a desired behaviour, because it shows that the recovery time is not heavily influenced by the number of nodes whose state must be reconstructed. Therefore, the histograms show that C-IMeFT is generally able to outperform the other two algorithms, especially when many faults occur.

[9][Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[10][Online]. Available: https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=components-lsf-documentation

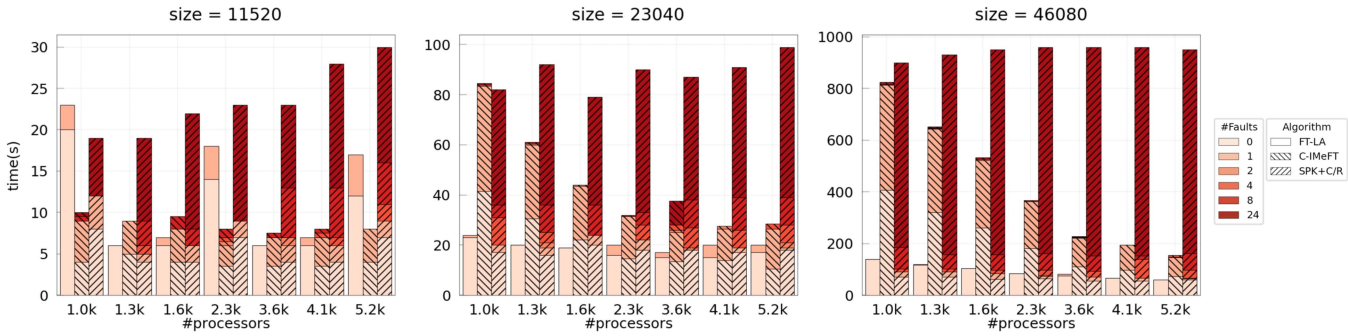

Fig. 8. Strong scalability analysis of C-IMeFT, SPK+C/R and FT-LA.

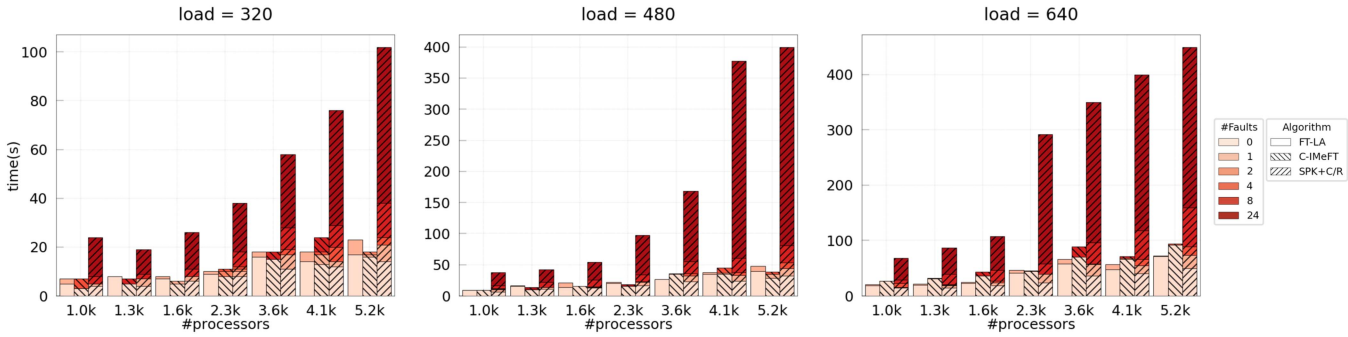

Fig. 9. Weak scalability analysis of C-IMeFT, SPK+C/R and FT-LA.

Similar trends are also visible when analysing the weak scalability. Fig. 9 reports the execution times (again averaged on 5 matrices and 10 repetitions) of the three algorithms when we increase both the matrix dimension and the computing processors so that the load on each computing node is kept constant: Fig. 9 reports the performance when each processor works on a portion of the original matrix of dimensions 320×320, 480×480 and 640×640. The four graphs show that C-IMeFT outperforms SPK + C/R even for low values of faults, especially when the load on each processor is contained. Furthermore, when fault occurrence is high (around 4 or higher) C-IMeFT is almost always the best-performing algorithm.

Again, we underline that the current implementation of FT-LA does not support multiple faults, so the graphs report only the computing time of the no-fault and the single-fault situations.

