Solving Linear Systems on High Performance Hardware with Resilience to Multiple Hard Faults

Solving Linear Systems on High Performance Hardware with Resilience to Multiple Hard Faults

Daniela Loreti
DISI
University of Bologna
Bologna, Italy
daniela.loreti@unibo.it

Marcello Artioli
Bologna research center
ENEA
Bologna, Italy
marcello.artioli@enea.it

Anna Ciampolini
DISI
University of Bologna
Bologna, Italy
anna.ciampolini@unibo.it

***Abstract*—As large-scale linear equation systems are pervasive in many scientific fields, great efforts have been done over the last decade in realizing efficient techniques to solve such systems, possibly relying on High Performance Computing (HPC) infrastructures to boost the performance. In this framework, the ever-growing scale of supercomputers inevitably increases the frequency of faults, making it a crucial issue of HPC application development.**

A previous study [1] investigated the possibility to enhance the Inhibition Method (IMe) – a linear systems solver for dense unstructured matrices – with fault tolerance to single hard errors, i.e. failures causing one computing processor to stop.

This article extends [1] by proposing an efficient technique to obtain fault tolerance to multiple hard errors, which may occur concurrently on different processors belonging to the same or different machines. An improved parallel implementation is also proposed, which is particularly suitable for HPC environments and moves towards the direction of a complete decentralization. The theoretical analysis suggests that the technique (which does not require checkpointing, nor rollback) is able to provide fault tolerance to multiple faults at the price of a small overhead and a limited number of additional processors to store the checksums. Experimental results on a HPC architecture validate the theoretical study, showing promising performance improvements w.r.t. a popular fault-tolerant solving technique.

***Index Terms*—Fault tolerance, multiple hard faults, High Performance Computing, linear equation systems solver, Inhibition Method**

I. Introduction

Solving linear equation systems is a crucial task for several real world applications, spanning from the field of engineering to that of medicine. As these disciplines often deal with large-scale algebraic systems, characterized by a remarkable number – order of thousands – of linear equations, over the last decade a great deal of effort has been invested on efficient techniques to solve linear systems, possibly resorting to parallelization, so that the solver's algorithm could be executed on a High Performance Computing (HPC) system to improve performance. Circuit simulation, computed tomography and medical image processing in general, aerodynamic design, and electric power network analysis are just few examples of well-known applications that already benefit from efficient, parallel implementations of linear solvers.

The existing variety of methods for linear system resolution can be classified into two main categories: direct solvers, able to identify the exact solution; and iterative solvers, performing incremental enhancements of the solution until the desired accuracy is reached. Albeit iterative solvers are generally preferable for better performance, they are sometimes subjected to the issue of convergence, so that the desired approximation might be unreachable. In this case, the developer can use direct methods, such as Gaussian Elimination, QR, LU [54], or – when the system matrix has particular properties – Cholesky decomposition [43]. Furthermore, when the significant size of the problem forces the employment of a HPC system, iterative methods might require additional effort to cope with some classical issues connected to parallel implementations. For example, deterministic iterative methods impose many synchronization points because each iteration intrinsically depends on the result of the previous one, whereas stochastic iterative approaches require pseudo-random number generation, which is a challenging task in a decentralized system.

A HPC system may include thousands of computational nodes, either traditional CPUs, or GPU cores. In this framework, although the Mean Time Between Failures (MTBF) of a single node has increased over the years, the huge number of hardware components makes failures still extremely frequent for HPC systems [2]. It is therefore crucial to identify and handle malfunctions during the computation, possibly avoiding to restart it from the beginning. Most of the modern HPC systems provide solutions for the promptly isolation of *hard errors* (which cause the machine to stop), but their following management (i.e. the sequence of recovery operations to be carried out once the malfunction has been detected) must be specific of the executed algorithm in order to obtain the most efficient resource usage. Besides hard errors, *soft errors* or Silent Data Corruption (SDC) [3] might also occur: these are malfunctions (like thermal drift or radiation) that do not cause the machine to stop, but may induce errors in the result. To cope with this kind of malfunctions Algorithm Based Fault Tolerance (ABFT) frameworks have been proposed [4]–[6].

Hard errors are far easier than soft errors to be identified. On the other hand, their management is – in a sense – more problematic w.r.t. soft errors due to the catastrophic nature of their effects (e.g., aborted calculations). Currently, a popular approach to tolerate hard errors in large HPC clusters

is Checkpoint/Restart (C/R) at application level: a periodic dump of the calculation state is saved somewhere under the control of the application. In case of error, once the faulty unit has been replaced and the state restored, the application can continue (instead of being started over if no checkpoint is available). This approach comes in two main variants: *(i)* disk checkpointing i.e., saving the state to mass storage, which leads to a big overhead during normal operations and thus, is almost impractical for large clusters data structures; and *(ii)* diskless checkpointing i.e., saving the checksummed state to the internal memory of redundant nodes, which entails smaller but not negligible overhead (checksumming and storing to RAM is faster but interrupts the calculation flow).

In this work, we employ ABFT to address the need for an efficient, hard error-resilient algorithm to solve linear equation systems characterized by dense unstructured matrices. The technique is grounded on an existing direct solver, namely Inhibition Method (IMe), which was initially conceived to analyse complex electric circuits [7], [8] and later generalized to cope with linear systems [9], [10]. IMe has already proven successful in providing efficient single-fault resilience to the linear system resolution process [1]. Nonetheless in large HPC environment, each machine usually has numerous cores and the case of a failure stopping a single core while all the others continue to work, is rather rare. More frequently, a hard failure contemporary involves many – if not all – the computing processor on the same machine. Furthermore, the presence of a single point of failure in the Master/Slave algorithm presented in [1] intrinsically exposes the system to higher risks of aborted calculations. Therefore, this paper focuses on enhancing the previous study [1], with resilience to multiple hard faults while allowing a simpler parallelization of the solving process where no central coordinator is needed. Fault tolerance is not obtained through periodical checkpointing on synchronization points, but rather by encoding the data in a checksum only once at the beginning of the computation. The algorithm can later operate in the same way on both the input and the encoded data. Whereas for other existing methods, changes in the calculation state normally trigger a new checksum for a checkpoint to be valid, in the proposed approach, checksumming is implicit and transparent, with virtually no overhead for the application. Besides a brief introduction to IMe's working principles (Section II), the main contributions of this work are:

- a decentralized IMe-based parallel linear systems solver and a study of its complexity in terms of floting point operations (flops), memory occupation, number and volume of exchanged messages (Section III);
- a description (supported by proof) of how the algorithm can be extended to tackle multiple hard errors, and a further analysis of the complexity introduced by such enhancement (Section IV);
- an empirical performance comparison of the proposed approach with a traditional diskless C/R method to ensure fault tolerance in solving linear systems (Section V).

Related work (Section VI) and conclusion (Section VII) follow.

II. Background on the Inhibition Method

Initially conceived in 1963 [7] to analyse physical linear systems and in particular linear electric circuits, IMe is an exact method based on the Effects Superposition Theorem, which was later applied to the resolution linear equation systems [8] and square matrix inversion [11].

