Exploiting Asynchrony from Exact Forward Recovery for DUE in Iterative Solvers

Luc Jaulmes Marc Casas Miquel Moretó
Eduard Ayguadé Jesús Labarta Mateo Valero
Barcelona Supercomputing Center - Centro Nacional de Supercomputación, Barcelona, Spain
Universitat Politècnica de Catalunya, Barcelona, Spain
name.surname@bsc.es

ABSTRACT

This paper presents a method to protect iterative solvers from Detected and Uncorrected Errors (DUE) relying on error detection techniques already available in commodity hardware. Detection operates at the memory page level, which enables the use of simple algorithmic redundancies to correct errors. Such redundancies would be inapplicable under coarse grain error detection, but become very powerful when the hardware is able to precisely detect errors.

Relations straightforwardly extracted from the solver allow to recover lost data exactly. This method is free of the overheads of backwards recoveries like checkpointing, and does not compromise mathematical convergence properties of the solver as restarting would do. We apply this recovery to three widely used Krylov subspace methods, CG, GMRES and BiCGStab, and their preconditioned versions.

We implement our resilience techniques on CG considering scenarios from small (8 cores) to large (1024 cores) scales, and demonstrate very low overheads compared to state-of-the-art solutions. We deploy our recovery techniques either by overlapping them with algorithmic computations or by forcing them to be in the critical path of the application. A trade-off exists between both approaches depending on the error rate the solver is suffering. Under realistic error rates, overlapping decreases overheads from 5.37% down to 3.59% for a non-preconditioned CG on 8 cores.

1 Introduction

As High-Performance Computing (HPC) systems continuously scale up and chips shrink accordingly towards the exascale era [7, 8], the increased risk of faults in a computing system [38] causes reliability features to flourish in Operating Systems (OS) and processors [22, 26, 36]. Though the studies on actual software error rates are few [28, 32], they indicate that the increase in faults due to decreasing size and increasing multiplicity of components is real, and that we can not rely solely on hardware to be resilient anymore. This tendency is expected to be aggravated by the reduction of voltages required for exascale systems [14, 24].

This paper is published in the Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2015), 2015, pp. 53:1-53:12. The final publication is available at http://dx.doi.org/10.1145/2807591.2807599.

Memory cells are among the most vulnerable hardware components [30]. They are typically protected by Error Correcting Codes (ECC) [31] implemented at the hardware level. Through these ECCs all single bit flips are covered and corrected. Despite the fact that multiple bit flips are usually detectable, they can cause faults that hardware ECC protection is unable to correct. This kind of errors are called Detected and Uncorrected Errors (DUE). For instance, on modern x86 and AMD64 architectures, a DUE is reported in specific registers. When such a register is set, the processor generates a machine check exception that can be caught at the OS level [2, 23]. Different kinds of DUE can be caught and signalled to the software stack and thus assimilated to some data loss, provided the architectural state is safe.

The responsibility of reacting against DUE is handed to the software stack. Some straightforward approaches like cancelling the affected process or relocating a faulty memory page to another physical location may be effective against low fault rates, but they are insufficient against the predicted rates that processors will suffer in the future. Also, very aggressive resilience strategies like process triplication are completely impractical unless we face very high fault rates [17]. Therefore, intermediate solutions that recompute an approximation of the lost data [27] or that save the process state in a checkpoint with a certain frequency have been extensively used [11,29,34]. However, most of these solutions involve backward recoveries and thus significant slowdowns.

The application itself may be able to handle the error and terminate cleanly [4] or perform some sort of recovery procedure relying on Algorithmic-Based Fault Tolerance (ABFT), which has been extensively applied to MPI programs [9, 16, 27], as well as shared memory programming models [35, 37]. Algorithmic approaches have demonstrated to be more efficient than backward recoveries like checkpointing-rollback. However, ABFT techniques are application dependent and their overheads are still significant, which are two issues that have avoided the wide-spread usage of algorithmic resilience. In this paper we aim to reduce the impact of these two main drawbacks that algorithmic-based resilience has. The proposed ABFT methods to deal with DUE are based on very simple algebraic relations that do not require any kind of deep understanding of the algorithm and can be almost always derived for iterative methods. Such kind of techniques are not applicable with some patterns of data loss, such as those implied by a fail-stop error model in a distributed memory environment, but they become useful when smaller amounts of information are lost. When a DUE is signalled,

we always discard the whole memory page where affected data resides, as OSs do for bus and memory ECC errors, which gives us an error granularity that is exploitable for redundancy relations. The overheads related to ABFT mechanisms are reduced by overlapping them with algorithmic computations. Since the responsibility of such overlapping is left to the runtime system, we do not significantly increase the programming burden.