Results of Group 3. Memory occupation: The graphs of Fig. 10 compare the peak memory occupation caused by each of the three considered algorithms when solving a system of fixed size on an increasing number of computing nodes. All values are averaged on 5 different matrices for each chosen matrix dimension and 10 repetitions. Fig. 10(a), (b), and (c) focus on the fault-free performance, i.e., the memory footprint of each algorithm when the fault tolerance mechanism is present but no fault occurs. Conversely, Fig. 10(d), (e), and (f) compare the memory occupation when a fault occurs and is recovered. All the graphs highlight the capability of C-IMeFT to sensibly reduce the memory occupation w.r.t to ScaLAPACK+C/R. However, it can be noted that for big matrices C-IMeFT performs only slightly better than FT-LA. Indeed, C-IMeFT halves the memory overhead of the checksum matrix w.r.t. FT-LA: when the matrix of coefficients is big, its memory occupation dominates that of the checksums, and the advantage of C-IMeFT appears more limited.

Results of Group 4. Reconstruction Error: Fig. 11 shows the difference between the solution computed by a fault-free execution of C-IMeFT and that of a computation in which the occurrence of a fault triggers the execution of the IMERECOVER procedure. For each matrix size, the results are averaged over five executions with different randomly generated matrices. Overall, the graphs show a rather contained error, rarely reaching the order of 10^{-8}.

Fig. 11(a) compares the errors when the fault occurs at the same iteration but involves different locations of the matrix $\boldsymbol{V}$. In general, a fault causing a hole at the bottom right of $\boldsymbol{V}$ seems to cause a slightly higher error than a top left hole. This is indeed expected because values at the bottom right of $\boldsymbol{V}$ are involved in initial iterations and contribute to all the following.

Fig. 11(b) instead, shows the difference in the impact of a fault occurring at the beginning of the computation (i.e., initial iterations), w.r.t. one towards the end. For all matrix dimensions, the slightly higher values of errors at the end of the computation suggest the presence of a desirable cancellation phenomenon, i.e., an error at the beginning seems to be progressively reduced during computation. Further investigations—out of the scope of this work—are needed in order to assess the precise nature and entity of this phenomenon.