The method considers the linear system $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ is the $n \times n$ matrix of coefficients, $\boldsymbol{b}$ is the vector of constant terms and $\boldsymbol{x}$ is the vector of the unknowns. From this specification, the first step is to build a matrix $\boldsymbol{T}^{(n)}$, namely *inhibition table* computed as follows:

$$\boldsymbol{T}^{(n)}=\left[\begin{array}{ccccc|ccccc} \frac{1}{a_{1,1}} & 0 & \dots & \dots & 0 & 1 & \frac{a_{2,1}}{a_{1,1}} & \dots & \dots & \frac{a_{n,1}}{a_{1,1}} \\ 0 & \frac{1}{a_{2,2}} & \dots & \dots & 0 & \frac{a_{1,2}}{a_{2,2}} & 1 & \dots & \dots & \frac{a_{n,2}}{a_{2,2}} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & \dots & \frac{1}{a_{n-1,n-1}} & 0 & \vdots & \vdots & \dots & 1 & \frac{a_{n,n-1}}{a_{n-1,n-1}} \\ 0 & \dots & \dots & 0 & \frac{1}{a_{n,n}} & \frac{a_{1,n}}{a_{n,n}} & \dots & \dots & \frac{a_{n-1,n}}{a_{n,n}} & 1 \end{array}\right]$$

where $a_{i,j}$ are the elements of $\boldsymbol{A}$. The matrix $\boldsymbol{T}^{(n)}$ can be seen as a decomposition of the original problem into n sub-problems (one for each row).

IMe applies a *fundamental formula* to iteratively reduce by one the number of rows and columns in $\boldsymbol{T}^{(n)}$, so that at each step (usually called *level*) l (with $l = n \dots 1$), $\boldsymbol{T}^{(l)}$ represents the original problem decomposed into l sub-problems:

$$\begin{aligned} t_{i,j}^{(l-1)} &= (t_{i,j}^{(l)} - t_{l,j}^{(l)} \cdot t_{i,n+l}^{(l)}) \cdot h_i^{(l)}, \\ & l = n \dots 1,\ i = 1 \dots l-1,\ j = 1 \dots n+l-1 \end{aligned} \tag{1}$$

where $h_i^{(l)}$ is the i^{th} element of the *auxiliary vector* $\boldsymbol{h}^{(l)}$, which must be recomputed at each level as well:

$$h_i^{(l)} = \frac{1}{1 - t_{l,n+i}^{(l)} \cdot t_{i,n+l}^{(l)}}, \quad l = n \dots 2,\ i = 1 \dots l-1.$$

Note that this entails (as shown in Fig. 1) having, at each level, any element ($t_{i,j}^{(l-1)}$) of the inhibition table recomputed employing only its previous value ($t_{i,j}^{(l)}$), the elements on the last row ($t_{l,j}^{(l)}$) and column ($t_{i,n+l}^{(l)}$) of the previous inhibition table, and the corresponding auxiliary quantity ($h_i^{(l)}$). Again, for the computation of $\boldsymbol{h}^{(l)}$, only the elements in the last row ($t_{l,n+i}^{(l)}$) and column ($t_{i,n+l}^{(l)}$) of the inhibition table are needed.

In order to actually compute the system's solution, the vector of constant terms $\boldsymbol{b}$ and the solution vector $\boldsymbol{x}$ are initialized and updated at each level as shown in Table I. We underline that – as prescribed by the values assumed by the index i – at each level l, only the first $l-1$ elements of $\boldsymbol{b}$ and last $l-n$ of $\boldsymbol{x}$ are modified. At the end of all iterations – i.e., at level $l = 1$, when $\boldsymbol{T}^{(1)}$ matrix has only one row and n columns – the vector $\boldsymbol{x}^{(1)}$ hosts the solution of the linear system.

A more detailed description of the method's motivations and proof can be found in [8], whereas [1] clarifies IMe working principles through the pseudocode of a serial implementation.

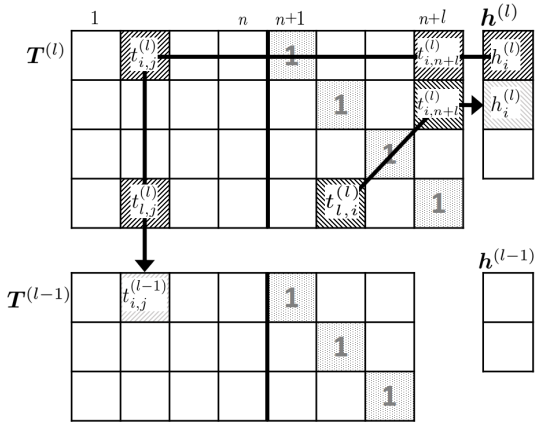


Figure 1: Graphical visualization of the fundamental formula and the computation of the auxiliary quantities.

III. High Performance Implementation

In the following, we focus on the execution of IMe on a network of N computing nodes relying on the message passing model. Obviously, we must split the computation while minimizing the amount of data exchanged to limit the communication overheads. The structure of IMe is relatively easy to parallelize: given the fundamental formula in Equation (1), if the last row $\boldsymbol{t}^{(l)}_{l,*}$ and column $\boldsymbol{t}^{(l)}_{*,n+l}$ of $\boldsymbol{T}^{(l)}$ are made available to all nodes, the computation of each element of $\boldsymbol{T}^{(l-1)}$ can be conducted independently. This observation enables various parallelization schemas for IMe: the elements of $\boldsymbol{T}^{(l)}$ can be distributed row-wise, column-wise or block-wise. Nonetheless, for the purposes of this work we focus on column-wise parallelization because – as will be clear in the following – it is the best solution to allow the fault tolerant enhancement.

The work [1] presents a master/slave algorithm which intrinsically suffers from many synchronization points. In the following, we propose an enhanced column-wise implementation of IMe, depicted in Algorithm 1, where all nodes perform the same operations on different data and no central coordination is present.

After an initialization step, where the inhibition table is computed from the matrix $\boldsymbol{A}$ and scattered (line 2), all processors iterate over the levels, exchanging the last column and row with each other. As we chose a column-wise parallelization, the last column is available on a single computing node and can be broadcasted to all the others (line 8), whereas the last row is scattered across all processes, thus requiring a more complex communication schema (AllGather function in line 9).

After this communication, the last row and column are sufficient to allow each processor to independently compute the auxiliary vector (line 20) and apply the fundamental formula to its portion $\boldsymbol{T'}$ of columns of $\boldsymbol{T}$ (line 22).

The computation of the auxiliary vector is actually the exact same operation repeated on all nodes. With respect to an more centralized implementation, where the master is in charge of $\boldsymbol{h}^{(l)}$ computation and distribution [1], this redundancy inevitably increases the number of total flops. Nonetheless, it reduces the number of messages and frees all processors from the burden of synchronizing with a central coordinator at each level. It also reduces the risks connected to the presence of a single coordinator, which represents a single point of failure during the computation.

Algorithm 1 makes only one exception to an embarrassingly-parallel computation: a single node is actually responsible for the computation of the solution by

Table I: IMe prescribed steps to compute the system's solution

	Initialization	Update
$\boldsymbol{b}$	$b^{(n)}_i = \begin{cases} b_n, & i = n \\ b_i - t^{(n)}_{n,n+i} b_n, & \text{o/w} \end{cases}$	$b^{(l-1)}_i = b^{(l)}_i - t^{(l-1)}_{l-1,n+i} b^{(l)}_l,$ $i = 1 \dots l-2$
$\boldsymbol{x}$	$x^{(n)}_i = \begin{cases} t^{(n)}_{n,i} \cdot b_n, & i = n \\ 0, & \text{o/w} \end{cases}$	$x^{(l-1)}_i = x^{(l)}_i + t^{(l-1)}_{l-1,i} b^{(l)}_l,$ $i = l-1 \dots n$

Algorithm 1 Parallel IMe factorization with fault resilience.

Input: $\boldsymbol{A}$, matrix of coefficients; $\boldsymbol{b}$ vector of constant terms.
Output: $\boldsymbol{x}$ solution vector.

