Error Resilience of Three GMRES Implementations under Fault Injection

José A. Moríñigo[1], Andrés Bustos, Rafael Mayo-García

Dept. de Tecnología, CIEMAT
Avda. Complutense 40, Madrid 28040, Spain

[1] Corresponding author. E-mail: josea.morinigo@ciemat.es

Abstract. The resilience behaviour of three GMRES prototyped implementations (with Incomplete LU, Flexible, and randomized-SVD -based preconditioners) has been analized with a soft errors injection approach. A low-level fault injector is inserted into the GMRES solvers, which randomly select locations in the program to inject the fault across multiple executions. This fault injection approach combines the configurability of high-level and the accuracy of low-level techniques at the same time, so the effect of faults may be closely emulated. In order to gather enough statistical data, a set of eighteen sparse matrix-based linear systems $Ax=b$ has been solved with these GMRES implementations in the injection experiments and monitored. The results of this prototype-based fault injection suggest an improved error resilience behaviour of the randomized-SVD -based preconditioned GMRES version in many of the analyzed matrices, which points out to its interest in supercomputing applications where silent errors are more prominent.

Keywords: Randomized SVD, preconditioned GMRES, LLFI, fault injection, iterative solvers.

1- INTRODUCTION

Numerical solvers need to improve their tolerance for faults and failures in the operational environment set by a supercomputing facility. As the exascale era approaches and HPC systems become larger and more complex, recreation of failure scenarios becomes harder but necessary to scientific codes developers. Thus, fault injection tools are important to evaluate the behaviour of algorithmic implementations, by deploying an emulation-based computing environment, then making possible to guide decisions about the more convenient way to design a solver to be used with a class of problems. These tools imply the capacity to inject faults

mimicking the real scenario under some assumptions, as well as gathering the statistics of error and failures to monitor their effects. It should be stressed that the focus of this work is on software-implemented fault injection, which permits to emulate hardware faults at the software layer [1-3] (i.e., silent data corruption provoked by transient bit-flips in memory cells) following a low level approach, that is by operating at the assembly or machine code; or even using high-level mechanisms, which operate closer to the source code (i.e., fault injection is performed onto program variables and statements). Pros and cons of both are balanced by the fault model used in this study, based onto an intermediate representation level, which has been demonstrated to be accurate compared to assembly-level fault injections [1] [4] and permits to explore the effect of various fault injections parameters on the solver resilience. This methodology is presented in the next section. Opposite to them, another technique is the hardware-implemented fault injection [3] intended to tackle with faults which impose a permanent value onto a point in a circuit, which is out of the scope of this investigation.

Among the iterative methods to solve linear systems of equations, those based on Krylov subspace techniques with suitable preconditioners are commonly considered a good choice to attain both robustness and efficiency [5-8]. An important class within them is the Generalized Minimum RESidual (GMRES) method, which is common in many research and commercial solvers of a wide variety of scientific and technological disciplines (fluid and structural dynamics, electromagnetism, multiphysics simulation,...). This scheme permits to solve general sparse non-symmetrical linear systems of equations. Another iterative scheme, competitor to GMRES in computational efficiency, but limited to symmetric linear systems of equations, is the Conjugate Gradient (CG) scheme, not focused here since this work circumscribes to the more general scenario of non-symmetrical matrices. A characterization of the resilience behaviour under fault injection of three variants of GMRES has been accomplished in this investigation.

These three variants correspond to different preconditioners embedded into the GMRES: the common Incomplete LU factorization (ILU) [9], of widespread usage; a so-called “Flexible” variant [9] [10], which permits to vary the preconditioner in each step of the GMRES inner-loop; and a novel preconditioner based on a randomized implementation of the Singular Value Decomposition (SVD) algorithm, built on an error matrix which measures the deviation of ILU from the LU decomposition of the problem matrix [11]. These are described in section 3. Prior to their analysis, a set of kernels extracted from the Parboil [12] and Linpack [13] suites has been benchmarked under fault injection to identify important aspects for the setup of the experiments performed with the GMRES solvers, prototyped in C code.

In summary, this paper makes the next contributions:

- The Randomized-SVD -based GMRES prototype is analysed (for the first time to the authors' knowledge) under fault injection, to assess its resilience across other versions. The results over 18 sparse matrices suggest a competitive resilient behaviour compared to the Flexible GMRES version.

- The randomized-SVD algorithm is identified as a promising mathematical tool to be exploited in the design of GMRES versions intended to be used in error prominent environments.

A major result of this investigation is that the GMRES with randomized-SVD -based preconditioning performs notably better than the other two GMRES implementations in terms of attained resilience, being then a promising idea to exploit by iterative solvers intended for supercomputing applications.

This paper is organized as follows. Section 2 introduces the fault injection tool used to analyse the error resilience of the benchmarks considered. Their behaviour has been assessed and major guidelines are set to conduct injection tests on the C-prototypes of the GMRES implementations. The three GMRES variants are described in section 3, with emphasis on the algorithmic differences. Section 4 presents the collection of sparse matrices selected on which the GMRES solver has been executed embedded into the fault injection framework. A discussion of the results follows in section 5. Last, a survey of related work is given and some conclusions provided.

2- FAULT MODEL AND BENCHMARKING

The fault model used in the present investigation, which drives the design of the Low Level Fault Injection (LLFI) tool [1] [2] [14], relies on the following assumptions. Transient and intermittent faults may occur in the CPU, typically provoked by the impact of alpha particles or cosmic rays onto flip flops and logic devices. Both arithmetic-logic units and memory addresses are considered susceptible of experiencing a fault (on the contrary, memory components -as the cache- are excluded as these used to be protected at the architectural level by ECC or parity). Neither faults affecting the control logic of the CPU (it is a small physical area of the CPU), not the instructions encoding (handled through control-flow checking techniques) are accounted for.

2.1 Fault injection tool

This study exploits the LLFI fault injection tool to analyse the resilience of scientific programs by allowing fault-injection at defined program points during execution. LLFI is built on the LLVM compiler infrastructure [15] and works at the LLVM compiler's intermediate representation (IR) level in a closely integrated manner. The IR code is then used internally by a compiler to represent the source code without loss of information, and permits further optimization or translation. In particular, LLVM can translate code from C/C++ to IR and to translate it back from IR to machine code for various architectures (x86 processors, ARM...). The program source code is given as input to the LLVM to translate it to IR. In a second step, LLFI instruments the IR code using fault injection functions in all those program points where a fault can be potentially injected. The fault types and program points (injection sites) are defined based on a set of compile-time options given by the user in an input file, read in the instrumentation step. At execution, following the run-time options specified by the user, injection sites are randomly chosen from the total number of possible injection sites, then a fault is injected by a custom fault injection library and the program is run to completion. Further details on the LLFI workflow (see Figure 1) and architecture may be found in the above mentioned references.

Among the several fault injection types of LLFI, the present work has focused on the bit-flip fault type. Those bit-flip events span different sets of instructions, source registers and operands as random injection targets in each experiment. Execution behaviour after fault injection is classified according to four types of program outcomes:

- Crash: a bit flip may trigger the termination of the application by the operating system due to an exception (i.e., a bit flip modifies a pointer variable and the corrupted address is outside the process address space, causing a segmentation fault). There is no simple way to recover from crashes, which are more related to hard faults. That is why both SDC and OK executions are the two more relevant group in this study.

- Hang: it occurs when the program takes much more run time than the reference solution and times out.

- OK (or masked) execution: a successful output of the application, not affected by the presence of an error. The fraction of OK executions quantifies application resilience. Some algorithms are, to some extent, intrinsically resilient (i.e., iterative solvers as the GMRES and others) and may still produce correct results in the presence of faults at the cost of executing a larger number of iterations to converge.

▪ Silent Data Corruption (SDC): an undetected error silently contaminates application data structures, leading to an erroneous output but without triggering an error (the output deviates from the reference solution more than an acceptable range). This may be assessed by comparison with a reference solution computed in a fault-free execution.

For each of the analised programs under fault injection, these four rates have been calculated by simply dividing the number of their respective ocurrences in the program outcome by the total number of fault injection experiments. To avoid confusion, given a program which has been run in N injection experiments, being N_i (with i = OK, SDC, Crash and Hang) the number of ocurrences of the failure type i, the rates are calculated with the formulae $Rate_i\,(\%) = 100\,N_i/N$, where $\Sigma N_i = N$. The statistical accuracy of these rates is estimated with the squared root of the number of the ocurrences, i.e. $\Delta N_i = \sqrt{N_i}$. And in %, the expression reads $\Delta Rate_i\,(\%) = 100\,\sqrt{N_i}/N$. These errors appear indicated in parenthesis in the tables of the benchmarks and GMRES versions in the next sections, affecting then the last significative digit of each rate (in this way, 13 (4) %, means 13 ∓ 4 %).

2.2 Benchmarking

The Parboil benchmark suite [12] of the University of Illinois is a collection of accelerated, heterogeneous programs emphasizing throughput-oriented computing. It comprises a variety of algorithms: iterative methods, dense and sparse array operations, data-dependant memory access patterns, etc. Each benchmark kernel is intended to be scalable and may be tested with a series of small to large input datasets included in the distribution. Besides, multiple architecture-optimized implementations are available in the suite (multithreaded CPU-coded, heterogeneous CPU+Accelerator sources...), also written in different programming models (C, C++, OpenCL and CUDA) to support benchmarking comparison scenarios. These benchmarks are relatively simple compared to end-to-end user applications.

In this study, the baseline accelerated CPU version has been analysed under fault injection with the LLFI tool. A constrain that should be noticed is that LLFI only works with CPU-coded serial programs [14]. An extension of LLFI to analise MPI-based parallel programs has been accomplished in [16] [17], albeit not included in the official development [14]. The four selected benchmarks of Parboil (see Table 1) are described next.

a. *factorial*: factorization of an integer provided as input. It spans ten to twelve lines of code with only three integer storage variables. It is included into the Parboil suite as an example, not as a benchmark itself.

b. *stencil:* corresponds to an iterative Jacobi solver, which results from the discretization of the 3D heat equation on a structured grid. It may be seen as a building block of more advanced multi-grid partial differential equations (PDEs) solvers.

c. *spmv:* sparse matrix - dense vector multiplication. It is a basic repetitive operation of many iterative solvers and leads to memory-bandwidth boundness for very large matrices, mainly in such cases of irregular access over non-zero elements patterns of both matrix and vector. Typically, the sparse matrix is stored in a compressed notation to reduce the data allocation requirements.

d. *sgemm*: product of two dense matrices, which appears as a common building block in many numerical linear algebra codes and packages (i.e., the numeric library BLAS - Basic Linear Algebra Subprograms-, typically provided by hardware vendors in highly optimized versions: MKL, CUBLAS,...). Dense matrix operations are ubiquitous, thus their relevance.

Table 1 Benchmarks used in sensitivity analysis

Benchmark	Input / size	Short description
factorial	-	Factorization (n!)
stencil	Small / 128 x 128 x 32	3D stencil operation
spmv	Small / 1138 x 1138	Sparse-matrix Dense-vector multiplication
sgemm[1]	Small / 128 x 160	Dense matrix-matrix multiplication
linpack_bench[2]	256 x 256	Linear system of equations solver

[1] Portions written in C++ have been adapted to C for their use with the faults injector software.
[2] C program which mimics the corresponding *Ax=b* LINPACK benchmark program.