This paper proposes an integrated resilience approach, where the error detection is performed by hardware mechanisms that report DUE to the OS, which identifies lost data at a memory page level and triggers a signal caught by the application. We use the OmpSs task-based data-flow programming model [15], in which serial code is split into several pieces, called tasks, that are dynamically scheduled according to data dependencies explicitly expressed by the programmer. We combine the OmpSs annotations with MPI to scale our implementation up to over thousand cores. We demonstrate the feasibility of our approach by applying it to relevant iterative methods of the Krylov subspace family: the Conjugate Gradient (CG), Bi-Conjugate Gradient Stabilized (BiCGStab), and Generalized Minimal RESidual (GMRES) [3], and implementing it for CG. The main contributions of this paper are:

- A general resilience solution for DUE based on straightforward algorithmic recoveries that operate at memory page level. With the lowest error injection rate considered, corresponding to one expected error per baseline execution time, the overhead of this technique is 5.37%, whereas that of the checkpointing-rollback technique is close to 55%.
- An asynchronous and programmer transparent variant of our recovery implementation that reduces the overhead down to 3.59% under the lowest error rate, and that offers a trade-off between low overhead and convergence rate for higher error rates.
- An exhaustive and comprehensive evaluation, using real world matrices, of our method against a more sophisticated algorithm-specific restart method, derived from Langou et al.'s Lossy Approach [27], and checkpointing-rollback mechanisms. We consider different parallel scenarios from 8 up to 1024 cores and we show that our methods always improve the performance of the above mentioned state-of-the-art methods.
- A mathematical proof of stronger properties of the Lossy Approach recovery.

The rest of this paper is organized as follows: In Section 2, we explain how to recover from hardware detected memory errors by using inherent redundancy, while Section 3 shows how to use this redundancy to make Krylov subspace methods resilient, as well as implementation details of this methodology for CG. Section 4 introduces the methods with which we compare our recoveries, and the next section provides numerical validation results of our implementations. Section 6 examines related work and, finally, Section 7 provides our concluding remarks.

2 Forward Interpolation Recovery

2.1 Error Detection and Reporting

Due to the advent of faults, many processors have specific registers dedicated to signalling errors to the OS layer. On modern x86 and AMD64 architectures for example, a memory controller discovering data that is incoherent with the ECC, while accessing or periodically scrubbing it, reports it in a specific register [2, 23]. For memory pages, when the corrected errors exceed a threshold, the OS transparently relocates the page at another physical location. When a DUE is reported, the OS kills the affected process. This feature is known as memory page retirement on Solaris, and soft or hard page offlining in Linux kernels [26, 36].

In practice, application termination after a page failure is done by a `SIGBUS` signal that can be caught. This signal also specifies the failing memory addresses. To make an application resilience aware, it is sufficient to catch that signal and request a new hardware memory page at the same virtual address. Thus, to be resilient against memory DUE, an HPC application simply has to be able to replace lost data.

All data consists of constant data, which does not change during execution time (matrix, preconditioners, right-hand side), and dynamic data, which may be modified. Constant data is assumed to be saved to a reliable backing store, from which it is reloaded when errors are detected, similarly to other work using memory-page level fault models [6]. While there typically is not enough information available in the solver to recover constant data, there may likely be where the matrix was generated or read. Alternately, such data could be protected by software ECC at low cost, by exploiting the fact that this second ECC tier only needs to correct, and not detect errors [39], while the long lifetime of the data allows to increase codeword size and thus to reduce overheads further. Thus, only dynamic data needs to be made recoverable.

2.2 Extracting Redundancies of Linear Solvers

Linear iterative solvers perform operations like matrix-vector multiplications $q = Ap$, linear combinations $u = \alpha v + \beta w$, and combinations of the above, e.g. the very common residual $g = b - Ax$, where A is a matrix while q, p, u, w, g, b and x are vectors and α and β are scalars. In many cases the left and right hand side of these operations coexist during the whole execution of the solver.

In some cases, we know that such a relation between vectors holds true (minus round-off errors) without having to recompute them. For example, if we define $x' = x + p$ and $g' = g - q$ with the above notations, then we have $g' = b - Ax'$. Finally, similar relations can hold true by construction, without ever having been computed. Thus by analysing an iterative solver, we find redundancies expressed in terms of explicit or implicit relations between data.

Trivially, if any vector g, q, u is lost or partially corrupted, it can be recovered by repeating the operation that produced it, or applying a relation known to hold true. Given the inverses of A, α, β and other potential operands of a relation, it would also be possible to recover a lost or corrupted p, v, w, t or x. However, computing the inverse of the whole matrix A is as expensive as running the whole iterative method. Thus, recoveries based on trivial redundancies are only applicable if a small portion of the data structures involved in a relation is lost. This is the case with our error model since modern hardware is able to report errors at memory page level. In order to operate at such fine grain level, the redundancies must be decomposed in terms of relations between small blocks of data.