1: **procedure** IMeHP($\boldsymbol{A}, \boldsymbol{b}$)
2: $\boldsymbol{T}, \boldsymbol{x}, \boldsymbol{b} \leftarrow$ InitIMe($\boldsymbol{A}$)
3: ▷ Each processor gets a subset $\boldsymbol{T'}$ of columns in $\boldsymbol{T}$:
4: $\boldsymbol{T'} \leftarrow$ scatterColumns($\boldsymbol{T}, root = 0$)
5: **for** $l \leftarrow n \dots 2$ **do**
6: $q' \leftarrow$ processor holding last column of $\boldsymbol{T}$
7: ▷ processors exchange last column and row of $\boldsymbol{T}$:
8: $\boldsymbol{t}_{*,n+l} \leftarrow$ Broadcast($t_{*,n+l}$,root=q')
9: $\boldsymbol{t}_{l,*} \leftarrow$ AllGather($t_{l,*}$)
10: ▷ only one processor computes the solution:
11: **if** $rank == 0$ **then**
12: **for** $i \leftarrow l \dots n$ **do**
13: $x_i \leftarrow x_i + t_{l,i} b_l$
14: **end for**
15: **end if**
16: **for** $i \leftarrow 1 \dots l-1$ **do**
17: **if** $rank == 0$ **then**
18: $b_i \leftarrow b_i - t_{l,n+i} b_l$
19: **end if**
20: $h_i = 1/(1 - t_{i,n+l} * t_{l,n+i})$
21: **for each** $\boldsymbol{t}_{*,j} \in \boldsymbol{T'}$ **do**
22: **if** $j == i$ **or** ($j \geq l$ **and** $j \neq n+i$) **then**
23: ▷ apply the fundamental formula:
24: $t_{i,j} \leftarrow (t_{i,j} - t_{l,j} \cdot t_{i,n+l}) \cdot h_i$
25: **end if**
26: **end for**
27: **end for**
28: **end for**
29: **return** $\boldsymbol{x}$
30: **end procedure**

iteratively updating vectors $\boldsymbol{b}$ and $\boldsymbol{x}$ (lines 18 and 13).

This choice, compliant with what is done by other linear algebra libraries [12], comes from the usual need to finally collect the solution into a single node to use it. To improve fault resilience, other implementations are possible, where the computation of $\boldsymbol{b}$ and $\boldsymbol{x}$ (which is rather small in terms of flops) is performed by different nodes at the same time. Nonetheless, as we will explain in the following, even in the case of a fault involving the node which is computing the solution, it is possible to restore its state at the cost of a minimal overhead.

A. Algorithm complexity

We consider the theoretical complexity of the proposed approach in terms of flops, memory occupation *mo*, number M and volume V of exchanged messages.

As regards the flops, we can observe that, during initialization, although $\boldsymbol{T}$ has $2n^2$ elements, only half of them is computed from the elements of $\boldsymbol{A}$. The others are initially set to 1 or 0. Thus, the initialization of IMe only requires $\mathit{flops}_{\mathit{IMe\_init}} = n^2$. Then, the total number of flops to compute the solution are due to the four contributes of $\boldsymbol{b}$, $\boldsymbol{x}$, $\boldsymbol{h}$ and $\boldsymbol{T}$ for each level. In particular, the contributes of $\boldsymbol{b}$ and $\boldsymbol{x}$ (lines 18 and 13) can be computed together because in each level, the two flops of $\boldsymbol{b}_i$ are executed only for $i = 1 \dots l-1$, whereas those of $\boldsymbol{x}_i$ only for $i = l \dots n$. This is equivalent to a two-flops operation executed n times in each level. As we have n levels, we can conclude $\mathit{flops}_{\boldsymbol{b}} + \mathit{flops}_{\boldsymbol{x}} = 2n^2$. The computation of $\boldsymbol{h}$ (line 20) is given by three flops repeated $l-1$ times for each of the n levels, and this operation is repeated on all the N computing processors involved i.e., $\mathit{flops}_{\boldsymbol{h}} = 3N\sum_{l=1}^{n}(l-1)$. Finally (as in [1]) we perform the three flops of the fundamental formula only for the $l \times n$ elements that can actually result modified in each of the $n-1$ levels following the initialization. Thus, $\mathit{flops}_{\boldsymbol{T}} = 3n\sum_{l=1}^{n-1} l$.

The total flops are therefore: $\mathit{flops}_{\mathit{IMe}} = \frac{3}{2}n^3 + \frac{3}{2}n^2(N+1) + \frac{3}{2}Nn$. This value is comparable to that of popular algorithms for linear system resolution such as Gaussian elimination and LU decomposition i.e., $\frac{2}{3}n^3 + O(n^2)$.

The memory utilization is limited to $\mathit{mo}_{\mathit{IMe}} = 2n^2 + 3Nn + 2n$ i.e., the dimension of $\boldsymbol{T}$, plus that of $\boldsymbol{h}$, $\boldsymbol{t}_{*,n+l}$ and $\boldsymbol{t}_{l,*}$ (which must be stored on all N computing nodes), plus the dimension of $\boldsymbol{b}$ and $\boldsymbol{x}$.

As regards the messages exchanged, during the initial phase the coordinator scatters the meaningful elements of $\boldsymbol{T}$, generating $(N-1)$ messages and a volume of n^2 floating point values. Then, at each level, the algorithm states that:

- the node that is in charge of the last column $\boldsymbol{t}_{*,n+l}$ broadcasts it to all the other nodes (line 8), which implies $M_{\boldsymbol{t}_{*,n+l}} = (N-1)n$ messages generating a volume of $V_{\boldsymbol{t}_{*,n+l}} = (N-1)\sum_{l=1}^{n} l$;
- all the nodes exchange their portion of the last row (line 9) entailing $M_{\boldsymbol{t}_{l,*}} = nN(N-1)$ and $V_{\boldsymbol{t}_{l,*}} = n^2$.

The latter operation (AllGather in line 9) is rather time-consuming in general because it requires each node to synchronize with all the others. Nonetheless, several more performing strategies than a set of $nN(N-1)$ point-to point communications are possible. All modern MPI libraries adopt more performing implementations: depending on various parameters like the message size and the number of involved processors, the library may choose different algorithms at runtime, such as binomial dissemination, recursive-doubling exchange, or ring all-to-all broadcast [13]. All these strategies significantly reduce the messages exchanged in practice. However, disregarding MPI library optimizations, the theoretical total number and volume of messages exchanged is:

$$\begin{aligned} M_{\mathit{IMe}} &= nN^2 + N - n - 1 \\ V_{\mathit{IMe}} &= \tfrac{n^2}{2}(N+3) + \tfrac{n}{2}(1-N) \end{aligned} \tag{2}$$

When compared to the complexity of the previous parallel implementation [1], this solution shows a slight increase of flops and *mo* due to the computation and storing of the auxiliary vector $\boldsymbol{h}$ on all N nodes. Nonetheless, as $\boldsymbol{h}$ is very small and requires very little computation w.r.t. the $\boldsymbol{T}$, such increase does not significantly affect the performance. On the other hand, besides the great advantage of eliminating the single point of failure, this version overcomes some of the shortcomings of [1] by limiting the sychronization points: the total number of messages is no longer of the order $O(n^2)$ as it was in [1].