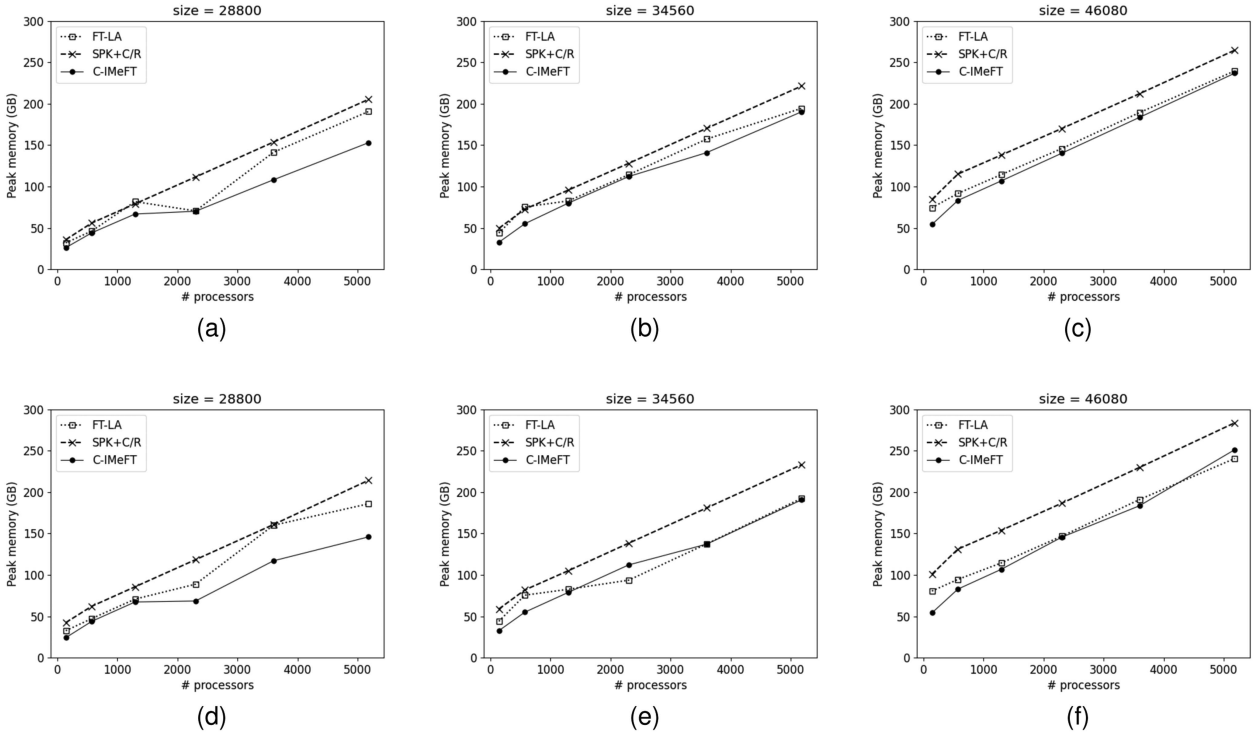

Fig. 10. Peak memory occupation of C-IMeFT, SPK+C/R and FT-LA. (a), (b), and (c) refer to the fault-free condition; (d), (e), and (f) to the occurrence and recovery from one fault.

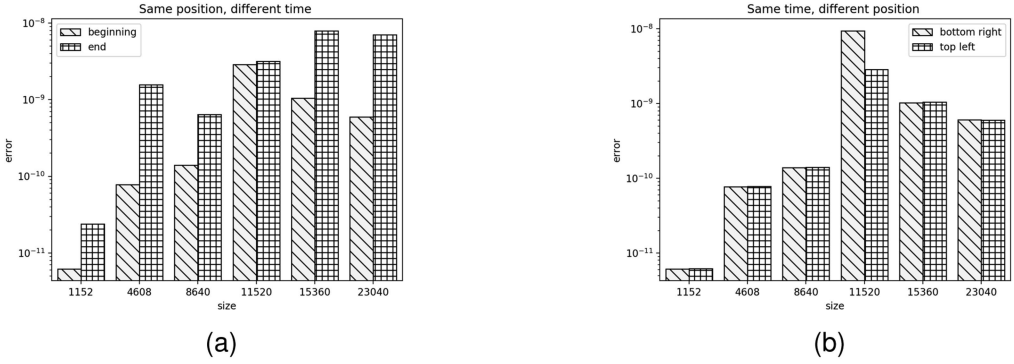

Fig. 11. Comparison of norm-wise reconstruction error of IMeRecover procedure when (a) the fault occurs at different points in time during the computation, and (b) when it involves different matrix elements.

IX. Discussion

The experimental evaluation highlights a series of key advantages when employing C-IMeFT on an HPC infrastructure. Nonetheless, some practical aspects need to be further discussed.

First of all, environmental conditions such as the expected frequency of faults, must always be considered when choosing a fault-tolerant strategy in an HPC context. Indeed, the proposed approach states that, in order to be guaranteed from up to R faults, $P \times R$ additional nodes must be involved in the computation from the beginning. C-IMeFT shows the great advantage that the faults can be located anywhere, and (as shown in Fig. 4) in many cases, with $P \times R$ checksum nodes, we can recover from more than R faults. On the other hand, the strategy comes with the drawback of an increased cost of running due to spare nodes that must be allocated and take part in the computation even if no fault actually happens. Therefore, the cost of employing C-IMeFT must always be evaluated w.r.t. the probability of fault occurrence for the underlying infrastructure. Furthermore, analogously to diskless C/R, C-IMeFT cannot cancel the risk of having to recompute everything from scratch in case of losing all the nodes. If the infrastructure is such that this circumstance is probable, C/R on a reliable storage is certainly preferrable.

Another point of discussion regards the nature of the underlying HPC system. The proposed algorithm was designed to run on a network of interconnected CPUs. Recent works have shown how graphical accelerators can be used to boost the performance of linear algebra algorithms [45], [46]. C-IMeFT too can be adapted to run on a GPU (some recent attempts exist in this regard [47], [48]), but the nature of graphical accelerators can render the adoption of an ABFT technique such as C-IMeFT useless: hard faults involving specific GPU cores are rather infrequent events, whereas the failure of an entire GPU is more probable. In the latter case, the whole computation would be lost, making (again) the adoption of C/R on reliable storage preferable w.r.t. C-IMeFT. A different case is the execution on

a multi-GPUs system: C-IMeFT could—in theory—be adapted to recover the state of faulty GPUs. However, the current replication scheme would probably need to be redefined in order to reduce the number of additional nodes for recovery.