Table 1: Block relations used for recoveries

Block relation, recover lhs	Inverted relation, recover rhs
$q_i = \sum_{j=0}^{n-1} A_{ij} p_j$	$A_{ii} p_i = q_i - \sum_{j \neq i} A_{ij} p_j$
$u_i = \alpha v_i + \beta w_i$	$w_i = (u_i - \alpha v_i)/\beta$
$g_i = b_i - \sum_{j=0}^{n-1} A_{ij} x_j$	$A_{ii} x_i = b_i - g_i - \sum_{j \neq i} A_{ij} x_j$

2.3 Block Decomposition

The relations exposed previously, decomposed in n blocks, are listed in Table 1. We use the normal block relation to recover the left-hand side (lhs) of a relation, and the inverse of this relation for the right-hand side (rhs). If we know that a diagonal block is non-singular, e.g. when A is Symmetric Positive Definite (SPD), we solve the inverse block relations with a direct solver. Otherwise we solve this relation in the sense of least squares for the full columns of the matrix corresponding to the lost memory page as input, similarly to what Agullo et al. proposed for restart methods [1].

The formula for x_i's recovery, shown at the bottom of the right hand side of Table 1 has been used by Chen [10] to recover the iterate, in complement of implicit checkpointing methods. We however protect all vectors with interpolation methods, thus requiring no checkpoints. Exploiting these relations for recovery is a novel idea, since all previous work on making Krylov-subspace solvers fault-tolerant relies on a fail-stop failure model in a distributed memory environment. The required information to use redundancy of linear relations is then not available since corresponding parts of different vectors are lost simultaneously.

Furthermore, the granularity of the blocks of lost data in our recoveries is very different from the one in the context of process failure, which allows different and faster recoveries. Indeed, our block decomposition is dictated by the underlying layers (hardware detection, OS, runtime) that do the DUE reporting. This means the block size that we use as granularity for recovery is a memory page of typically 4K bytes, thus 512 double precision floating-point values.

Any DUE in our data protected by relations can thus be rectified by applying a small amount of computations, at worst factorizing a diagonal block of a matrix if one is used by that relation. This is a forward recovery scheme, since we can continue executing the program with our interpolated replacement data and the data that is not affected by the error. When A's diagonal block is non-singular or a linear relation is used, we can even guarantee the exact same data as was lost for all relations (up to rounding errors), thus guarantee the same convergence rate as when the algorithm is not subject to faults. These recovery operations are usually small compared to the total computations, since matrix dimensions reach up to more than a million rows for real-life problems, as available in the University of Florida sparse matrix collection [13].

2.4 Dealing with Multiple Errors

Our approach does not have to unrealistically assume that no more than one error happens at the same time. Indeed, our techniques can easily handle multiple errors (discovered simultaneously) in most situations. Errors can be recovered if there is no more than one impacting each instance of a blocked linear relation expressed in Table 1. However, if simultaneous errors impact a single relation we have two possible scenarios:

Listing 1: CG pseudo code with redundancy relations

```
1  ϵ_old ⇐ +∞
2  g ⇐ b − Ax                                   g = b − Ax
3  for t in 0..t_max
4    ϵ ⇐ ||g||^2
5    if ϵ < tol : break
6    β ⇐ ϵ/ϵ_old
7    d ⇐ βd + g                                 d = A^{-1}q     g = b − Ax
8    q ⇐ Ad                                     see 3.1.1
9    α ⇐ ϵ/ < q, d >                            q = Ad          d = A^{-1}q
10   x ⇐ x + αd                                 d = A^{-1}q     x = A^{-1}(b − g)
11   g ⇐ g − αq                                 q = Ad          g = b − Ax
12   ϵ_old ⇐ ϵ
```

1. *Simultaneous errors in a single vector* are not a problem for our recovery strategy. This is trivial for vectors recovered from linear relations, and straightforward for submatrix relations [27]. For two failed blocks i and j, we can combine both block relations:
$$\begin{pmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{pmatrix} \begin{pmatrix} x_i \\ x_j \end{pmatrix} = \begin{pmatrix} b_i - g_i - \sum_{k \neq i,j} A_{ik} x_k \\ b_j - g_j - \sum_{k \neq i,j} A_{jk} x_k \end{pmatrix}$$
This relation is generalizable to any number of errors, but factorizing the submatrix becomes more expensive.
2. *Simultaneous errors on related data*, e.g. both q_i and p_i for a given i in a $q = Ap$ relation. Assuming there is no other relationship that allows to recover these data, we may fall back to a restart method, e.g. the Lossy Restart which is adapted from the Lossy Approach [27] to fit our error model (see Section 4.3).