IV. Enhancing the HPC Inhibition Method with tolerance to multiple faults

Our hard error resilient enhancement of the parallel IMe – denoted with IMeMFT in the following – is based on a checksum technique to encode the state of the inhibition table and store it in the memory of additional processors [14]. Let us consider a column-wise parallelization where each processor receives just one column for the moment. Taking inspiration from Reed-Solomon coding in the field of RAID-like systems [15], if any m columns of $\boldsymbol{T}^{(l)}$ are involved in a fail-stop (and therefore no longer accessible), their values can be recomputed if m checksum columns have been conveniently defined. We therefore extend the inhibition table with the *checksum matrix* $\boldsymbol{S}^{(l)}$ as follows:

$$\boldsymbol{\Phi}^{(l)} = [\boldsymbol{S}^{(l)} | \boldsymbol{T}^{(l)}] = \left[\begin{array}{ccc|ccc} s_{1,1} & \dots & s_{1,m} & t_{1,1} & \dots & t_{1,n+l} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ s_{n,1} & \dots & s_{n,m} & t_{n,1} & \dots & t_{n,n+l} \end{array}\right]$$

According to [16], if the checksum matrix is computed as

$$\boldsymbol{S}^{(l)} = \boldsymbol{T}^{(l)} \cdot \boldsymbol{\Lambda}^{(l)}, \tag{3}$$

where $\boldsymbol{\Lambda}^{(l)}$ is a $(n+l) \times m$ matrix suitably conceived for the purpose (such that any square sub-matrix of $\boldsymbol{\Lambda}^{(l)}$ is non-singular), then in case of a fail-stop involving any m columns of $\boldsymbol{\Phi}^{(l)}$, their values can be recomputed by solving a linear system for each of its l rows as follows.

$$\left\{\begin{array}{lll} s_{i,1}^{(l)} = & \sum_{k=1}^{n+l} \lambda_{k,1} t_{i,k}^{(l)} & \\ \vdots & \vdots & \forall i = 1 \dots l \\ s_{i,m}^{(l)} = & \sum_{k=1}^{n+l} \lambda_{k,m} t_{i,k}^{(l)} & \end{array}\right.$$

Every such system has m equations and m unknowns (corresponding to the elements of the failed columns).

The application of this checksum technique to IMe comes with a notable advantage:

Lemma IV.1. *Given $\boldsymbol{S}^{(l)}$ checksum matrix of the inhibition table $\boldsymbol{T}$ at level l, the application of the fundamental formula (Eq. 1) to $\boldsymbol{S}^{(l)}$ produces a matrix $\boldsymbol{S}^{(l-1)}$, which is the checksum matrix of $\boldsymbol{T}^{(l-1)}$.*

In order to prove the correctness of this lemma, we have to demonstrate that each element of $\boldsymbol{S}^{(l-1)}$, checksum matrix of $\boldsymbol{T}^{(l-1)}$, can be computed with the fundamental formula from the elements of $\boldsymbol{\Phi}^{(l)}$ i.e.,

$$s_{i,j}^{(l-1)} = \sum_{k=1}^{n+l-1} \lambda_{k,j} t_{i,k}^{(l-1)} \stackrel{?}{=} (s_{i,j}^{(l)} - s_{l,j}^{(l)} \cdot t_{i,n+l}^{(l)}) \cdot h_i^{(l)}, \quad l = n \dots 1, i = 1 \dots l-1, j = 1 \dots m.$$

Proof.

$$\sum_{k=1}^{n+l-1} \lambda_{k,j} t_{i,k}^{(l-1)} = \sum_{k=1}^{n+l-1} \lambda_{k,j} h_i^{(l)} (t_{i,j}^{(l)} - t_{l,j}^{(l)} t_{i,n+l}^{(l)}) =$$

$$= h_i^{(l)} \sum_{k=1}^{n+l-1} (\lambda_{k,j} t_{i,j}^{(l)}) - h_i^{(l)} t_{i,n+l}^{(l)} \sum_{k=1}^{n+l-1} (\lambda_{k,j} t_{l,j}^{(l)}) =$$

$$= h_i^{(l)} (s_{i,j}^{(l)} - \cancel{\lambda_{n+l,j} t_{i,n+l}^{(l)}} - t_{i,n+l}^{(l)} s_{l,j}^{(l)} + \cancel{t_{i,n+l}^{(l)} \lambda_{n+l,j}} \cancel{t_{l,n+l}^{(l)}}^{1})$$

We can observe that, since $t_{l,n+l}^{(l)}$ is the last entry of the last column of $\boldsymbol{T}^{(l)}$, its value is always 1 as a consequence of how $\boldsymbol{T}^{(n)}$ was initially built. This entails that, the second and last contributions of the sum eliminates each other, and we can conclude that Lemma IV.1 is demonstrated. □

Lemma IV.1 is crucial for fault tolerance because it allows to calculate the checksums through Eq. (3) only once at the beginning of the computation (i.e., only for $\boldsymbol{T}^{(n)}$) and then the computing nodes hosting the columns of $\boldsymbol{S}^{(l)}$ can operate on them using the fundamental formula (just as all the others do on the elements of $\boldsymbol{T}$). The columns computed in this way will continue to hold the checksums of $\boldsymbol{T}^{(l)}$ for any following level. The property stated by Lemma IV.1 is also the reason for our choice of a column-wise partitioning. In case of row-wise parallelization indeed, Lemma IV.1 no longer holds.

Furthermore, so far we considered the case of each processor hosting a single column, but as already explained in [1], a convenient checksum computation and assignment can protect from faults even when multiple columns are hosted on each node. In general – assuming that all nodes can host up to the same number of columns – the employment of C additional nodes to maintain the checksums can guarantee from up to C simultaneous hard faults located on any of the $N + C$ processors. In order to clarify the approach, consider the system:

$$\begin{cases} x_1 + & x_2 - & x_3 + & 2x_4 = & 25 \\ -x_1 + & x_2 + & x_3 + & x_4 = & 4 \\ 2x_1 + & 2x_2 - & x_3 - & x_4 = & -7 \\ 2x_1 + & 3x_2 + & 2x_3 - & 2x_4 = & -1 \end{cases} \quad (4)$$

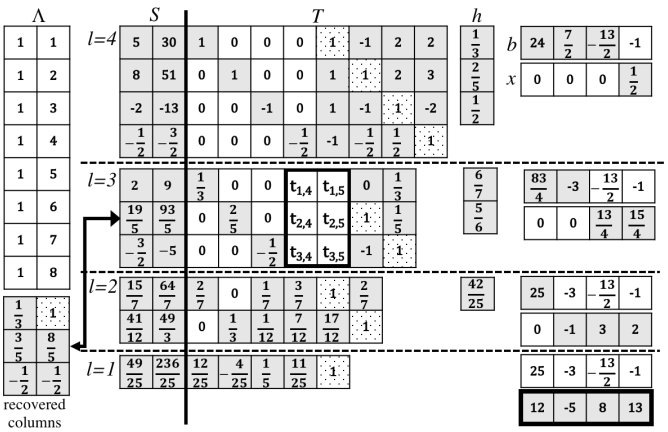


Figure 2: IMe applied to example in (4) with fault resilience to up to two faults.