X. Conclusion

In the last years, the reliability of supercomputers has not increased at the same pace as their impressive computing capabilities. ABFT is one of the (many) ways to provide fault tolerance to HPC systems. In this paper, we have concentrated attention on the resolution of dense, unstructured, linear systems—a task common to many scientific fields—and we have described how an existing solver can be enhanced to reduce its memory footprint and, contemporary, intrinsically provide fault tolerance to multiple anywhere-located faults at the price of a limited overhead. We have also proposed a rollback-free distributed approach to overcome the limitations of a centralised recovery strategy, and we have conducted an extensive experimental evaluation on a real-life HPC environment to asses the algorithm's performance in terms of scalability, memory footprint and reconstruction error. The analysis suggests that our method is generally able to outperform state-of-the-art thecniques, in particular when fault tolerance to a significant number of faults is required. Moreover, the memory occupation investigations highlight a desirable reduced footprint both in fault-free conditions and when a fault occurs. Finally, the reconstruction error analysis shows that the application of the recovery method has a limited impact on the final result.

As energy consumption is becoming a matter of primary concern in HPC environments, we plan to further investigate the advantages that IMe can bring on this front. A preliminary investigation seems to show that the promising results obtained in terms of reducing the computation time translate into equally valuable improvements in the containment of power consumption. In the near future, we plan to deepen the analysis of the energy gains achievable through our technique.

Moreover, the approach to fault tolerance employed in this work is based on the computation (and subsequent availability) of checksums. This feature could—in principle—be easily used to also highlight and repair soft errors (inducing variations in the results without causing a complete node failure) during the computation. Future works will deepen the concrete feasibility of employing this technique to tackle soft errors.

Related to this topic, the implementation of C-IMeFT on GPUs could be valuable. Indeed, differently form hard faults, soft errors involving only some GPU cores are not infrequent. Once a soft error management strategy is theoretically defined for the IMe algorithm, it could be interesting to evaluate the performance of a GPU implementation.

Acknowledgments

The computing resources and the related technical support used for this work have been provided by CRESCO/ENEAGRID High Performance Computing infrastructure and its staff. CRESCO/ENEAGRID is funded by ENEA and by Italian and European research programmes. This work has been realized by Daniela Loreti with a research contract co-financed by the European Union - PON Ricerca e Innovazione 2014-2020 ai sensi dell'art. 24, comma 3, lett. a), della Legge 30 dicembre 2010, n. 240 e s.m.i. e del D.M. 10 agosto 2021 n. 1062. The authors want to acknowledge also the CINECA Consortium for the availability of the HPC resources and the technical support provided in the framework of the IscraC project En-FRLS.

References

[1] M. Malms et al., "ETP4HPC's SRA 5 - strategic research agenda for high-performance computing in Europe - 2022," 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7347009

[2] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 4, pp. 337–350, Fourth Quart. 2010, doi: 10.1109/TDSC.2009.4.

[3] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing Front. Innov.*, vol. 1, no. 1, pp. 5–28, Apr. 2014, doi: 10.14529/jsfi140101.

[4] M. Bouguerra, A. Gainaru, L. A. Bautista-Gomez, F. Cappello, S. Matsuoka, and N. Maruyama, "Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 501–512, doi: 10.1109/IPDPS.2013.74.

[5] G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI," in *Proc. Supercomputing Symp.*, 1994, pp. 379–386.

[6] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "A job pause service under LAM/MPI BLCR for transparent fault tolerance," in *Proc. IEEE 21th Int. Parallel Distrib. Process. Symp.*, 2007, pp. 1–10, doi: 10.1109/IPDPS.2007.370307.

[7] P. V. Cardoso and P. P. Barcelos, "Definition of an architecture for dynamic and automatic checkpoints on apache spark," in *Proc. IEEE 37th Symp. Reliable Distrib. Syst.*, 2018, pp. 271–272.

[8] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs," in *Proc. ACM/IEEE SC2004 Conf. High Perform. Netw. Comput.*, 2004, p. 38, doi: 10.1109/SC.2004.29.

[9] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of MPI programs," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2003, pp. 84–94, doi: 10.1145/781498.781513.

[10] K. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, no. 6, pp. 518–528, Jun. 1984, doi: 10.1109/TC.1984.1676475.

[11] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, Oct. 1998, doi: 10.1109/71.730527.

[12] M. Artioli, D. Loreti, and A. Ciampolini, "Fault tolerant high performance solver for linear equation systems," in *Proc. IEEE 39th Symp. Reliable Distrib. Syst.*, 2019, pp. 113–11309.