In addition to the Parboil benchmarks, a fifth benchmark called *linpack_bench* has been also tested (see Table 1). It is a C program corresponding to the LINPACK solver of linear systems of equations by LU factorization of the system matrix [13]. It is noticed that four of these benchmarks clearly mimic code portions of the preconditioned GMRES versions. In particular, *sgemm* is an iterative algorithm over a sparse matrix with a convergence criterion based on a residual norm, like GMRES; *spmv* occurs every time a vector of the Krylov space is generated into the inner-loop of the GMRES; the product of two dense matrices (*sgemm*) is executed into the preconditioner portion of code when the randomized-SVD is built in the third variant of GMRES. And finally, the LU decomposition coded into *linpack_bench* to solve a linear system

is similar to the incomplete LU used every time the preconditioner is applied. Only *sgemm* is specific of the Randomized-SVD -based GMRES implementation. As seen, the toy problem *factorial* is not directly related to the GMRES algorithm, but its simplicity is taken here as an example to stress the effect of silent data corruption even in very simple scenarios, for completion purposes.

Table 2 Classification of the benchmarks failure types obtained under single-bit flip injection. Four failure types are considered at code execution: crash, hang, successful execution (OK) and silent error (SDC) rates (error in the last significative digit is indicated in parenthesis. For those rates strictly zero, the error calculation is not well defined; so it is indicated as "(-)").

Number of experiments per test								
	10^2		10^3		10^4		10^5	
factorial								
Crash (%)	13	(4)	8.6	(9)	10.0	(3)	10.2	(1)
Hang (%)	0	(-)	0	(-)	0.1	(1)	0.07	(1)
OK (%)	16	(4)	21	(1)	20.9	(5)	21.4	(1)
SDC (%)	71	(8)	71	(3)	69.1	(8)	68.4	(3)
spmv								
Crash (%)	48	(7)	44	(2)	44.6	(7)	45.5	(2)
Hang (%)	0	(-)	0	(-)	0	(-)	0.02	(1)
OK (%)	50	(7)	55	(2)	54.0	(7)	53.0	(2)
SDC (%)	2	(1)	1.4	(4)	1.4	(1)	1.3	(1)
stencil								
Crash (%)	37	(6)	38	(2)	39.7	(6)	39.5	(2)
Hang (%)	0	(-)	0	(-)	0.06	(1)	0.06	(1)
OK (%)	21	(5)	25	(2)	23.7	(5)	23.3	(2)
SDC (%)	42	(7)	38	(2)	36.6	(6)	37.2	(2)
sgemm								
Crash (%)	32	(6)	32	(2)	33.7	(6)	34.0	(2)
Hang (%)	0	(-)	0	(-)	0	(-)	0	(-)
OK (%)	14	(4)	11	(1)	10.6	(3)	10.5	(1)
SDC (%)	54	(7)	57	(2)	55.8	(7)	55.5	(2)
linpack_bench								
Crash (%)	32	(6)	41	(2)	41.5	(6)	41.5	(2)
Hang (%)	0	(-)	0	(-)	0	(-)	0	(-)
OK (%)	11	(3)	6.5	(8)	7.3	(3)	7.6	(1)
SDC (%)	57	(8)	53	(2)	51.2	(7)	51.0	(2)

In the experimental LLFI setup, the general scenario of injecting bit flips in any instruction type of the LLVM IR code (arithmetic, logical, load instructions,... see [1] for further details) and in any source register of the benchmark programs, has been considered. The number of executions is prescribed as part of the input to LLFI, thus gathering enough outcomes of failure types statistics in every LLFI test. When an instruction uses more than one register, LLFI randomly choses one to inject into it. The previously enumerated four failure types are considered at runtime: crash, hang, successful execution (OK) and silent data corruption (SDC).

A sequence of fault injection tests, each comprising 100, 1000, 10,000 and 100,000 experiments, has been executed for all the benchmarks (an experiment is a definite benchmark execution with a single or double-bit flip injection). Single-bit flip injection results are summarized in Table 2. The very few allocated variables and instructions into the *factorial* program explain such a small number of crashes, high rate of SDC and moderate rate of successful executions. On the contrary, the *smpv* benchmark shows a very low contribution of SDC (1.4%) and rather large crash rate (45%), which is justified by the compact storage of the sparse matrix involved (few data allocation is needed), then a small memory zone is susceptible of experiencing a bit-flip data corruption. *Stencil, sgemm* and *linpack_bench* are more extensive and complex C codes, with larger data structures which seems to be in consonance with the higher rate of crashes and SDC observed during the experiments and, as a result, a lower rate of successful executions (that is, 23%, 11% and 8% respectively). All experiments revel that the hang failure type is a rare event, under a 0.1% rate.

Table 2 clearly shows that failure rates within the range 10,000 to 100,000 experiments flatten on definite values, which leads to say that 100,000 experiments is appropriate to attain reliable figures of the failure type rates. This criterion of performing such large number of experiments has been enforced as well in the GMRES fault injection tests described in the next sections. Failure rates flattening is also visible in the SDC rate plot in Fig. 2 (also to 100,000 experiments performed for each benchmark has been plotted). The large differences in SDC rates stresses their very different behaviour in terms of resilience, therefore the need to carry out a specific analysis for each class of code since they differ in data volume allocation, loops and conditional branching, structures calls, etc.

A repetition of the tests has been conducted including double-bit flip injection per execution and identical sequence of increasing number of experiments (up to 100,000) to get an insight on

how the benchmarks failure types are affected. The second bit-flip is injected randomly either in the same source register or in another register as the first bit-flip (injection in different registers mimics those double-bit flip faults that may occur within a narrow time window, but not at the same location). The performance attained with single-bit and double-bit flip test injections is plotted side-by-side in Fig. 3. The results show that doubling the bit-flip injection implies an increase in the crash rate, as well as a decrease in the successful execution rate across all the benchmarks. As a result, the trend of the SDC rate is to decrease, although the f*actorial* benchmark is an exception to this rule. These trends are discernible by the fact that with 100,000 experiments, the error in the four measured rates is below 0.3%. This bit-flip behaviour is in agreement with other reported work [1] [18] and matches the intuitive reasoning that as fault injection rate per experiment increases, proner is the code to exhibit a crash.

Since a major concern of this study is to analyse the resilience of the GMRES solvers under silent data corruption, and being the SDC rate typically larger for single-bit flip injections, the scenario of single-bit flip has been adopted in the GMRES tests.

3- GMRES IMPLEMENTATIONS

The original preconditioned restarted GMRES iterative solution method of Saad and Schultz [19] and two additional modified versions, all of them using right-preconditioning, are described in this section. It is noticed that this work is restricted to matrices and vectors of real numbers, but a more general GMRES formulation to deal with complex numbers arithmetic is possible. For any linear system $Ax=b$, A is a square non-symmetric, non-singular, real $n \times n$ matrix; and b is an n-length, real vector. The system solution x is computed iteratively using the residual $r=b-Ax$ to update an m-dimensional $\{A, r, m\}$-Krylov space [9].

The present study prescribes the same size of the Krylov subspace and termination criterion for the three GMRES variants. In particular, the inner-loop builds a Krylov subspace with m=20 direction vectors, and the outer-loop restarts twenty times (hence, a maximum of 400 iterations is permitted. Once this value is reached, a non-convergence flag is raised by the solver execution). Convergence of the GMRES is evaluated using the equations residuals (RHS) as iteration progresses. A termination criterion based on both absolute and relative residuals norms decay is checked after each inner-loop iteration, such that the process is stopped as soon as they drop below the tolerance, set to 10^{-13}.

3.1 Restarted (Standard) GMRES

Let x_0 be an initial guess for this linear system, $r_0=b-Ax_0$ be its corresponding residual and M^{-1} the right-preconditioner, prescribed for the whole iteration and which is some approximation to the matrix A. The modified system solved by the GMRES is reformulated as: $AM^{-1}(Mx) = b$. Pseudocode may be found elsewhere [9] [19] [20] and it is laid out in Fig. 4 for clarity of the notation and comparison of the GMRES versions. It is noticed that AM^{-1} is not needed explicitly. But efficient preconditioning implies that the operation $Mz_j=v_j$ has to be solved for z_j whenever it is required. Hence, $M^{-1}v_j$ should be easily computable for an arbitrary v_j.

In this method, the Arnoldi loop builds an orthogonal basis of the preconditioned m-dimensional Krylov subspace using the modified Gram-Schmidt algorithm. The last step of the pseudocode (see Fig. 4) generates the solution as a linear combination of the preconditioned vectors, that is: $z_j = M^{-1}v_j$, with j =1,2..., m, where the preconditioning matrix M^{-1} is the same for all the vectors.

This m-dimensional Krylov restarted version, shortly GMRES(m), sets M as a low-accuracy LU factorization of the sparse matrix A. This is a widely used alternative that avoid making the factorization of A too expensive because of the much higher level of fill-in of the lower and upper triangular factors (L and U, respectively). Typically, LU factors are kept under a given threshold to enforce better sparsity. The incomplete LU version adopted here is the zero-order LU factorization, in short ILU(0), which corresponds to a sparsity pattern of LU equal to A and makes the direct solve of $Mz_j=v_j$ cheap.

3.2 Flexible GMRES (FGMRES)

In 1993, Saad [10] introduced the FGMRES, which extends the GMRES(m) method by allowing the preconditioner to change in each step of the Arnoldi process, leading to a more flexible preconditioner (this added flexibility is suggested by its acronym) and a moderate additional computational cost. As in the standard GMRES, it satisfies the residual norm minimization condition over the preconditioned Krylov subspace.

A formally important drawback of this idea is that the construction of a Krylov subspace does not allow the preconditioner to change from step to step in strict sense, and no general proven statement of convergence may be given. That is, mathematically it may break down. On the contrary, the GMRES(m) cannot break down, regardless of the positiveness of A. Another drawback of the FGMRES formulation is that it typically has a similar arithmetic cost but doubles the memory requirement compared to the standard GMRES framework, depending on

the preconditioner formulation. Since with a varying preconditioner, it is $z_j = M_j^{-1} v_j$ for j=1,2,...m (being M_j specific at the j-th iteration step), there is a need to store the intermediate sequence of vectors z_j of the Arnoldi process. Hence, the solution is updated by composing the approximate solution as $x_m=x_0 + Z_m Y_m$ (see Fig. 5 for notation details), were $Z_m=[z_1,....z_m]$ is an m-dimensional Krylov subspace.

In practical terms, any iterative solution method may be used as a preconditioner in the FGMRES version, because its normwise termination criterion generates a de-facto M_j, different for every z_j. Under this approach, the preconditioner is not explicitly set, but implicitly stated via some computation (some few steps of an iterative method) [9]. Particularly, the iterative GMRES(m) procedure (preconditioned with ILU(0) itself) may play this role, as a preconditioner of the outer GMRES(m). This is the chosen method in the FGMRES implementation of this work. For the situation in which GMRES(m) is embedded as preconditioner of an outer GMRES, a more efficient memory implementation may be designed, precisely because there is a corresponding allocation matching with the available free vectors already allocated during the outer-loop, which can be reused by the preconditioner vectors.