Fig. 2 highlights the subsequent steps of IMe as well as the values assumed by vectors $\boldsymbol{b}$ and $\boldsymbol{x}$ at each level. Imagine to partition the work such that each processor receives exactly one column. To guarantee resilience to up to $m = 2$ faults, the checksum matrix $\boldsymbol{S}^{(n)}$ is computed with Eq. 3 only once at the beginning of the execution. Then, at each iteration, $\boldsymbol{S}^{(l)}$ is computed through the fundamental formula. Now, consider the case of a fail-stop that causes the loss of columns 4 and 5 of $\boldsymbol{T}^{(3)}$ as highlighted in Fig. 2. Their values can be recovered by simply solving the three linear systems (one for each row of $\boldsymbol{T}^{(3)}$):

$$\begin{cases} 2 = \frac{1}{3} + t_{1,4} + t_{1,5} + \frac{1}{3} \\ 9 = \frac{1}{3} + 4t_{1,4} + 5t_{1,5} + 7 \cdot \frac{1}{3} \end{cases}$$

$$\begin{cases} \frac{19}{5} = \frac{2}{5} + t_{2,4} + t_{2,5} + 1 + \frac{1}{5} \\ \frac{93}{5} = 2 \cdot \frac{2}{5} + 4t_{2,4} + 5t_{2,5} + 6 + 7 \cdot \frac{1}{5} \end{cases}$$

$$\begin{cases} -\frac{3}{2} = -\frac{1}{2} + t_{3,4} + t_{3,5} - 1 + 1 \\ -5 = -3 \cdot \frac{1}{2} + 4t_{3,4} + 5t_{3,5} - 6 + 7 \end{cases}$$

This example underlines the advantages of IMe when solving linear systems on hardware subject to frequent hard errors. Traditional C/R mechanisms save the state of the computation at specific predefined intervals. When a fail occur, the recovery system is in charge of performing a rollback, so that the computation can restart from the latest saved checkpoint. Our approach comes with the great advantage of being able to efficiently maintain a continuous checksum of each step of the computation. The encoded version of the state is not computed at specific predefined intervals, but rather continuously updated at runtime without any additional communication or synchronization points between the computing nodes. Hence, in case of failure no rollback is needed.

A. Complexity of Multiple Faults Tolerant IMe

The proposed approach states that an additional $l \times m$ matrix, $\boldsymbol{S}^{(l)}$, must be maintained at each level in order to guarantee the recovery from failures involving at most m columns.

The initialization of $\boldsymbol{S}^{(n)}$ must be performed through Eq. (3) (with $l = n$). Each of the $n \times m$ elements of $\boldsymbol{S}^{(n)}$ requires the computation of $2n$ multiplications and $2n-1$ sums. Therefore, in order to initialize the fault tolerant version of IMe $nm(4n-1)$ additional flops are required. Then, the fundamental formula is applied to compute the elements of $\boldsymbol{S}^{(l)}$ for all the remaining levels, generating $3m\sum_{l=1}^{n}(l-1)$ additional flops. Therefore, adding fault tolerance does not affect the order of magnitude of the algorithm's complexity:

$$\textit{flops}_{\textit{IMeMFT}} = \frac{3}{2}n^3 + \frac{n^2}{2}(3N + 5m + 3) - \frac{n}{2}(m + 3N)$$

Similarly, fault tolerance makes the memory occupation increase by just the dimension of $\boldsymbol{S}$:

$$\textit{mo}_{\textit{IMeMFT}} = \textit{mo}_{\textit{IMe}} + mn = 2n^2 + 3nN + 2n + mn$$

As regards the number and volume of messages, they are influenced by the fact that there are C additional nodes to maintain the m additional columns. Depending on the specific implementation, the parameters m and C might be related to each other. For example, if we adopt a balanced distribution of the columns of $\boldsymbol{T}$ along the nodes, we will end up with a $2n/N$ columns on each node. Any of the C additional nodes will be able to host the same number of checksum columns. Therefore in that case, the number m of columns that can be recovered from failure is $m = C \times 2n/N$. Nonetheless, in order to provide a more general, implementation-independent study of complexity we hereby consider the two parameters m and C as not related.

Considering that any communication of IMeMFT involves $Q = N + C$ instead of just N nodes, the number and volume of exchanged messages can be calculated as follows.

$$M_{\textit{IMeMFT}} = (QN + Q - 2)n + Q - 1$$
$$V_{\textit{IMeMFT}} = \frac{n^2}{2}(Q + 3) + \frac{n}{2}(1 - Q) + mn$$

where the only significant difference w.r.t. Eq. (2) is in the volume of messages. Indeed, during initialization, the columns to be scattered are m more than IMe (thus mn additional floating point values are sent).

V. Evaluation

In order to test the proposed approach, we employ CRESCO5 [17] and Marconi-A2 [18], two HPC systems funded by ENEA and Cineca consortium, respectively. CRESCO5 is composed of 640 2.40GHz Intel Xeon cores. Each node is equipped with 16 cores (2 sockets with 8 cores each), 64GB RAM and a 40Gb/s InfiniBand QDR interface. The employed A2 partition of Marconi is composed of 3,600 computing nodes, each of which has a 68-cores Intel Xeon Phi7250 (KnightLandings) processor at 1.4 GHz (244,800 cores in total), 16 GB MCDRAM and 96 GB DDR4. The internal Network is a 100Gb/s Intel OmniPath.

One of the most important characteristics of a HPC application is its scalability. Nonetheless, when evaluating a fault tolerant mechanism, another crucial element is the overhead introduced by such feature as a consequence of, for example, the extra flops to compute checksums or maintain checkpoints. We compare IMe with a popular direct method for solving linear systems: Gauss-Jordan Elimination (GJE). Analogously to our approach, GJE iteratively operates over the matrix of coefficients until the exact solution is determined. We implemented both the methods with the Intel MPI library [19]. We consider input linear systems characterized by artificially generated matrices of coefficients, and restrict our attention to dense unstructured invertible matrices of real numbers.

A. Evaluation approach

We separately evaluate the features of our approach by means of three different kinds of tests.

Strong scalability. We evaluate the performance (in terms of clock ticks to compute the solution) of IMe and GJE when solving a linear system of fixed dimension on an increasing number of computing processor. In order to better understand the advantages and weaknesses of our approach, we first compare the performance when no fault tolerant method is implemented, and then we introduce resilience to an increasing number of faults.

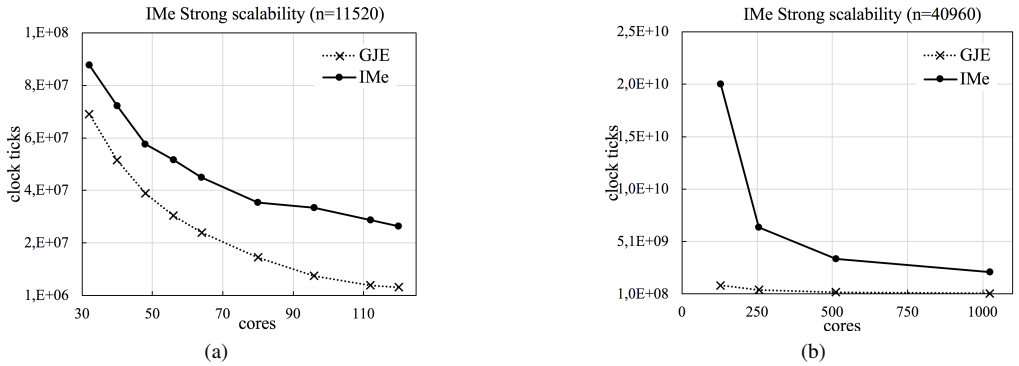
Overhead evaluation. We compare the computing overhead introduced by the two fault tolerant methods w.r.t. their non-resilient implementations when the linear system is solved on different numbers of computing nodes with resilience to increasing faults.

Resilience scalability. Since the rate of tolerance to faults is not fixed for IMeMFT but can be adjusted according to necessity, it is important to understand the relation between such rate of resilience and performance. To this end, we evaluate the computation times to solve the same linear system on the same number of processors, when we increase the number of tolerable faults. This allows to understand how the implementation of different levels of resilience affects the time to compute the solution.