[13] D. Loreti, M. Artioli, and A. Ciampolini, "Solving linear systems on high performance hardware with resilience to multiple hard faults," in *Proc. IEEE Int. Symp. Reliable Distrib. Syst.*, 2020, pp. 266–275, doi: 10.1109/SRDS51746.2020.00034.

[14] M. Artioli and F. Filippetti, "IMe: A. general method to analyse linear systems and electric circuits," in *Software for Electrical Engineering Analysis and Design V*. Ashurst, Southampton, U.K.: WIT Press, 2001, pp. 147–162. [Online]. Available: https://www.witpress.com/elibrary/wit-transactions-on-engineering-sciences/31/3346

[15] F. Filippetti and M. Artioli, "IMe: 4-term formula method for the symbolic analysis of linear circuits," *IEEE Trans. Circuits Syst.*, vol. 51, no. 3, pp. 526–538, Mar. 2004, doi: 10.1109/TCSI.2003.822374.

[16] S. S. Mukherjee, J. S. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *Proc. IEEE 11th Int. Conf. High-Perform. Comput. Architecture*, 2005, pp. 243–247, doi: 10.1109/HPCA.2005.37.

[17] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *J. Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013, doi: 10.1007/s11227-013-0884-0.

[18] O. O. Sudakov, I. S. Meshcheriakov, and Y. V. Boyko, "CHPOX: Transparent checkpointing system for Linux clusters," in *Proc. IEEE 4th Workshop Intell. Data Acquisition Adv. Comput. Systems: Technol. Appl.*, 2007, pp. 159–164.