In addition, it should be stressed that a key point behind letting the preconditioner change (even unboundedly) in successive inner solves, is that faults affecting memory data or arithmetic involved in the preconditioning portion of the algorithm, may be taken as part of a new preconditioner itself in the iteration sequence. Thus, a somehow fault-tolerant behaviour seems to be inherent to the idea of a changing preconditioner.

3.3 Randomized-SVD -based GMRES

An approximated LU factorization of A can be written as $A = L_0U_0 + \Delta A$, with L_0, U_0 the factors obtained from the ILU(0) decomposition. Compared to the standard GMRES(m), this approach improves the fidelity of the incomplete LU factorization of A by introducing an error matrix E linked to ΔA. Following the previous notation, the expression of the error matrix is $E = M^{-1}A - I$, which under the ILU(0) based preconditioning, it reads $E = U_0^{-1}L_0^{-1}A - I$, with the expectation of accelerating the convergence by E retaining the low-rank properties of A^{-1}.

This form of novel preconditioning can deal with both well-conditioned and ill-conditioned sparse matrices (that is, $\kappa(A) = \|A\|\|A^{-1}\| >> 1$, where $\|A\|$ is the Frobenius norm of A). However, accordingly to [11], it seems to be especially efficient to cope with the ill-conditioned cases since E is likely to have low numerical rank (a further mathematical analysis on why

ill-conditioned matrices used to have a small population of small singular values can be found in [11]).This approximation of A^{-1} leads to a preconditioner defined as

$$M^{-1} = (I + E)^{-1}U_0^{-1}L_0^{-1} \approx (I + E_k)^{-1}U_0^{-1}L_0^{-1}$$

where E_k with $k<<n$ is a rank-k approximation to E, to make the inverse of $I+E_k$ within an affordable time [22]. Two important issues must be stressed at this point: first, E is less sparse than A. And, second and fundamental in the adopted approach, the dense matrix E_k obtained as a truncated randomized-SVD of E, is conducted to exploit this SVD-decomposition in the above mentioned inverse building. Both randomized and non-randomized SVD approaches lead to express the matrix E as a product of factors, say $E=USV^T$ (where U, V are orthonormal matrices; and S a diagonal matrix with the singular values), but they imply rather different computing cost. These mathematical details are not presented here for brevity, to avoid a cumbersome presentation. A more thoroughly description may be found in [11] [23] [24]. In addition, the pseudocode of the Randomized-SVD -based GMRES is provided in Fig. 6 (the details of the randomized-SVD itself is given at the end of Fig. 6).

4- SET OF PROBLEMS AND METHODOLOGY

A group of real, sparse, non-symmetrical, squared matrices has been analysed in the present work. All of them are taken from the SuiteSparse Matrix Collection, which comes from the initially called Sparse Matrix Collection of the University of Florida [25], a widely used, large repository of matrices, easily accessible, actively growing and many of them derived from real world applications. A presentation of this matrix dataset may be found in [26].

Hence, in the linear system $Ax=b$ to be solved by GMRES, the $n \times n$ matrix A is taken from this dataset. The vector b is computed by initializing the solution vector to all ones and its last element to n. Then computing $b=Ax$ ensures that the system of equations has a valid solution. Several criteria have driven the selection of the analysed matrices among the diversity of them included in the collection. First, all they derive from a discretized system of equations, casted into matrix form, so they have a clear mathematical sense. This is a relevant issue as some of the SuiteSparse matrices correspond to combinatorial problems, graphs or images (thus, not appropiate to search for a solution using GMRES). In addition, a vast range of the condition number has been explored. Hence, these tests comprise a number of 18 matrices (see Table 3 for the full list), which have been classified into three different types: a set of 6 matrices with low condition number, that is $\kappa(A) \approx O(1) - O(10^2)$. Another set of 6 matrices with high condition

number $\kappa(A) \approx O(10^3)$ - $O(10^4)$. And a third subgroup of 6 matrices, with very high condition number $\kappa(A) \approx O(10^6)$ - $O(10^{12})$. The effect of the condition number is discussed in section 5.

Table 3 Set of square, non-symmetrical matrices taken from the SuiteSparse Matrix Collection [25][26]. Matrices are classified into three types according to their condition number $\kappa(A)$.

Matrix	Size	$\kappa(A)$	Application
Low condition number: $\kappa(A) \approx O(1)$ - $O(10^2)$			
dw256B	512 x 512	3.7	Electromagnetism
pde225	225 x 225	39	Model Partial Differential equation
cdde4	961 x 961	68	Convection-Diffusion model
poisson2D	367x367	133	Poisson PDE in two-dimension
ex37	3565 x 3565	180	Computational Fluid Dynamics
bfwa62	62 x 62	553	Electromagnetism
High condition number: $\kappa(A) \approx O(10^3)$ - $O(10^4)$			
bwm200	200 x 200	$2.4\ 10^3$	Chemical Processes
ex1	216 x 216	$3.3\ 10^4$	Computational Fluid Dynamics
ex22	839 x 839	$3.3\ 10^4$	Computational Fluid Dynamics
orsirr_2	886 x 886	$6.3\ 10^4$	Computational Fluid Dynamics
tub100	100 x 100	$1.3\ 10^4$	Tubular Reactor model
olm100	100 x 100	$1.5\ 10^4$	Computational Fluid Dynamics
Very large condition number: $\kappa(A) \approx O(10^6)$ - $O(10^{12})$			
pores_1	30 x 30	$1.8\ 10^6$	Computational Fluid Dynamics
steam1	240 x 240	$2.8\ 10^7$	Oil-Steam modelling
DK01R	903 x 903	$5.9\ 10^7$	Turbulent Flow
saylr1	238 x 238	$7.9\ 10^8$	Computational Fluid Dynamics
steam3	80 x 80	$5.0\ 10^{10}$	Oil-Steam modelling
lung1	1650 x 1650	$4.9\ 10^{12}$	Bio-Fluid Mechanics - Multiphysics

It is noticed that matrix symmetry has been avoided in the selection because it is well known that the iterative Conjugate Gradient (CG) method turns to be more efficient than GMRES in this class of problems. So the chosen matrices circumscribe to the more realistic scenario of matrix *A* exhibiting a non-symmetric pattern (either geometric or numerical) within.

Last but not least, an important criterion is related to the matrix size, to cope with the number of LLFI injection experiments set to 100,000 to attain injection-independence results at affordable computing cost. To this regard, matrix size has been restricted to rather small dimensions since LLFI is a serial application and this constitutes a bottleneck for its application to very big matrices.

As seen in Table 3, the field of application of most of the matrices is closely related to numerical fluid dynamics in its diversity of branches (from canonical transport problems, reacting and turbulent flows, multiphysics, etc). To extend the silent error statistics and balance the number of analysed matrices of each type (in terms of condition number), a couple of matrices of the electromagnetism area has been added to the low condition number subgroup. Matrices of the SuiteSparse Matrix Collection are read with the Rutherford-Boing (RB) format [26], which only stores the non-zero entries and creates a data structure accordingly to their location. It is simple and easy to use and exploits the compressed column representation of the sparse matrix as only three vectors. This format has its origin in the Rutherford Sparse Matrix Collection repository [27] for research in sparse linear algebra, being itself an update of the Harwell-Boing collection and data format. A RB-reader has been C-coded as part of the data input portion of the solvers and it is shared by the three implementations of GMRES.

A total of 54 tests (18 matrices per GMRES version), each corresponding to a population of 100,000 injection experiments, have been carried out. The strategy of injecting the faults only within the preconditioner portion of the C-code of each solver has been adopted; that is, into the part of code which is specific of the GMRES versions. This is an important aspect as the fault injection location is randomly set by LLFI and, under this approach, the entire set of injections may be concentrated. Hence, the combination of a random fault injection location; large size of the experiments population; and a definite, bounded extension of C-code within the injection is performed, guarantees a rather informative statistics of the GMRES solvers under injection.

Computations have been done on the Ubuntu14.04 -based virtual machine of LLFI, available from [14] and sized with 14 CPUs Intel Xeon (4 cores) X5560 @ 2.80GHz and 20Gbs of RAM, set in a Ciemat server to accelerate the completion of the experiments. The repetition of half of the tests on a Centos7 -based laptop with one CPU Intel i7-8565U (4 cores) @ 1.80GHz has demonstrated that the obtained results are insensitive to the computing platform. All computations have been performed in double precision. The number of iterations needed to converge strongly depends on the GMRES version and dataset, and it ranges from 2 to 400 (once this last number is reached, iterated solutions are tagged as non-converged and stopped in the GMRES implementations).

5- RESULTS AND DISCUSSION

For clarity and better identification of the resilience performance, Figs. 7 and 8 group the behaviour of each dataset for the three GMRES variants according to the following two criteria: Fig. 7 is plotted with the premise of checking the same equations residual (RHS) than the baseline solution to accept the execution as correct (OK flag). Fig. 8 corresponds to a less restrictive criterion which includes into the OK group those executions that achieve the prescribed tolerance (RHS ≤ tolerance) in fewer or equal number of iterations than the baseline solution (that is, iter ≤ $iter_{baseline}$). As a result, both plots exhibit the same length of the crash bar, but differ in the other rates. Their comparison under these two criteria stresses that the number of OK executions increase when iter ≤ $iter_{baseline}$ is used, which is something expected to occur since the solver convergence according to the second criterion is set in a wider sense.

It should be mentioned that iterations-to-converge strongly depend on the dataset (that is, matrix properties) and the solver version itself. This work focuses on resilience and not assesses the convergence acceleration provided by each particular algorithm, which is another important issue that deserves further research and is not addressed in this work. Several representative cases of the statistical distribution of iterations-to-solution are shown in Fig. 9. Because the error injection affects the distribution of iterations-to-converge, there is typically a set of executions which converge within fewer iterations compared to the baseline case (a kind of super-convergence) to match the same tolerance criterion of convergence. These cases are precisely counted for within the criterion provided in Fig. 8 and then included into the OK executions. Most of the tests reveal that the population of executions that exhibit super-convergence contributes to the OK type within a very small amount, less than 1%. Only in very few tests this contribution has reached up to about 5% of the OK rate.
The criterion including super-converged executions, instead of the one based on looking for convergence with strictly the same number of iterations than the baseline solution, has been preferred and is plotted in Fig. 8 because of its higher relevance when discriminating well-converged solutions in practical terms.

Overall, the Crash rate is within the interval 26% to 57%, with average between 45% and 51% depending on the GMRES version (see Fig. 10 for more details). In addition, SDC rates are within 11% and 68% for the criterion RHS=$RHS_{baseline}$ of Fig. 7. It is observed that the percentual contribution of the Hang event is very low (of about 0.1% in the worse cases), so it is not discernible in the plots.

Interestingly, Crash bars show that there is a general trend of having lower percentage of crashes when applying the Randomized-SVD -based preconditioner, independently of the matrix condition number. To this respect, the average and range of crashes across all matrices for each solver are plotted in Fig. 10 and it is seen that the Randomized-SVD -based GMRES version incurs in a 10% less crashes than the other two when taken into consideration (since the Hang rate is very small, this observed drop of the Crash rate implies an increase of the OK rate, SDC rate, or both in this version of GMRES). Likewise the Standard GMRES, the FGMRES exhibits similar margin and average numbers for the Crash event. Besides it is well known that iterative solvers provide an inherent resilient behaviour compared to their direct counterparts, it is seen from Figs. 7, 8 and 10, that the Randomized-SVD -based GMRES provides a beneficial outcome in terms of extra resilience with the prescribed settings coded in its algorithm.