B. Results

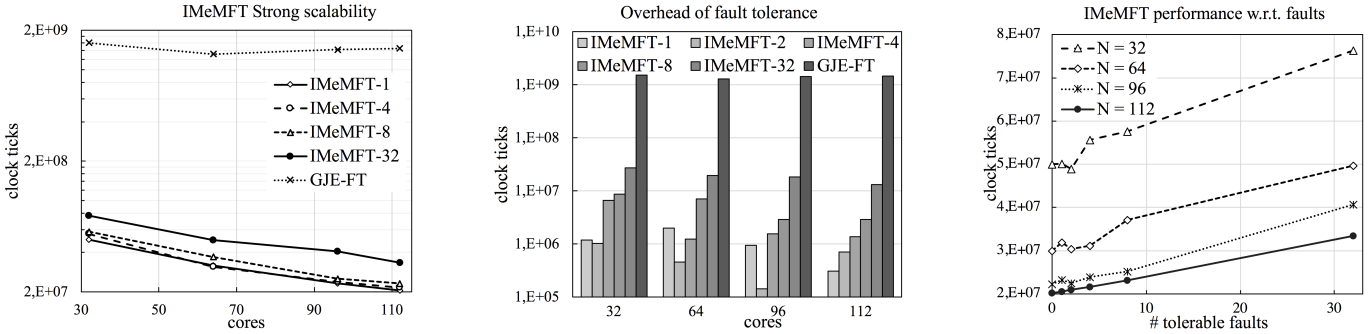
As regards the *Strong scalability* test, Fig. 3 compares the performance of the parallel implementations of IMe and GJE without any checksumming or checkpointing mechanism to ensure fault tolerance. The evaluation of Fig. 3a refers to CRESCO5, whereas that of Fig. 3b required an higher number of computing cores, and was therefore conducted on Marconi-A2. For both the considered matrix dimensions IMe cannot outperform GJE. This is indeed expected, as the latter directly operates on the matrix $\boldsymbol{A}$ of coefficients, whereas IMe works on the inhibition table $\boldsymbol{T}$ (which is initially twice the size of the $\boldsymbol{A}$) and an additional vector $\boldsymbol{h}$.

The advantages of IMe become visible when a fault tolerant mechanism is implemented. In this regard, we enhance GJE with a diskless C/R mechanism: a checksum is computed at each iteration step and stored in the memory of additional processors to provide resilience to multiple fail-stops. Fig. 4a highlights the strong scalability of IMeMFT i.e., the performance trend of the same solving task ($n = 11520$) executed on an increasing number of computing processors compared to that of GJE with C/R ("GJE-FT" series in the plot). The



(a) (b)

Figure 3: Performance comparison between IMe and GJE solving a system with $n = \{11520, 40960\}$ equations when no fault tolerant method is implemented. y-axes in logarithmic scale.



(a) (b) (c)

Figure 4: Comparison of fault tolerant GJE and IMe on strong scalability (Fig. 4a) and overhead introduced (Fig. 4b) conducted on CRESCO5. Fig. 4c reports the performance of IMeMFT for increasing resilience to faults. y-axes in logarithmic scale.

graph clearly shows a desirable decrease of the computing time when the work is subdivided into more nodes . The different series show the clock ticks required to provide resilience to different number of hard errors, e.g. series "IMeMFT-1" refers to single-fault tolerance, "IMeMFT-4" refers to resilience to up to four faults, etc. In this regard, it is also worth noting that – has expected – providing resilience to a higher number of faults inevitably increases the computation time. Nonetheless, the time to compute the solution with IMeMFT is always significantly lower than that of GJE-FT.

Concerning the *Overhead evaluation*, Fig. 4b shows the increment of computation time introduced by the two fault tolerant mechanisms w.r.t. their non-fault tolerant implementations. When solving a system of $n = 11520$ equations on an increasing number of processors, GJE with C/R requires an additional number of clock ticks of almost two orders of magnitude more then IMeMFT. For the sake of fairness, we must underline that our implementation of GJE with C/R is able to recover from a maximum number of fail-stops equal to the number of processors, whereas IMeMFT-4, for example, guarantees from a maximum of 4 anywhere-located hard faults. Nonetheless, the case of 32 computing processors in Fig. 4b shows how an implementation of IMeMFT able to recover from a maximum number of fail-stops equal to the number of processors (the "IMeMFT-32" series) is still able to keep lower overheads w.r.t. GJE-FT.

Finally, regarding the *Resilience scalability*, Fig. 4c highlights how clock ticks of IMeMFT increase when we augment the number of tolerable faults. As GJE-FT always guarantees from the maximum possible number of failures, it is not considered in this test. The relation between performance and resilience shows a desirable linear trend. The graph reports the clock ticks required to solve the same linear system ($n = 11520$) when we increase resilience from 0 (no fault-tolerance) to 32 (i.e., up to 32 fail-stops involving any of the computing processors). Different series show the trends for different number of processors employed to compute the solution. Since for IMeMFT providing resilience to an additional anywhere-located hard error only requires an additional processor (which works on the checksum independently from all the other nodes), the computation time is only slightly affected by the increasing resilience, thus realizing the graceful linear trends in figure.

VI. Related work

Since the linear systems that solve scientific problems are large-scale, HPC parallelization is often an inevitable choice. In this regard many solutions have been proposed to improve the performance of existing approaches [20]–[22]. In particular, Habgood et al. [21] presented an efficient

framework for solving large-scale linear systems by means of a combination of Cramer's rule and Chio's condensation, which was later parallelized in [23]. Agullo et al. [22] combine direct and iterative methods to solve linear systems with sparse matrices on a parallel computing platform. All these works focus on the performance enhancements that linear algebra can obtain through parallelization disregarding the topic of fault tolerance. Whenever extreme-scale systems are employed, the large number of hardware components involved causes the overall system's MTBF to decrease to just few hours, thus making the implementation of resilient applications a crucial task in this framework [24].

C/R has been one of the most popular approaches to fault tolerance for many years [25]. Some works applied C/R at system level i.e., through message passing middlewares that automatically handle the fault without the intervention of the application [26]–[28]; whereas others intervene at application level i.e., providing the developer with functions to periodically dump the relevant state to stable storage [29]–[32]. Lately, Gholami et al. [33] proposed an interesting strategy to combine system- and user-level checkpointing, which exploits the advantages and avoid the shortcoming of both.

Aiming to overcome the C/R bottleneck caused by multiple accesses to disk, Plank et al. [14] developed the *diskless* checkpointing technique, which suggests to store the checkpoints in the memory of dedicated processors after an encoding operation, such as one-dimensional parity, evenodd or Reed-Solomon coding [15]. Although not able to survive failures involving the whole systems, diskless checkpointing provides significant enhancements in several contexts, especially when paired with techniques to improve its scalability [16], [34]. As also underlined in [35], it might anyway exhibit relevant overheads for applications modifying large memory regions between two checkpoints (as for example, when matrix factorization is used to solve linear systems). Taking inspiration from Plank et al.'s work [14], our approach uses Reed-Solomon codes to enable the recovery from multiple faults. Nonetheless, differently from [14], we relay on ABFT [4] to improve scalability.