[19] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of zap: A system for migrating computing environments," in *Proc. 5th Symp. Operating Syst. Des. Implementation*, 2002, pp. 361–376. [Online]. Available: http://www.usenix.org/events/osdi02/tech/osman.html
[20] W. Bland, A. Bouteiller, T. Hérault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in MPI," *Computing*, vol. 95, no. 12, pp. 1171–1184, 2013, doi: 10.1007/s00607-013-0331-3.
[21] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Balatonfüred, Hungary: Springer, 2000, pp. 346–353, doi: 10.1007/3-540-45255-9_47.
[22] G. Bosilca et al., "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *Proc. ACM/IEEE Conf. Supercomputing*, 2002, pp. 31:1–31:18, doi: 10.1109/SC.2002.10048.
[23] A. Bouteiller, G. Bosilca, and J. J. Dongarra, "Redesigning the message logging model for high performance," *Concurrency Computation: Pract. Experience*, vol. 22, no. 16, pp. 2196–2211, 2010, doi: 10.1002/cpe.1589.
[24] M. Gholami, F. Schintke, and T. Schütt, "Checkpoint scheduling for shared usage of burst-buffers in supercomputers," in *Proc. 47th Int. Conf. Parallel Process. Companion*, 2018, pp. 1–10, doi: 10.1145/3229710.3229755.
[25] Z. Chen et al., "Fault tolerant high performance computing by a coding approach," in *Proc. 10th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2005, pp. 213–223, doi: 10.1145/1065944.1065973.
[26] Z. Chen, "Scalable techniques for fault tolerant high performance computing," Ph.D. dissertation, Univ. Tennesse, Knoxville, TN, USA, 2006.
[27] A. Bouteiller, T. Hérault, G. Bosilca, P. Du, and J. J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy," *ACM Trans. Parallel Comput.*, vol. 1, no. 2, pp. 10:1–10:28, 2015, doi: 10.1145/2686892.
[28] C. J. Anfinson and F. T. Luk, "A linear algebraic model of algorithm-based fault tolerance," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1599–1604, Dec. 1988, doi: 10.1109/12.9736.
[29] P. Banerjee et al., "Algorithm-based fault tolerance on a hypercube multiprocessor," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1132–1145, 1990, doi: 10.1109/12.57055.
[30] D. L. Boley, R. P. Brent, G. H. Golub, and F. T. Luk, "Algorithmic fault tolerance using the Lanczos method," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 1, pp. 312–332, 1992, doi: 10.1137/0613023.
[31] Z. Chen and J. J. Dongarra, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 12, pp. 1628–1641, Dec. 2008, doi: 10.1109/TPDS.2008.58.
[32] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *Proc. 20th ACM Int. Symp. High Perform. Distrib. Comput.*, 2011, pp. 73–84, doi: 10.1145/1996130.1996142.
[33] T. Davies and Z. Chen, "Fault tolerant linear algebra: Recovering from fail-stop failures without checkpointing," in *Proc. IEEE 24th Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–4, doi: 10.1109/IPDPSW.2010.5470775.
[34] D. Hakkarinen and Z. Chen, "Algorithmic cholesky factorization fault recovery," in *Proc. IEEE 24th Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–10, doi: 10.1109/IPDPS.2010.5470436.
[35] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Prog.*, 2012, pp. 225–234, doi: 10.1145/2145816.2145845.
[36] C. Coti, L. Petrucci, and D. A. T. González, "Fault-tolerant LU factorization is low cost," in *Euro-Par*. Berlin, Germany: Springer, 2021, pp. 536–549.
[37] Y. Zhu, Y. Liu, and G. Zhang, "FT-PBLAS: Pblas-based fault-tolerant linear algebra computation on high-performance computing systems," *IEEE Access*, vol. 8, pp. 42674–42688, 2020.
[38] V. L. Fèvre, T. Hérault, J. Langou, and Y. Robert, "A comparison of several fault-tolerance methods for the detection and correction of floating-point errors in matrix-matrix multiplication," in *Proc. Eur. Conf. Parallel Process.*, 2020, pp. 303–315.
[39] X. Kang, D. F. Gleich, A. H. Sameh, and A. Grama, "Adaptive erasure coded fault tolerant linear system solver," *ACM Trans. Parallel Comput.*, vol. 8, no. 4, pp. 21:1–21:19, 2021.
[40] X. Kang, D. F. Gleich, A. H. Sameh, and A. Grama, "Distributed fault tolerant linear system solvers based on erasure coding," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 2478–2485.
[41] F. Ciampolini, "Un metodo di soluzione dei circuiti lineari," L'Elettrotecnica, vol. L, no. 10, 1963.
[42] 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 Gener. Comput. Syst.*, vol. 106, pp. 467–481, 2020.
[43] G. E. Fagg et al., "Process fault tolerance: Semantics, design and applications for high performance computing," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 4, pp. 465–477, 2005.
[44] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974.
[45] T. Dong, A. Haidar, P. Luszczek, J. A. Harris, S. Tomov, and J. J. Dongarra, "LU factorization of small matrices: Accelerating batched DGETRF on the GPU," in *Proc. EEE Int. Conf. High Perform. Comput. Commun., IEEE 6th Int. Symp. Cyberspace Saf. Secur., IEEE 11th Int. Conf. Embedded Softw. Syst.*, 2014, pp. 157–160.
[46] A. Haidar, A. Abdelfattah, M. Zounon, S. Tomov, and J. Dongarra, "A guide for achieving high performance with very small matrices on GPU: A case study of batched LU and Cholesky factorizations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 5, pp. 973–984, May 2018, doi: 10.1109/TPDS.2017.2783929.
[47] G. Cortesi, "Analisi e uso di architetture GPU per la risoluzione di piccoli sistemi lineari in ambiente HPC," Bachelor's thesis, Univ. Bologna, Bologna, Italy, 2019.
[48] N. Thomopulos, "Sviluppo e sperimentazione di algoritmi per la soluzione di sistemi lineari su GP-GPU" Master's thesis, Univ. Bologna, Bologna, Italy, 2020.

Daniela Loreti received the PhD degree in computer science in 2016. She is currently a junior assistant professor of operating systems with the Department of Computer Science and Engineering, University of Bologna, Bologna, Italy. Her research interests include distributed systems for Big Data management and stream processing and parallel paradigms for high performance computing. She is also interested in the parallelization of artificial intelligence techniques in the fields of machine learning, process mining, and expert systems.

Marcello Artioli received the PhD degree in electrical engineering. He is currently a researcher with ENEA, the Italian National Agency for New Technologies, Energy and Sustainable Economic Development, where he works as system integrator and data analyst. His research interests include high performance computing, fault diagnosis, and signal analysis. He is involved in several international projects, and due to his interdisciplinary vocation, he has successfully contributed to projects outside his core research domain.

Anna Ciampolini received the PhD degree in computer science and engineering. She is currently a full professor with the Department of Computer Science and Engineering, University of Bologna, where she teaches operating systems. Her research interests include operating systems, virtualization techniques and cloud computing, parallel and distributed programming, automatic management of Cloud computing systems, distributed platforms for Big Data analysis, distributed artificial intelligence, with particular regard to distributed automated reasoning. She is involved in several international projects, also in coordination roles. Her research interests include application and theoretical aspects as shown by her broad bibliographic production.

Open Access funding provided by 'Alma Mater Studiorum - Università di Bologna' within the CRUI CARE Agreement