Table 4 Convergence success (%) for (RHS, iter) ≤ (tolerance, iter)$_{baseline}$[1] for the three types of matrices. The best combination matrix-GMRES version is shown with bolded numbers (error in the last significative digit is indicated in parenthesis).

Low condition number: $\kappa(A) \approx O(1)$ - $O(10^2)$

GMRES	dw256B	pde225	cdde4	poisson2D	ex37	bwfa62
Standard	50.1 (2)	25.8 (2)	37.7 (2)	31.0 (2)	48.1 (2)	28.9 (2)
Flexible	**55.6 (2)**	**45.8 (2)**	**49.5 (2)**	22.6 (1)	**58.6 (2)**	**43.4 (2)**
Randomized-SVD	50.5 (2)	37.5 (2)	34.8 (2)	**31.6 (2)**	24.9 (2)	30.3 (2)

High condition number: $\kappa(A) \approx O(10^3)$ - $O(10^4)$

GMRES	bwm200	ex1	ex22	orsirr_2	tub100	olm100
Standard	28.0 (2)	32.3 (2)	29.6 (2)	35.9 (2)	28.9 (2)	21.1 (1)
Flexible	**43.5 (2)**	**44.6 (2)**	**47.9 (2)**	46.7 (2)	**44.2 (2)**	42.1 (2)
Randomized-SVD	27.9 (2)	36.0 (2)	32.9 (2)	**51.4 (2)**	43.1 (2)	**54.4 (2)**

Very large condition number: $\kappa(A) \approx O(10^6)$ - $O(10^{12})$

GMRES	pores_1	steam1	DK01R	saylr1	steam3	lung1
Standard	33.0 (2)	52.1 (2)	28.7 (2)	28.9 (2)	48.2 (2)	49.9 (2)
Flexible	43.5 (2)	**52.5 (2)**	**51.6 (2)**	**44.2 (2)**	46.4 (2)	**58.4 (2)**
Randomized-SVD	**44.0 (2)**	31.0 (2)	51.1 (2)	38.0 (2)	**52.6 (2)**	46.8 (2)

[1] This criterion corresponds to Fig. 8.

It is stressed that no attempts of searching for a set of optimal, tuned parameters into the Randomized-SVD -based GMRES has been done in this study, and the same parameters values have been applied to the entire population of matrices analysed. Under this, say, rough approach, the mentioned improvement has been obtained.

Convergence success is shown in Tables 4 and 5 for the three types of matrices according to their condition number. While Table 4 counts as converged solutions all the executions which satisfy the double condition stated as (RHS, iter) ≤ (tolerance, iter)$_{baseline}$, Table 5 comprises also those solutions that converge according to RHS < tolerance; that is, no matter the number of iterations needed, but less than the maximum permitted (400 iterations).

Table 5 Convergence success (%) for criterion RHS ≤ tolerance, for the three types of matrices. The best combination matrix-GMRES version is shown with bolded numbers (error in the last significative figure is indicated in parenthesis).

Low condition number: $\kappa(A) \approx O(1)$ - $O(10^2)$

GMRES	dw256B	pde225	cdde4	poisson2D	ex37	bwfa62
Standard	56.2 (2)	45.9 (2)	50.5 (2)	45.0 (2)	58.3 (2)	44.3 (2)
Flexible	**57.2 (2)**	45.8 (2)	**52.6 (2)**	71.1 (3)	58.5 (2)	43.5 (2)
Randomized-SVD	56.7 (2)	**48.4 (2)**	50.3 (2)	**73.8 (3)**	**69.0 (3)**	**52.8 (2)**

High condition number: $\kappa(A) \approx O(10^3)$ - $O(10^4)$

GMRES	bwm200	ex1	ex22	orsirr_2	tub100	olm100
Standard	44.0 (2)	**49.0 (2)**	48.6 (2)	46.3 (2)	45.7 (2)	42.4 (2)
Flexible	43.6 (2)	47.3 (2)	48.3 (2)	46.7 (2)	44.9 (2)	42.2 (2)
Randomized-SVD	**50.3 (2)**	47.5 (2)	**51.5 (2)**	**57.3 (2)**	**47.6 (2)**	**54.4 (2)**

Very large condition number: $\kappa(A) \approx O(10^6)$ - $O(10^{12})$

GMRES	pores_1	steam1	DK01R	saylr1	steam3	lung1
Standard	46.2 (2)	52.8 (2)	52.3 (2)	45.7 (2)	49.8 (2)	55.1 (2)
Flexible	43.7 (2)	52.3 (2)	52.2 (2)	44.4 (2)	46.4 (2)	**59.7 (2)**
Randomized-SVD	**51.3 (2)**	**61.9 (2)**	**56.7 (2)**	**48.9 (2)**	**53.8 (2)**	50.1 (2)

It is noticeable that no linking pattern of the SDC and OK rates with the condition number is observed in the plots and tables. Nevertheless, the criterion RHS=RHS$_{baseline}$ set in Fig. 7 points

out to an increase of the rate of OK executions for the matrices with very-high condition number. It is noticed that this is not strictly related to the Randomized-SVD -based GMRES algorithm itself, as it was expected to be observed from the discussion provided in [20]. On the contrary, the results suggest that the three GMRES versions behave especially well in those very ill-conditioned linear systems under fault injection. Unfortunately, the population of matrices considered is not large enough to be conclusive on this.

Table 6 Comparison of convergence success with the three GMRES versions.

Metric	Standard	Flexible	Randomized-SVD
RHS ≤ tolerance[1]	1	2	**15**
(RHS, iter) ≤ (tolerance, iter)$_{baseline}$	0	**13**	5

[1] This metric counts every converged experiment, not considering how many iterations takes to reach the tolerance criterion. The (RHS, iter) - metric is more demanding as it counts only equal- and super-converged solutions w.r.t. the baseline experiment.

If the above mentioned convergence success is taken as the metric (see Tables 4 and 5), the results show that FGMRES and Randomized-SVD -based GMRES clearly outperform the Standard GMRES algorithm. Table 6 summarizes in which scenarios each GMRES may be the preferred option: while FGMRES typically achieves faster iterations-to-solution behaviour, the Randomized-SVD -based GMRES version provides a larger population of converged executions, with a significant drop of observed crashes, but paying the cost of being less competitive in time-to-solution, at least under the present parameter set in its prototyped implementation. Obviously, a much more accurate performance comparison of the Flexible and Randomized-SVD -based GMRES versions in terms of time-to-solution behaviour, would require their parallel implementation, not accomplished in the present study. In a massive parallelization scenario, delicate aspects regarding the limits of scalability of several specific algorithmic portions of the GMRES versions (i.e., number of matrix-vector multiplications, Householder orthogonalization, randomized-SVD,...) should be addressed and tested. For the present serial, prototyped implementation of the GMRES versions, the iteration-to-solution behaviour is emphasized in Fig. 9, where the criterion iter=iter$_{baseline}$ is plotted in dark gray.

Finally, it should be noticed that the completion of 100,000 injection experiments per test case is rather time consuming for the bigger matrices analysed. Therefore, a better resilience assessment of the targeted applications (that is, the GMRES versions in the present study) would be desireable. In this context, an important issue is the understanding of SDC propagation in codes [28] [29], to lower the number of experiments required to gather an informative, confident resilience statistics.

6- RELATED WORK

Error injection to understand their impact in HPC, may be performed at various abstraction levels: from circuit to application level. Circuit-level is probably the most accurate, but quite demanding since it requires sophisticated radiation-exposure facilities to irradiate hardware to induce bit flips in a CPU or other components. A common approach is to expose a processor's area to a proton or neutron beam, and then to measure soft error rates [30-32], but these methods are used to be very costly in time and budget (besides, to mention that these kind of experiments introduce new uncertainties, such as in the measurement of the distance between the CPU and radiation source; or the radiation source fluence itself). A survey on these approaches and a detailed list of the most known tools to tackle fault injection may be found in [3] [33] [34]. Another approach is to exploit Register Transfer Level (RTL) simulation. To this respect, most of the RTL-based tools presented in these references (Verify, Mefisto_C), as well as other developments to inject faults on CPUs and/or memory (Ferrari, Ftape, Fiat, Doctor,...) had their origin within very few years (1995-1997) in the past. They provide pointers to specific papers for further details on their usage.

On the other side of the abstraction level, there is the application-level methodology (that is, software-implemented fault injection tools), which permits to carry out fault injections in an accelerated fashion and do not require expensive hardware. The related frameworks are more recent developments and rather similar to the LLFI tool (compiler-level injectors), all using the LLVM infrastructure and IR code instrumentation to simulate transient faults. These are KULFI [35] [36] and VULFI [37], which are configurable as LLFI; FLIPIT [38], intended to inject faults into large-scale parallel applications; EDFI [39] which transforms the code at compile-time and instruments it; or FlipTracker [40], a framework to analise the resilience properties and error propagation using fine-grain tracking to identify resilient patterns in parallel applications, among others. In particular, some attempts to extend the functionality of LLFI from sequential applications to parallel ones have been conducted in [41], where various HPC parallel

applications have been analysed. The modified LLFI version randomly selects a participating processor and injects a failure into it during the execution, to mimic the occurrence of a bit flip on a arbitrary location. In [16] a more elaborated LLFI extension to MPI-based codes under fault injection has been accomplished with regard to the effect of soft-errors affecting the MPI communications as well. In [42] the performance of the REFINE compiler-based injection framework is presented and compared to LLFI on the same benchmarking applications. In [43] the high-level fault injector CAROL-FI is used to analyse the error propagation path and identify the portions of the code that once corrupted, are more likely to affect the output. In this case the authors focus on the appearance of SDC during executions of a set of benchmarks running on Intel Xeon-Phi processors. Other tools for fault injection in programs running on GPUs exist [44] [45].

Several works focus on the characterization of application resilience to SDC and their sensibility to various types of fault injection models. Kestor et al. [16] investigate the fault behaviour in a set of experiments carried out with distributed MPI-based scientific codes and assume one bit flip error per application run. A comparative analysis of soft-error on five iterative methods derived from the CG (CG itself, ICCG, BiCG, BiCGSTAB and CGS), is conducted by Kestor et al. [17] using single and multi-bit flip (2 and 4-bit flip) with identical fault injection configuration to evaluate the design of soft error injectors. Mutlu et al. [46-48] analyse the resilience of up to six iterative CG solvers intended for symmetric matrices with an error-injection strategy based on single, 2 and 4-bit flip injection in the same memory word and randomly chosen locations. Both are close works, but they focus on CG variants (CG requires symmetric matrices) and do not include the GMRES algorithm targeted in the present work for non-symmetrical matrices. As they concluded, corruption of more bits instead of single-bif flip increases the likelihood of an anomalous outcome (SDC and crashes).

Controversy exists about the need of taking into account single or multiple-bit flips in injection experiments. The present study follows the guidelines of [18], [49], which states that single-bit flip yields in most cases a higher SDC rate compared to multiple-bit flips. Besides, multiple-bit flips do not cause as much difference in the SDC results of experiments conducted using single-bit flip error, as other researcher have speculated [49]. Finally, considering that multiple-bit flips injection is a more complex scenario due to the additional degrees of freedom needed to explore (i.e., bit flips injected within the same or different memory words or registers; the temporal gap among injections in a given execution,...), the present investigation bounds its objectives at experiments performed with single-bit flip injection.