Thanks to its capability to avoid periodic checkpointing and onerous rollbacks, ABFT is one of the most explored techniques to enhance the performance of fault resilient HPC applications for linear algebra [24]. In particular, it has been successfully applied to matrix-matrix multiplication [36], [37], and eigenvalues computation [38], [39]. Tao et al. [40] and Langou et al. [41] apply ABFT to linear system solvers in case of iterative approximated methods, whereas the works [42], [43] use it to reliably compute the solution through direct methods when the matrix of coefficients has particular properties. The "interpolation-restart" approach by Agullo et al. [44], [45] exploits the properties of the hybrid (i.e. direct/iterative) sparse solver described in [22] to design two resilient solutions based on neighbourhood redundancy: one to recover from single faults and another to survive faults on neighbour processes. The same fault-tolerant approach has been later applied to eigensolvers [46]. Differently from these works, we use ABFT to provide resilience to faults while we iteratively compute the exact solution of linear systems through a direct method, without any restriction on the matrix characteristics.

Several works [5], [47]–[49] adopt ABFT frameworks to provide resilience to *soft errors* (otherwise called *fail-continue* or SDC [3]), i.e. failures inducing incorrect results without causing the computation to abort. Indeed, in the context of linear algebra even a small miscalculation involving a single coefficient can heavily affect the correctness of the final result. A seminal work coping with soft errors in large scale systems is that of Jou et al. [50] which propose a matrix encoding scheme to correct errors in matrix operations. ABFT is exploited in FT-ScaLAPACK [49]: a fault tolerant version of Scalable LAPACK (ScaLAPACK) [12] able to locate and correct miscalculations in Cholesky, QR, and LU factorizations in an on-line fashion (i.e., during the computation, thus to avoid the error accumulation and propagation). In the field of widely used Krylov subspace iterative methods, Chen [51] proposes an efficient online technique to detect soft errors, whereas Jaulmes et al [52] specifically address Detected and Uncorrected Errors (DUE) by means of very simple algebraic relations. Another important work in this framework is that of Du et al. [5], which propose an ABFT technique to handle single soft errors while solving linear systems with dense matrix of coefficients. This work has been later improved to provide resilience to multiple soft errors [6].

Differently from these works, our approach deals with failures causing the processor to stop: a situation often addressed as *hard error* or *fail-stop*. In this regard, ABFT has been applied to High Performance Linpack (HPL) [53], Cholesky [43], QR and LU factorizations [54]. These contributions address the case of a single hard error causing one processor to abort its computation, whereas the technique proposed here is an extension of [1] that is able to deal with multiple errors. The aforementioned "interpolation-restart" method by Agullo et al. was also applied to multiple hard errors in the framework of iterative methods for the solution of sparse linear systems [55]. More related to our work is that by Bouteillers et al. [35] (an evolution of [54]), which handles multiple fail-stops in matrix factorization with a hybrid approach: ABFT is used to continuously maintain a checksum during the computation of the right factor, whereas a novel checkpointing scheme protects the left factor. Compared to that, our approach requires an inferior number of additional processors dedicated to resilience and shows a less complicated checksumming mechanism, which is completely checkpoint-free.

VII. Conclusion and future work

Over the last decade, fault tolerant HPC has been deeply explored to boost the performance of large-scale linear algebra applications and provide resilience to hard and soft errors.

In this work, we show how an existing technique for solving linear systems, namely IMe, can be parallelized by means of a message passing framework, and then enhanced

with a simple, yet effective strategy to provide checkpoint-free ABFT to multiple fail-stops. A theoretical study of complexity (in terms of flops, memory occupation, number and volume of exchanged messages) of the proposed approach is presented. The preliminary performance evaluation conducted on a medium-scale HPC system shows a graceful scalable behaviour. Although IMe cannot outperform GJE when no fault tolerant method is implemented, the tests highlight that the overhead introduced by error resilience is significantly reduced in case of IMe. Furthermore, the computation time increases linearly with the number of faults from which we provide protection.

For the future, we plan to improve the performance of the proposed decentralized approach by combining the message passing with a shared memory model, thus to better exploit the enhancements offered by modern supercomputers. Then, an evaluation of IMe performance w.r.t. the de facto standard ScaLAPACK routines [12], the approach of Du et al. [54] (as regards single faults), and that of Boutiellers et al. [35] (on multiple faults) would be interesting.

Furthermore, in this work, we try to reduce the overhead of maintaining an encoded copy of the computation status. So, we can say that we mostly focus on the performance of a fault-free execution. Another interesting study regards what happens after the failure. In case of IMe, different parallel implementations of the recovery strategy are possible. In near future, we will investigate the performance of these solutions in order to determine which is the best approach to minimize the overhead of recovery.

In this regard, albeit preliminary empirical studies showed promising results, also the practical numerical issues involved in recovering multiple simultaneous fail-stops have to be investigated. For example, the possibility of overflow, underflow and cancellation due to round-off errors, need to be studied to understand to which extent they affect the computed solution and the reconstruction error. The approaches presented in [56] and [57] can be partially exploited for the purpose.

As some studies [58], [59] showed the prevalence of soft errors w.r.t. fail-stops on some architectures, the application of IMeMFT to provide robustness to such silent corruptions of the output is another interesting matter of future work.

Ultimately, we will also study the applicability of IMe to other linear algebra problems such as matrix inversion or, more ambitiously, eigenvalues computation and singular value decomposition.

Acknowledgment

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 HPC infrastructure is funded by ENEA, and by Italian and European research programmes. The authors want to acknowledge also the Cineca Consortium for the availability of the HPC resources of Marconi and the technical support.

References

[1] M. Artioli, D. Loreti, and A. Ciampolini, "Fault tolerant high performance solver for linear equation systems," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2019, pp. 113–122.

[2] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, Oct 2010.

[3] S. S. Mukherjee, J. S. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *11th International Conference on High-Performance Computer Architecture*. IEEE Computer Society, 2005, pp. 243–247.

[4] K. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Computers*, vol. 33, no. 6, pp. 518–528, 1984.

[5] P. Du, P. Luszczek, and J. J. Dongarra, "High performance dense linear system solver with soft error resilience," in *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*. IEEE Computer Society, 2011, pp. 272–280.

[6] P. Du, P. Luszczek, and J. Dongarra, "High performance dense linear system solver with resilience to multiple soft errors," *Procedia Computer Science*, vol. 9, pp. 216 – 225, 2012, proceedings of the International Conference on Computational Science, ICCS 2012.

[7] F. Ciampolini, "Un metodo di soluzione dei circuiti lineari," *L'Elettrotecnica*, vol. L, no. 10, 1963.

[8] F. Filippetti and M. Artioli, "IMe: 4-term formula method for the symbolic analysis of linear circuits," *IEEE Trans. on Circuits and Systems*, vol. 51-I, no. 3, pp. 526–538, 2004.

[9] 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*, ser. WIT Transactions on Engineering Sciences. WIT Press, 2001, vol. 31, pp. 147–162.

[10] ——, "IME: implementations for linear system and electric circuit analysis," in *Software for Electrical Engineering Analysis and Design V*, ser. WIT Transactions on Engineering Sciences, U. K. Wessex Institute of Technology, Ed. WIT Press, 2001, vol. 31, pp. 163–172.

[11] M. Artioli, "Symbolic techniques addressed to electric circuit analysis and diagnosis," Ph.D. dissertation, Università di Bologna, March 2001.

[12] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[13] H. Zhou, V. Marjanovic, C. Niethammer, and J. Gracia, "A bandwidth-saving optimization for MPI broadcast collective operation," in *ICPP Workshops*. IEEE Computer Society, 2015, pp. 111–118.

[14] J. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 10, pp. 972–986, 1998.

[15] J. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems," *Softw., Pract. Exper.*, vol. 27, no. 9, pp. 995–1012, 1997.

[16] Z. Chen, "Scalable techniques for fault tolerant high performance computing," Ph.D. dissertation, Knoxville, TN, USA, 2006, aAI3214395.

[17] (2019, Apr.) CRESCO: Centro computazionale di RicErca sui Sistemi COmplessi. [Online]. Available: http://www.cresco.enea.it

[18] (2019, Apr.) Marconi, Cineca HPC system. [Online]. Available: http://www.hpc.cineca.it/hardware/marconi

[19] (2019, Apr.) Intel MPI library. [Online]. Available: https://software.intel.com/en-us/mpi-library

[20] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki, "A survey of recent developments in parallel implementations of gaussian elimination," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 5, pp. 1292–1309, 2015.

[21] K. Habgood and I. Arel, "A condensation-based application of cramers rule for solving large-scale linear systems," *Journal of Discrete Algorithms*, vol. 10, pp. 98 – 109, 2012.

[22] E. Agullo, L. Giraud, A. Guermouche, and J. Roman, "Parallel hierarchical hybrid linear solvers for emerging computing platforms," *Comptes Rendus Mécanique*, vol. 339, no. 2, pp. 96 – 103, 2011, high Performance Computing.

[23] R. Armistead and F. Li, "Parallel computing of sparse linear systems using matrix condensation algorithm," in *2011 IEEE Trondheim PowerTech*, June 2011, pp. 1–6.

[24] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*, 1st ed. Springer Publishing Company, Incorporated, 2015.

[25] 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," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[26] G. Burns, R. Daoud, and J. Vaigl, "Lam: An open cluster environment for mpi," in *Proceedings of supercomputing symposium*, vol. 94, 1994, pp. 379–386.

[27] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "A job pause service under LAM/MPI+BLCR for transparent fault tolerance," in *2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007, pp. 1–10.

[28] P. V. Cardoso and P. P. Barcelos, "Definition of an architecture for dynamic and automatic checkpoints on apache spark," in *37th IEEE Symposium on Reliable Distributed Systems, SRDS*, 2018, pp. 271–272.

[29] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of mpi programs," *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '03*, 2003.

[30] G. Bronevetsky, D. J. Marques, K. K. Pingali, R. Rugina, and S. A. McKee, "Compiler-enhanced incremental checkpointing for openmp applications," *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming - PPoPP '08*, 2008.

[31] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Nov 2002, pp. 29–29.

[32] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, Sep 2002.

[33] M. Gholami, F. Schintke, and T. Schütt, "Checkpoint scheduling for shared usage of burst-buffers in supercomputers," *Proceedings of the 47th International Conference on Parallel Processing Companion - ICPP '18*, 2018.

[34] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault tolerant high performance computing by a coding approach," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[35] A. Bouteiller, T. Hérault, G. Bosilca, P. Du, and J. J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy," *TOPC*, vol. 1, no. 2, pp. 10:1–10:28, 2015.

[36] 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, 2008.

[37] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen, "Fault tolerant matrix-matrix multiplication: correcting soft errors on-line," in *Proceedings of the second workshop on Scalable algorithms for large-scale systems, ScalA@SC 2011, Seattle, WA, USA, November 14, 2011*, V. N. Alexandrov, A. Geist, and J. J. Dongarra, Eds. ACM, 2011, pp. 25–28.

[38] Y. Jia, P. Luszczek, and J. J. Dongarra, "Hessenberg reduction with transient error resilience on gpu-based hybrid architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, pp. 653–662.

[39] A. Schöll, C. Braun, M. A. Kochte, and H. Wunderlich, "Efficient algorithm-based fault tolerance for sparse matrix operations," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*. IEEE Computer Society, 2016, pp. 251–262.

[40] D. Tao, "Fault tolerance for iterative methods in high-performance computing," Ph.D. dissertation, UC Riverside, 2018.

[41] J. Langou, Z. Chen, G. Bosilca, and J. J. Dongarra, "Recovery patterns for iterative methods in a parallel unstable environment," *SIAM J. Scientific Computing*, vol. 30, no. 1, pp. 102–116, 2007.

[42] J. Chen, X. Liang, and Z. Chen, "Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus," in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, pp. 993–1002.

[43] D. Hakkarinen, P. Wu, and Z. Chen, "Fail-stop failure algorithm-based fault tolerance for cholesky decomposition," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 5, pp. 1323–1335, 2015.

[44] E. Agullo, L. Giraud, and M. Zounon, "On the resilience of parallel sparse hybrid solvers," in *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*. IEEE Computer Society, 2015, pp. 75–84.

[45] M. Zounon, "On numerical resilience in linear algebra. (conception d'algorithmes numériques pour la résilience en algèbre linéaire)," Ph.D. dissertation, University of Bordeaux, France, 2015.

[46] E. Agullo, L. Giraud, P. Salas, and M. Zounon, "Interpolation-restart strategies for resilient eigensolvers," *SIAM J. Scientific Computing*, vol. 38, no. 5, 2016.

[47] P. Wu, N. DeBardeleben, Q. Guan, S. Blanchard, J. Chen, D. Tao, X. Liang, K. Ouyang, and Z. Chen, "Silent data corruption resilient two-sided matrix factorizations," *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '17*, 2017.

[48] E. Coleman, M. Sosonkina, and E. Chow, "Fault tolerant variants of the fine-grained parallel incomplete lu factorization," in *Proceedings of the 25th High Performance Computing Symposium*, ser. HPC '17. San Diego, CA, USA: Society for Computer Simulation International, 2017, pp. 15:1–15:12.

[49] P. Wu and Z. Chen, "Ft-scalapack: Correcting soft errors on-line for scalapack cholesky, qr, and lu factorization routines," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, 2014, pp. 49–60.

[50] Jing-Yang Jou and J. A. Abraham, "Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures," *Proceedings of the IEEE*, vol. 74, no. 5, pp. 732–741, May 1986.

[51] Z. Chen, "Online-abft: an online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, Eds. ACM, 2013, pp. 167–176.

[52] L. Jaulmes, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, and M. Valero, "Exploiting asynchrony from exact forward recovery for DUE in iterative solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, J. Kern and J. S. Vetter, Eds. ACM, 2015, pp. 53:1–53:12.

[53] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High performance linpack benchmark: A fault tolerant implementation without checkpointing," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 162–171.

[54] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 225–234.

[55] E. Agullo, L. Giraud, A. Guermouche, J. Roman, and M. Zounon, "Numerical recovery strategies for parallel resilient krylov linear solvers," *Numerical Lin. Alg. with Applic.*, vol. 23, no. 5, pp. 888–905, 2016.

[56] Z. Chen and J. J. Dongarra, "Numerically stable real number codes based on random matrices," in *Computational Science - ICCS 2005, 5th International Conference, Atlanta, GA, USA, May 22-25, 2005, Proceedings, Part I*, ser. Lecture Notes in Computer Science, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, Eds., vol. 3514. Springer, 2005, pp. 115–122.

[57] ——, "Condition numbers of gaussian random matrices," *CoRR*, vol. abs/0810.0800, 2008. [Online]. Available: http://arxiv.org/abs/0810.0800

[58] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. Rogers, "A large-scale study of soft-errors on GPUs in the field," in *2016 IEEE International Symposium on High Performance Computer Architecture,HPCA*. IEEE Computer Society, 2016, pp. 519–530.

[59] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault site pruning for practical reliability analysis of GPGPU applications," in *51st Annual IEEE/ACM International Symposium on Microarchitecture,MICRO*. IEEE Computer Society, 2018, pp. 749–761.