Regarding injection of soft-errors in preconditioned GMRES versions, Elliot et al. [50] investigate the effect of SDC of bit flip type in preconditioners taken as a black box and faulty portion of the solver. Both preconditioned GMRES and CG iterative solvers are considered in their study. They demonstrated algorithm-based fault tolerance by adopting selective reliability to protect portions of the solvers. In [20] the same authors discussed the robustness improvements linked to the fault-tolerant FT-GMRES version, using the GMRES itself in the inner-solve. They showed that a rather robust behaviour may be achieved given a single fault injected in the orthogonalization phase of the inner-solve.

The idea of selective reliability is also explored in [21] by means of two Matlab prototypes of FGMRES and FT-GMRES. They compared the number of iterations to converge under fault-injections programmed in the data structures. They showed that convergence degrades as the fault rate increases. Additional research on the robustness of preconditioned FGMRES and FT-GMRES may be found in [51]. In [52] the effect of two soft error models is analysed with an elliptic PDE problem. The authors performed fault injections in definite matrix times vector operations and also on the preconditioner step. An investigation on the sensibility to the injection pattern that causes SDC, also using the FGMRES and FT-GMRES solvers (here preconditioned in a different way), may be found in [53], where the time-to-solution overhead in the presence of faults injection has been analysed.

To the authors' knowledge there is no previous analysis of the Randomized-SVD -based GMRES version under fault injection. This investigation compares its performance to the preconditioned standard and Flexible GMRES algorithms. Hence, this study aims at shedding light on the resilience properties of this novel implementation.

7- CONCLUSIONS

A study of the error resilience behaviour of three GMRES versions has been accomplished. The GMRES algorithm targets the iterative solution of large sparse non-symmetric linear systems and underlies many scientific applications executed in supercomputers. Its ubiquitous usage justifies the continuing efforts on improving its preconditioning portion, to attain shorter iteration-to-solution and better error resilience.

The recent GMRES version based on randomized-SVD preconditioning has been C-prototyped, in addition to the Standard GMRES and FGMRES. All they have been analysed under single

bit-flip injection across 100,000 experiments per test, to assess their properties in an ample scenario of matrices, ranging 18 sparse non-symmetric linear systems, from well- to very ill-conditioned problems and condition numbers within the range *O(1)* to $O(10^{12})$.

The results show that both FGMRES and Randomized-SVD -based GMRES outperform the Standard GMRES version in terms of resilience. The quite robust behaviour of FGMRES to single events of SDC is well known from previous studies and here is also so assessed. Its comparison with the Randomized-SVD -based GMRES clearly shows that preconditioning using randomized SVD leads to a higher number of successes in terms of achieved convergence, but typically, convergence needs more iterations than FGMRES.

Interestingly, even with the very general, non-optimized set of parameters included in the Randomized-SVD -based preconditioner implemented in the GMRES, it performs well across the variety of real-life matrices analysed and exhibits insensitivity to the degree of ill-conditioning of the linear systems. Consequently, further improvements in its parameters settings should be investigated. This is something to explore in the future, to identify a black-box setting able to tackle efficiently a wide group of problems. Its potential usage points out to another important issue that deserves attention: to assess its resilience on larger linear systems at scale, by implementing a CPU-distributed Randomized-SVD -based GMRES or CPU-GPU hybridized versions. These aspects are part of the roadmap to be walked next.

Acknowlegment

This work was partially funded by the Spanish Ministry of Science, Innovation, and Universities CODEC-OSE project (RTI2018-096006-B-I00) and the Comunidad de Madrid CABAHLA-CM project (S2018/TCS-4423), both with European Regional Development Fund (ERDF). It also profited from funding received by the H2020 co-funded projects Energy oriented Centre of Excellence for computing applications II (EoCoE-II, No. 824158), and Supercomputing and Energy in Mexico (Enerxico, No. 828947). Last, the authors thank the clusters administrators at CIEMAT: Pablo García-Muller and Antonio Rubio-Montero for their support.

References

[1] Lu Q., Farahani M., Wei J., Thomas A. Pattabiraman K. (2015): *LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults.* In Procs. of IEEE Int. Conference on Software Quality, Reliability and Security, Aug. 3-5, Vancouver, Canada. DOI: 10.1109/QRS.2015.13

[2] Thomas A., Pattabiraman K. (2013): *LLFI: An Intermediate Code-Level Fault Injector for Soft Computing Applications.* In IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE), March 26-27, Stanford, CA, USA.

[3] Hsuen M.C, Tsai T..K, Iyer R.K. (1997): *Fault Injection Techniques and Tools*, Computer, pp. 75-82. DOI:10.1109/2.585157

[4] Wei J., Thomas A., Li G., Pattabiraman K. (2014): *Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults.* In Procs. 44th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN), pp. 375-382, DOI: 10.1109/DSN.2014.2.

[5] Saad Y., van der Vorst H..A. (2000): *Iterative Solution of Linear Systems in the 20th Century*. J. of Computational and Applied Mathematics, Vol 123 (1-2), pp.1-33. DOI: https://doi.org/10.1016 /S0377-0427(00)00412-X.

[6] Benzi M. (2002): *Preconditioning Techniques for Large Linear Systems: A Survey*. Journal of Computational Physics, Vol. 182 (2), pp. 418-477. DOI: 10.1006/jcph.2002.7176.

[7] Vuik C. (1995): *New Insight in GMRES-like Methods with Variable Preconditioners*. J. Comp. and Applied Mathematics, Vol. 61 (2), pp. 189-204. DOI:10.1016/0377-0427(94)00067-B.

[8] Saad Y. (2019): *Iterative Methods for Linear Systems of Equations: A Brief Historical Journey.* arXiv:1908.01083v1. DOI:10.1090/conm/754/15141.

[9] van der Vorst, H.A. (2003): *Iterative Krylov Methods for Large Linear Systems*. Cambridge Monographs on Applied and Computational Mathematics, Cambridge University Press, UK.

[10] Saad, Y. (1993): *A Flexible Inner-Outer Preconditioned GMRES Algorithm*. SIAM J. Sci. Comput. Vol. 14, pp. 461-469.Higham N.J, Mary Th. (2019): *A New Preconditioner that Exploits Low-Rank Approximations to Factorizations Error*. SIAM Journal on Scientific Computing, Vol.41 (1), pp. A59-A82. DOI: https://doi.org/10.1137/18M1182802.

[11] Higham N.J, Mary Th. (2019): *A New Preconditioner that Exploits Low-Rank Approximations to Factorizations Error*. SIAM Journal on Scientific Computing, Vol.41 (1), pp. A59-A82. DOI: https://doi.org/10.1137/18M1182802.

[12] Stratton J.A., Rodrigues C., Sung I.J., Obeid N., Chang L.W., Anssari N., Liu G.D., Hwu W.W. (2012): Parboil: *A Revised Benchmark Suite for Scientific and Comercial Throughput Computing*, IMPACT Technical Report, IMPACT-12-01.

[13] LINPACK benchmark:
https://people.sc.fsu.edu/~jburkardt/c_src/linpack_bench/linpack_bench.html

[14] LLFI software download: https://github.com/DependableSystemsLab/LLFI

[15] Lattner C., Avre V. (2004): *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in CGO 2004, pp.75-86. DOI: 10.1109/CGO.2004.128166.

[16] Kestor G., Peng I.B., Gioiosa R., Krishnamoorthy S. (2018): *Understanding Scale-dependent Soft-error Behaviour of Scientific Applications.* In Procs. of IEEE/ACM 18th Int. Symposium on Cluster and Grid Computing (CCGRID), May 1-4, Washington DC, USA. DOI: 10.1109/CCGRID.2018.00075.

[17] Kestor G., Mutlu B.O., Manzano J., Subasi O., Unsal O., Krishnomoorthy S. (2018): *Comparative Analysis of Soft-Error Detection Strategies: A Case Study with Iterative Methods.* In Procs. of 15th ACM International. Conference on Computer Frontiers (CF-2018), pp.172-182, May 8-10, Ischia, Italy. https:/doi.org/10.1145/3203217.3203240.

[18] Ayatolahi F., Sangchoolie B., Johansson R., Karlsson J. (2013): *A Study of the Impact of Single Bit-Flip and Double Bit-Flip Errors on Program Execution*. In: Bitsch F., Guiochet J., Kaâniche M. (eds) Computer Safety, Reliability, and Security. SAFECOMP 2013. Lecture Notes in Computer Science, vol 8153. Springer, Berlin, Heidelberg. DOI: https://doi.org/10.1007/978-3-642-40793-2_24

[19] Saad Y., Schultz M.H. (1986): *GMRES: A Generalized Minimal Residual Algorithms for Solving Nonsymmetric Linear Systems*, SIAM J. Scientific and Statistical Computing, Vol.7(3), pp. 856-869. DOI: https://doi.org/10.1137/0907058.

[20] Elliot J., Hoemmen M., Mueller F. (2014): *Evaluating the Impact of SDC on the GMRES Iterative Solver.* In Procs. of IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14), pp. 1193-1202. DOI: 10.1109/IPDPS.2014.123.

[21] Bridges P. G., Ferreira K. B., Heroux M. A., Hoemmen M. (2012): *Fault-tolerant Linear Solvers via Selective Reliability*. arXiv:1206.1390v1.

[22] Henderson H. V., S. R. Searle S.R. (1981): *On Deriving the Inverse of a Sum of Matrices,* SIAM Review 23(1), pp. 53-60, https://www.jstor.org/stable/202983.

[23] Halko N., Martinsson P-G., Tropp J. (2011): *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*. SIAM Review, Vol. 53 (2), pp. 217-288, http://www.jstor.org/stable/23065163.

[24] Martinsson, P-G. (2019): *Randomized Methods for Matrix Computations*. In IAS/Park City Mathematics Series. American Mathematical Society, Vol. 25, pp. 187-230.

[25] SuiteSparse Matrix Collection (University of Florida Matrix Collection): https://sparse.tamu.edu/

[26] Davis T.A., Hu Y. (2011): *The University of Florida Sparse Matrix Collection*. ACM Trans. Math. Softw. 38(1), Article 1, 25 pages.

[27] Duff I.S., Grimes R.G., Lewis J.G. (1997): *The Rutherford-Boing Sparse Matrix Collection.* Report of the Rutherford Appleton Laboratory*, 62 pages, RAL-TR-97-031.*

[28] Calhoun J., Snir M., Olson L.N., Gropp W.D. (2017): *Towards a More Complete Understanding of SDC Propagation*. In Procs. 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17). Association for Computing Machinery, New York, NY, USA, pp. 131–142, 2017. DOI: https://doi.org/10.1145/3078597.3078617

[29] Li Z., Menon H., Mohror K., Bremer P.T., Livant Y., Pascucci V. (2021): *Understading a Program's Resiliency Through Error Propagation*. In Procs. Principles and Practice of Parallel Programming Conference (PPoPP), Feb. 27-March 3, Republic of Korea. https://doi.org/10.1145/3437801.3441589

[30] Oliveira D., Pilla L., De Bardeleben N., Blanchard S., Quinn H., Koren I., Navaux P., Rech. P. (2017): *Experimental and Analytical Study of Xeon Phi Reliability*. In Procs. International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17). Association for Computing Machinery, New York, NY, USA, Article 28, 1–12. DOI: https://doi.org/10.1145/3126908.3126960

[31] Oliveira D., Pilla L., Hanzich M., Fratin V., Fernandes F., Lunardi C.B., Cela J., Navaux P., Carro L., Rech P. (2017): *Radiation-Induced Error Criticality in Modern HPC Parallel Accelerators*. In Procs. 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 577-588, DOI:10.1109/HPCA.2017.41

[32] Cher C., Gupta M.S., Bose P., Muller K.P. (2014): *Understanding Soft Error Resiliency of Blue Gene/Q Compute Chip through Hardware Proton Irradiation and Software Fault Injection*. In SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 587-596. DOI: 10.1109/SC.2014.53.

[33] Ziade H., Ayoubi R.A., Velazco R. (2004): *A Survey on Fault Injection Techniques.* International Arab Journal of Information Technology, Vol 1(2), pp. 171-186.

[34] Cho H., Mirkhani S., Cher C., Abraham J.A., Mitra S. (2013): Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design, in Procs. 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1-10. DOI: https://doi.org/10.1145/2463209.2488859

[35] Sharma V.C., Haran A., Rakamaric Z., Gopalakrishnan G. (2013): *Towards Formal Approaches to System Resilience*. In Procs. IEEE 19th Pacific Rim International Symposium on Dependable Computing, pp. 41-50. DOI: 10.1109/PRDC.2013.14

[36] Kooli M., Natale G.D., Benoit P., Bosio A., Torres L., et al. (2014): *Fault Injection Tools Based on Virtual Machines.* In Procs. of IEEE 9th Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 26-28 May, Montpellier, France. DOI:10.1109/ReCoSoC.2014.6861351

[37] Sharma V.C., Gopalakrishnan G., Krishnamoorthy S. (2016): *Towards Resiliency Evaluation of Vector Programs*. In Procs. IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1319-1328. DOI: 10.1109/IPDPSW.2016.187

[38] Calhoun J., Olson L., Snir M. (2014): *FLIPIT: An LLVM Based Fault Injector for HPC*. In: Lopes L. et al. (Eds.) Euro-Par 2014: Parallel Processing Workshops. Lecture Notes in Computer Science, Vol. 8805. Springer, Cham. DOI: https://doi.org/10.1007/978-3-319-14325-5_47

[39] Giuffrida C., Kuijsten A., Tanenbaum A.S. (2013): *EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments*. In Procs. IEEE 19th Pacific Rim International Symposium on Dependable Computing, pp. 31-40. DOI: 10.1109/PRDC.2013.12

[40] Guo L., Li D., Laguna I., Schulz M. (2018): *FlipTracker: Understanding Natural Error Resilience in HPC Applications*. In Procs. SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 94-107. DOI: 10.1109/SC.2018.00011

[41] Ni X., L. V. Kale L.V. (2016): *FlipBack: Automatic Targeted Protection against Silent Data Corruption*. In Procs. International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16), Salt Lake City, UT, USA, pp. 335-346, DOI: 10.1109/SC.2016.28.

[42] Georgakoudis G., Laguna I., Nikolopoulos D.S., Schulz M (2017): *REFINE: Realistic Fault Injection via Compiler-based Instrumentation for Accuracy, Portability and Speed*. In Procs. of ACM Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '17), New York, USA, Article 29, pp.1–14. DOI: https://doi.org/10.1145/3126908.3126972

[43] Oliveira D., Fratin V., Navaux Ph., Koren I., Rech P. (2017): *CAROL-FI: An Efficient Fault-Injection Tool for Vulnerability Evaluation of Modern HPC Parallel Accelerators.* In Procs. of ACM Int. Conference on Computing Frontier, May 15-17, Siena, Italy.

[44] Li G., Pattabiraman K., Cher C., Bose P. (2016): *Understanding Error Propagation in GPGPU Applications*, in SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 240-251. DOI: 10.1109/SC.2016.20

[45] Tselonis S. Gizopoulos D. (2016): *GUFI: A Framework for GPUs Reliability Assessment*, in Procs. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 90-100. DOI: 10.1109/ISPASS.2016.7482077

[46] Mutlu, B.O., Kestor, G., Manzano, J., Unsal, O., Chatterjee, S., Krishnamoorthy, S. (2018): *Characterization of the Impact of Soft Errors on Iterative Methods.* In Procs. of 25th IEEE Int. Conf. on High Performance Computing (HiPC-2018), pp. 203-214. https:/doi.org/10.1109/HiPC.2018.00031

[47] Mutlu, B.O., Kestor, G., Cristal, A., Unsal, O., Krishnamoorthy, S. (2019): *Ground-Truth Prediction to Accelerate Soft-Error Impact Analysis for Iterative Methods*. In Procs. of IEEE 26th Int. Conf. on High Performance Computing (HiPC-2019), pp.333-344. https:/doi.org/10.1109/HiPC.2019.00048

[48] Mutlu B.O. (2019): *An Extensive Study on Iterative Solver Resilience: Characterization, Detection and Prediction*, University of Cataluña, 150 pages, Sept. 2019.

[49] Sangchoolie B., Pattabiraman K., Karlsson J. (2017): *One Bit is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors*. 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Denver, CO, USA, pp. 97-108. DOI: 10.1109/DSN.2017.30

[50] Elliot J., Hoemmen M., Mueller F. (2014): *Tolerating Silent Data Corruptions in Opaque Preconditioners*, SAND2014-3452C.

[51] Patrick G. Bridges P.G., Hoemmen M., Ferreira K.B., Heroux M.A., Soltero Ph., Brightwell R. (2012): *Cooperative Application/OS DRAM Fault Recovery*. In Procs. Euro-Par (Eds: Alexander M. et al.). (Eds.), Part II, LNCS 7156, pp. 241–250, Springer-Verlag, Berlin, Heidelberg.

[52] Coleman E., Jamal A., Baboulin M., Khabou A., Sosonkina M. (2017): *A Comparison of Soft-Fault Error Models in the Parallel Preconditioned Flexible GMRES*. In Procs. in Int. Conf. Parallel Processing and Applied Mathematics, Sept. 2017, Lublin, Poland, pp. 36-46. DOI: 10.1007/978-3-319-78024-5_4

[53] Ashraf R.A., Hukerikar S., Engelmann Ch. (2018): *Pattern-based Modeling of Multiresilience Solutions for High-Performance Computing*. In Procs. ACM/SPEC International Conference on Performance Engineering (ICPE '18), NY, USA, pp. 80–87. DOI: 10.1145/3184407.3184421

LIST OF FIGURES

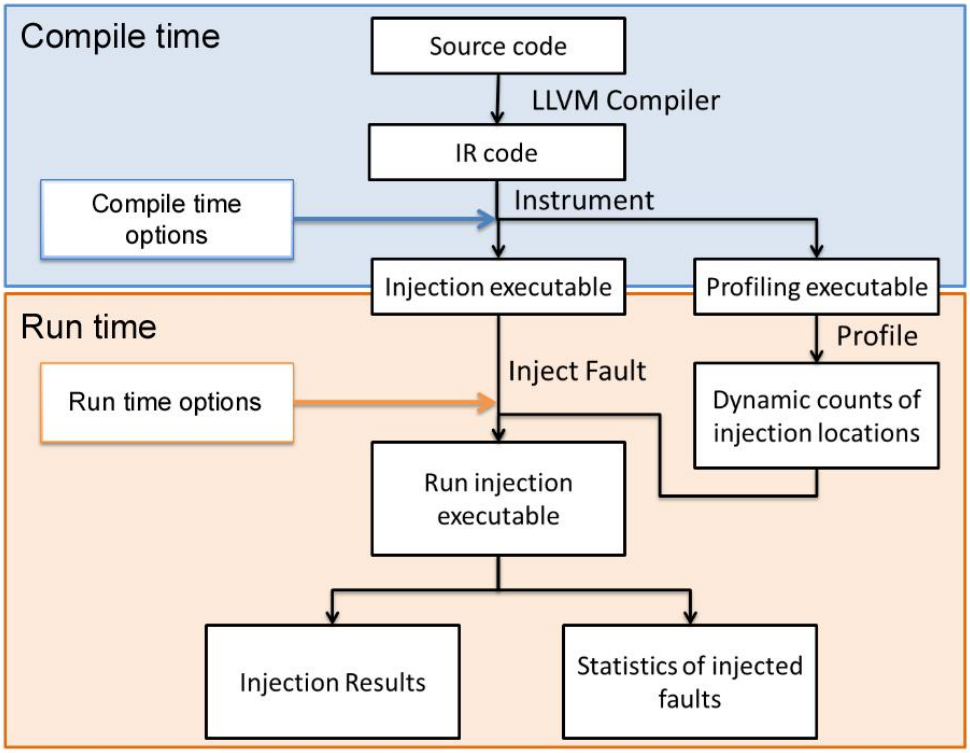

Fig. 1 LLFI workflow and design (taken from [1]).

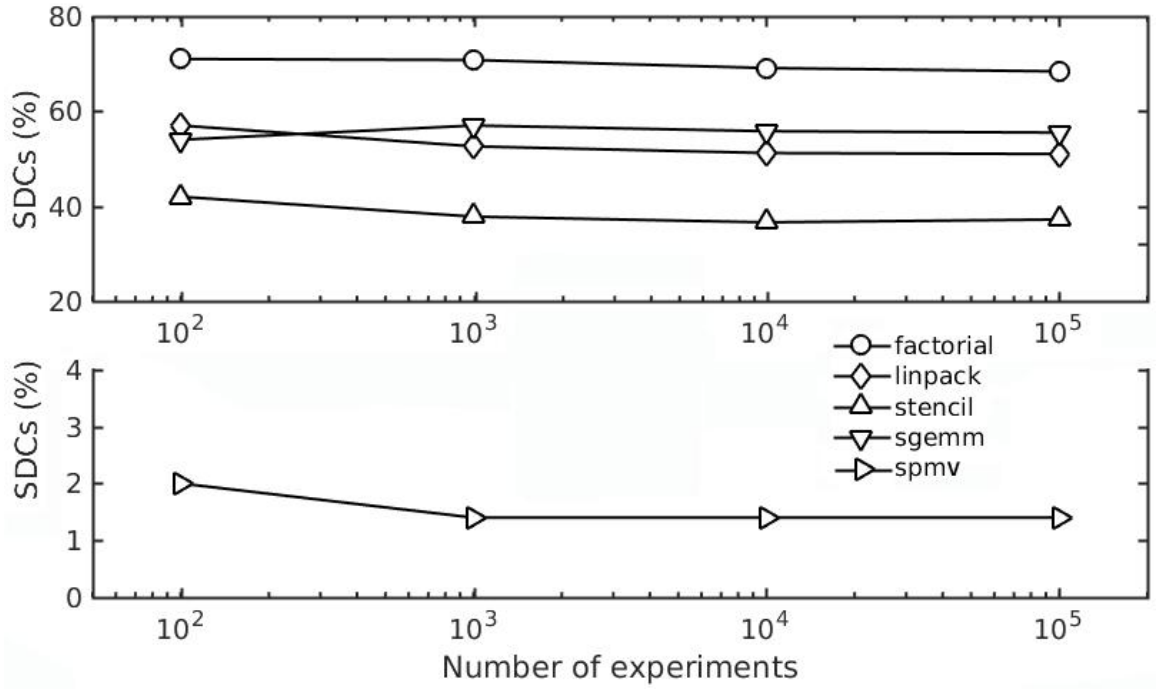

Fig. 2 SDC rate as a function of the number of experiments performed with single bit-flip injection for the five benchmarks considered.

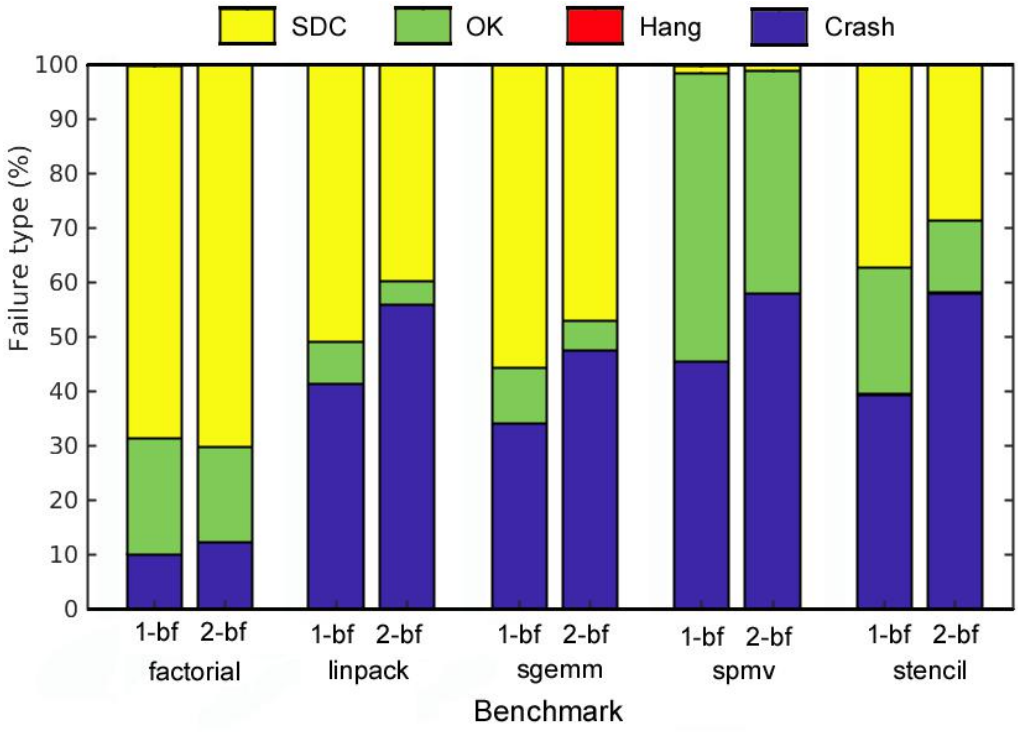

Fig. 3 Comparison of failure type rates after 100,000 experiments in two fault injection scenarios: single-bit and double-bit flip injection (1-bf and 2-bf, respectively). Hang rate is undiscernible in the plot scale of the figure (its values for single-bit flip injection are given in Table 2).

For a given linear problem $Ax=b$, defined by A and b:

1- Start

Chose an initial guess x_0 and a dimension m of the Krylov subspace. Allocate the (m+1) x m matrix H_m and initialize all its entries h_{ij} to 0.
Set preconditioner $M := L_0U_0$, being L_0 and U_0 the ILU(0) triangular factors of A.

2- Arnoldi process

$r_0 := b - Ax_0$, $\beta := ||r_0||_2$, $v_1 := r_0 / \beta$
for $j = 1, 2, \ldots, m$ ***do***
  $z_j = M^{-1}v_j$ (equivalent to directly solve $L_0U_0 z_j = v_j$ for z_j)
  $w := A z_j$
  for $i = 1, 2, \ldots, j$ ***do***
    $h_{i,j} := w \cdot v_i$ (dot product)
    $w := w - h_{i,j} v_i$
  end
  $h_{j+1,j} := ||w||_2$
  $v_{j+1} := w / h_{j+1,j}$
  $V_m := [v_1, \ldots, v_m]$
  $H_m := h_{i,j}$ (with $1 \le i \le j+1; 1 \le j \le m$)
end

3- Form the approximate solution

$y_m := \text{argmin}_y ||H_m y - \beta e_1||_2$
$x_m := x_0 + M^{-1}V_m y_m$ (with $e_1 = [1, 0, \ldots, 0]^T$)

4- Restart

If convergence is reached then return x_m, else set $x_0 \leftarrow x_m$ go to step 2

Fig. 4 Standard GMRES algorithm, preconditioned with ILU(0) and m-restarted.

For a given linear problem $Ax=b$, defined by A and b:

5- Start

Chose an initial guess x_0 and a dimension m of the Krylov subspace. Allocate the (m+1) x m matrix H_m and initialize all its entries h_{ij} to 0.
Compute $A \approx L_0U_0$, being L_0 and U_0 the ILU(0) factors of A.

6- Arnoldi process

$r_0 := b - Ax_0$, $\beta := \|r_0\|_2$, $v_1 := r_0 / \beta$
for $j = 1, 2, \ldots, m$ ***do***
  solve $L_0U_0 z_j = v_j$ for z_j using standard GMRES (ILU-preconditioned)
    (equivalent to $z_j := M_j^{-1}v_j$, with variable preconditioner M_j)
  $w := A z_j$
  for $i = 1, 2, \ldots, j$ ***do***
    $h_{i,j} := w \cdot v_i$ (dot product)
    $w := w - h_{i,j} v_i$
  end
  $h_{j+1,j} := \|w\|_2$
  $v_{j+1} := w / h_{j+1,j}$
  $Z_m := [z_1, \ldots, z_m]$
  $H_m := h_{i,j}$ (with $1 \le i \le j+1; 1 \le j \le m$)
end

7- Form the approximate solution

$y_m := \text{argmin}_y \|H_m y - \beta e_1\|_2$
$x_m := x_0 + Z_m y_m$ (with $e_1 = [1, 0, \ldots, 0]^T$)

8- Restart

If convergence is reached then return x_m, else set $x_0 \leftarrow x_m$ go to step 2

Fig. 5 Preconditioned Flexible GMRES algorithm. Tthe variable preconditioner M_j is implicitly set when applying the Standard GMRES-ILU(0) to solve for z_j in the Arnoldi process.

For a given linear problem $Ax=b$, defined by A and b:

1- Start

Chose an initial guess x_0 and a dimension m of the Krylov subspace. Allocate the (m+1) x m matrix H_m and initialize all its entries h_{ij} to 0.
Set preconditioner $M := L_0U_0(I+E_k)$, being L_0 and U_0 the ILU(0) triangular factors of A and $E_k \approx E$ a truncated-SVD aproximation with $k<<n$.
Compute $(I+E_k)^{-1}$

2- Arnoldi process

$r_0 := b - Ax_0$, $\beta := \|r_0\|_2$, $v_1 := r_0 / \beta$
for $j = 1, 2, \ldots, m$ ***do***
  $z_j = M^{-1}v_j$ (direct solve $L_0U_0t_j = v_j$ for t_j ; and $z_j = (I+E_k)^{-1}t_j$)
  $w := Az_j$
  for $i = 1, 2, \ldots, j$ ***do***
    $h_{i,j} := w \cdot v_i$ (dot product)
    $w := w - h_{i,j} v_i$
  end
  $h_{j+1,j} := \|w\|_2$
  $v_{j+1} := w / h_{j+1,j}$
  $Z_m := [z_1, \ldots, z_m]$
  $H_m := h_{i,j}$ (with $1 \le i \le j+1; 1 \le j \le m$)
end

3- Form the approximate solution

$y_m := \text{argmin}_y \|H_m y - \beta e_1\|_2$
$x_m := x_0 + Z_m y_m$ (with $e_1 = [1, 0, \ldots, 0]^T$)

4- Restart

If convergence is reached then return x_m, else set $x_0 \leftarrow x_m$ go to step 2

Construction of E_k for the preconditioner

E: n x n matrix, defined $:= (L_0U_0)^{-1}A - I$
Ω: n x (k+p) random matrix, with $k<<n$ and p small (typically $p \le 10$)
Sample E: $S = E\Omega$
$V :=$ orthonormalized(S)
Form V^TE and compute SVD$(V^TE) = X\Sigma Y^T$
Truncate X, Σ, Y into X_k, Σ_k, Y_k (first k singular vectors/values)
Then $E_k = (VX_k)\Sigma_k Y_k^T$

Fig. 6 Preconditioned Randomized-SVD -based GMRES algorithm. The preconditioner construction and randomized version of the SVD are detailed in the last portion of the pseudocode.

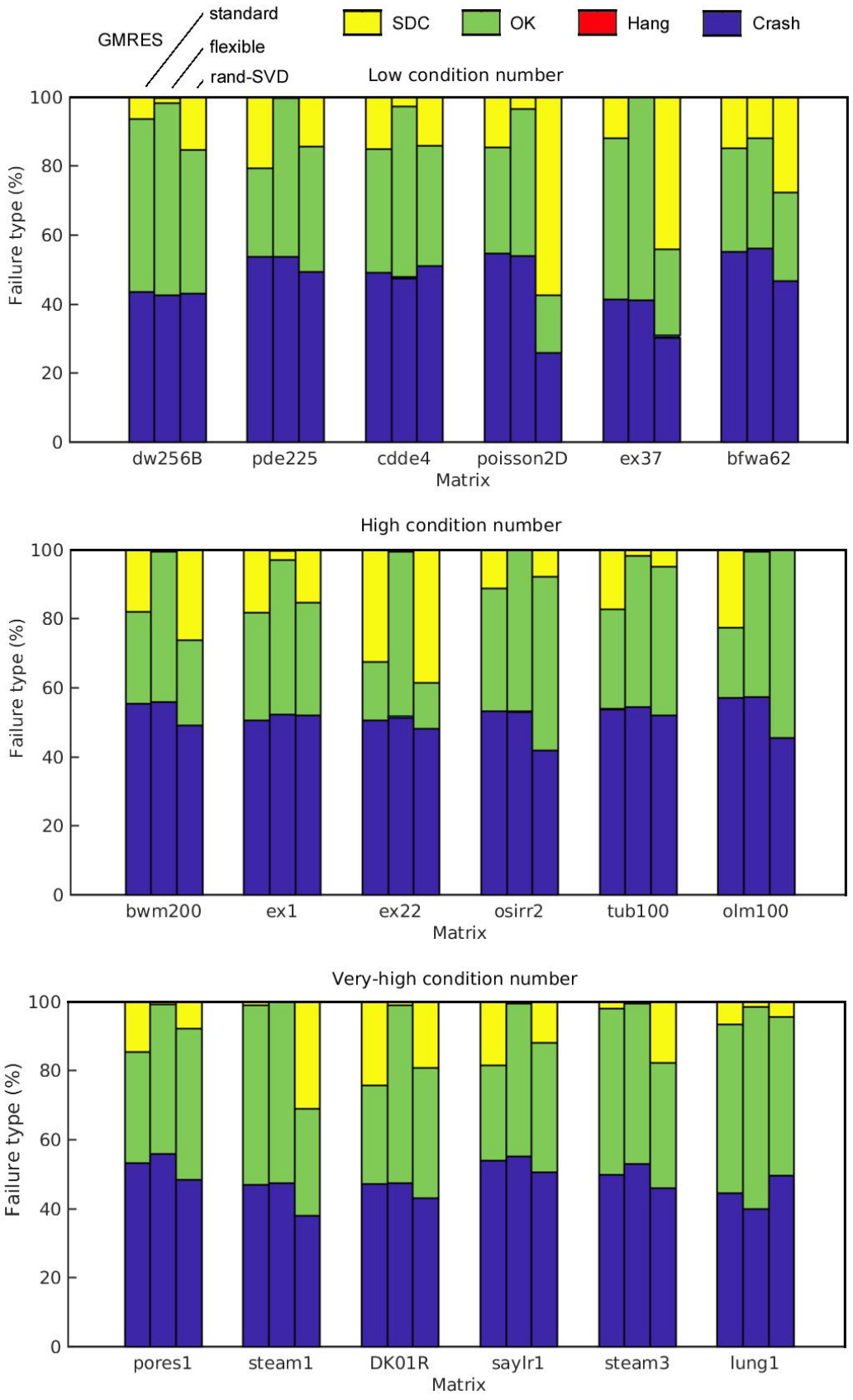

Fig. 7 Comparison of failure type rates for the set of matrices. OK executions correspond to the strict criteron RHS=$RHS_{baseline}$ (hang rate is undiscernible in the plot).

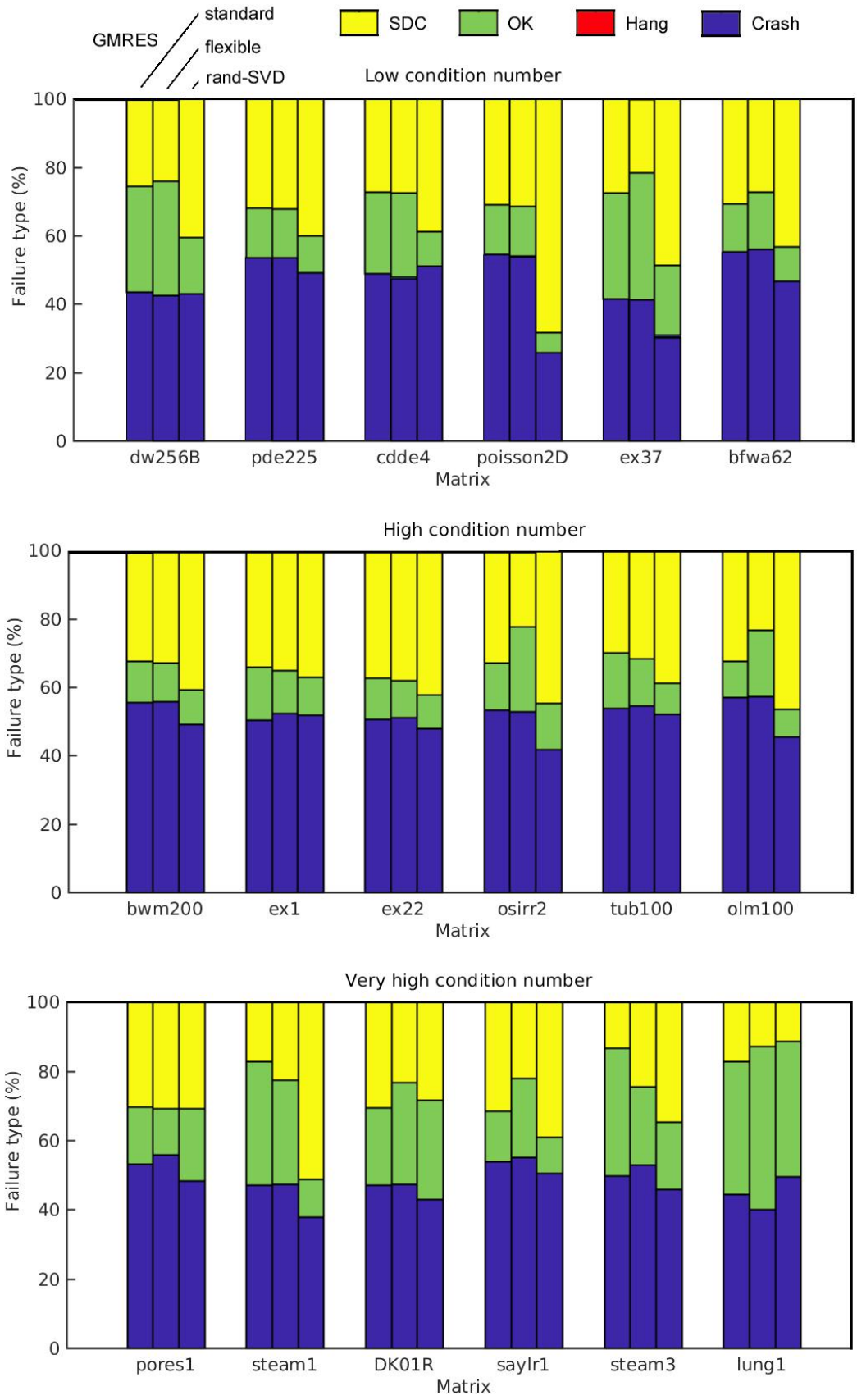

Fig. 8 Comparison of failure type rates for the set of matrices. OK executions correspond to the criterion: number of iterations needed to atain RHS ≤ tolerance, equal or lower than the baseline case (hang rate is again undiscernible in the plot).

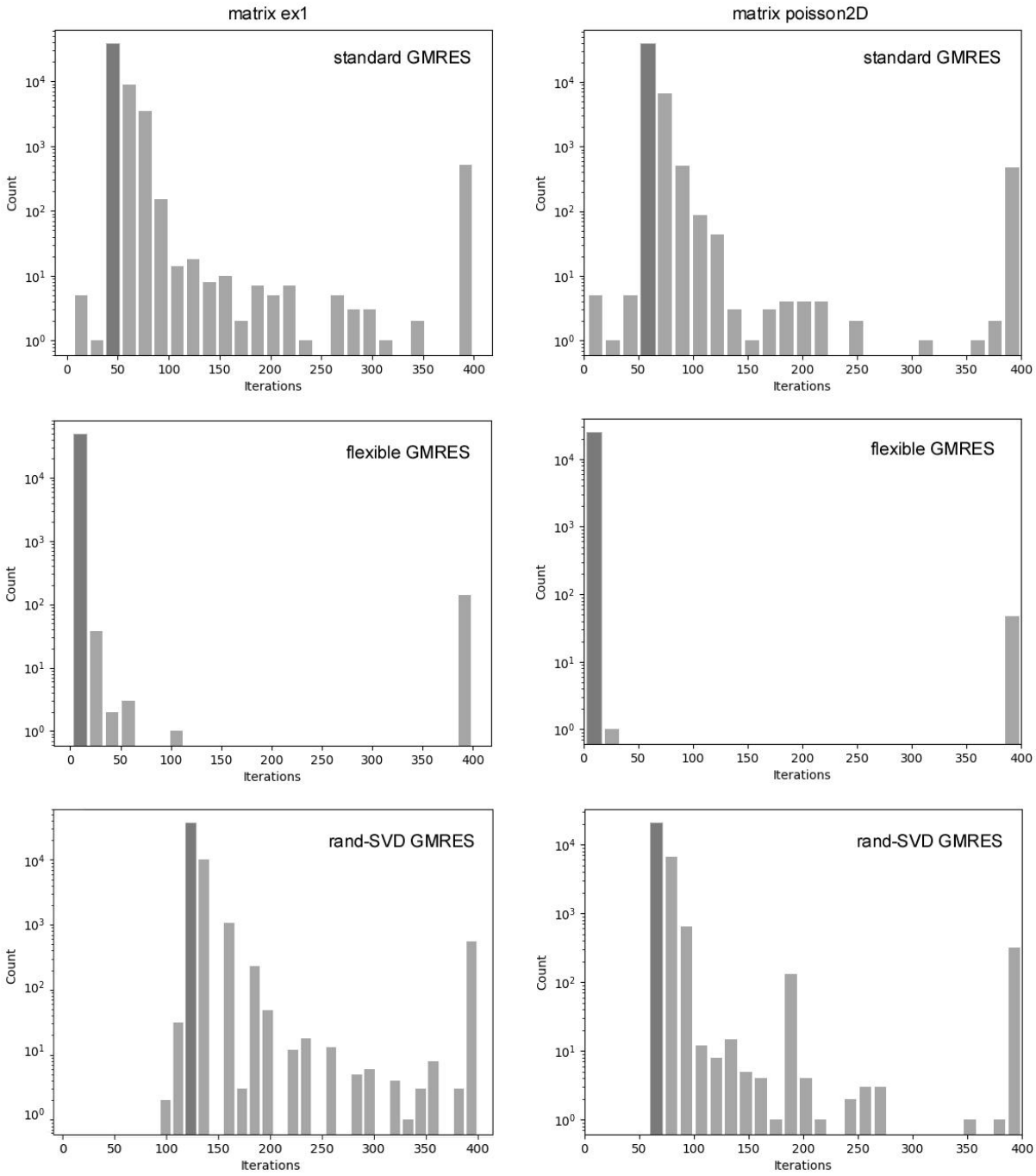

Fig. 9 Histograms of iterations-to-converge (the maximum 400 iterations corresponds to a non-convergence scenario). Matrices *ex1* and *poisson2D* are shown as tests solved with the three GMRES versions. The dark gray bar corresponds to iter=$\text{iter}_{\text{baseline}}$.

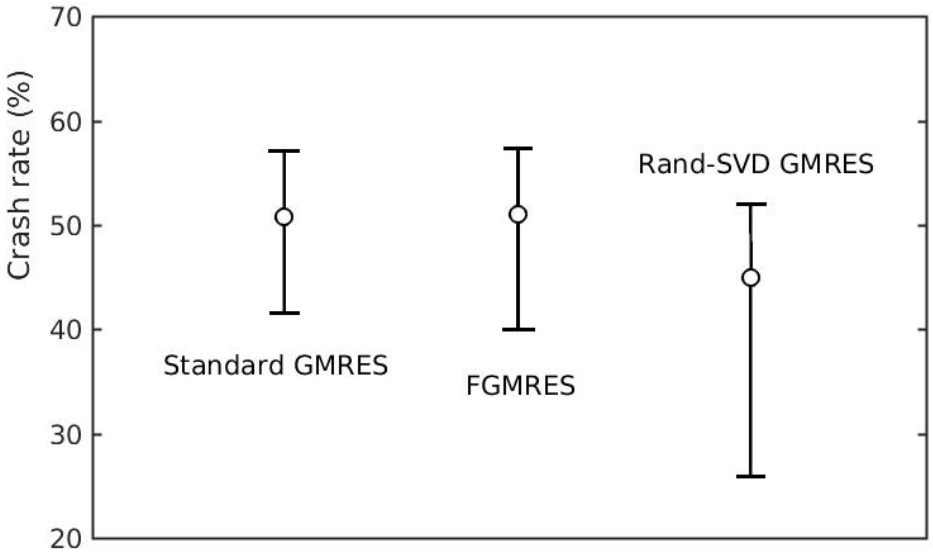

Fig. 10 Crash rate comparison for the Standard, Flexible and Randomized-SVD -based GMRES algorithms. Each percentage range is computed across the 18 matrices (average value is indicated by the circle).