In conclusion, our forward interpolation recovery relies on very simple redundancy relations that are easy to identify in any iterative method and they can efficiently operate at memory page level. Also, our scheme can deal with multiple errors, but that may imply a more expensive recovery or, at worst, the usage of a restart method as fallback.

3 Applying to Iterative Solvers

3.1 Making Redundancies Explicit

DUE are reported when a faulty operation is made or when trying to access data that is corrupted. Even for a memory error, if the OS discovers data corruption while periodically scrubbing memory pages, that data is marked as "poisoned". No error is signalled until it is accessed, in the hope the page will be freed or overwritten completely.

So in order to make an iterative solver resilient with our technique, it is sufficient to find for each operand of each operation done by the solver a relation that

- either allows to compute the result without this operand – thus picking an alternate formulation
- or allows to recover the operand, and then compute the result of the operation.

Note that the main difference between this work and previous application-level recoveries for the same iterative solvers is the error model: since we do not consider complete failure of a node, we do not incur the loss of *a part of every vector*, which would render the relations we use here inapplicable. Over the next subsection we explain in detail how three commonly used iterative methods, CG, BiCGStab, and GMRES, can be protected through redundancy relations.

Listing 2: CG with d double-buffered

```
1  for t in 0..t_max
2    ...
3    d_1 ⇐ βd_2 + g
4    q ⇐ Ad_1
5    α ⇐ ϵ/ < q, d_1 >
6    x ⇐ x + αd_1
7    ...
8    t++
9    d_2 ⇐ βd_1 + g
10   q ⇐ Ad_2
11   α ⇐ ϵ/ < q, d_2 >
12   x ⇐ x + αd_2
13   ...
```

Listing 3: BiCGStab pseudo code with redundancies listed

```
1  g, r, d ⇐ b − Ax
2  ρ ⇐< g, r >
3  for t in 0..t_max
4    q ⇐ Ad                      d double-buffered
5    α ⇐ ρ/ < q, r >             q = Ad
6    s ⇐ g − αq                  g = b − Ax          q = Ad
7    t ⇐ As                      s = g − αq
8    ω =< t, s > / < t, t >      t = As              s = A^{-1}t
9    x ⇐ x + αd + ωs             x = A^{-1}(b − g)   d = A^{-1}q
10   g ⇐ s − ωt                  t = As              s = A^{-1}t
11   check convergence
12   ρ_old ⇐ ρ
13   ρ ⇐< g, r >                 g = b − Ax
14   β ⇐ ρ/ρ_old ∗ α/ω
15   d ⇐ g + β(d − ωq)           q = Ad              d = A^{-1}q
```

Listing 4: GMRES pseudo code

```
1  for t in 0..t_max
2    g ⇐ b − Ax
3    v^0 ⇐ g/||g||_2
4    for l in 0..m − 1
5      w ⇐ Av^l
6      for k in 0..l
7        h_{k,l} ⇐< w, v^k >
8        w ⇐ w − h_{k,l}v^k
9      h_{l+1,l} ⇐ ||w||_2
10     v^{l+1} ⇐ w/h_{l+1,l}
11   solve H = QR
12   y ⇐ R^{-1}Q^T||g||_2 e_1
13   x ⇐ x + Σ_{l=0}^{m−1} y_l v^l
14   check convergence
```

3.1.1 Conjugate Gradient

The pseudo-code for CG is given in Listing 1 [33], with the relations used for recovering each accessed data annotated on the right. Relations are written as whole-matrix relations for the sake of readability, but we use the memory page grained system described previously. Whenever possible, the relation that last produced data is used, which is not possible when data is updated in place. We know that the algorithm conserves the relation $g = b - Ax$, and we can define an alternative way of computing q besides performing Ad from the update formula of $d \Leftarrow \beta d + g$, which is $q \Leftarrow \beta q + Ag$. We see that when computing q, if a page of d is missing, we may still compute q through an alternate formulation. When the whole matrix-vector multiplication is done, we may then recover d using "$d = A^{-1}q$", in order to continue computations. However, it is impossible to use this relation when updating d. Let us reconsider the block recovery:

$$d_i = A_{ii}^{-1}\big(q_i - \sum_{j \neq i} A_{ij} d_j\big)$$

At this stage in the update of d, all pages $d_0, ..d_{i-1}$ are at iteration $t+1$, and all pages $d_{i+1}..d_{n-1}$ are at iteration t.

We have two possibilities for recovery:

1. Compute q with $\beta q + Ag$, then factorize A_{ii} to get d.
2. Perform *double buffering*, thus have two copies of a vector, either d or q, and use them alternately from one iteration to the next to remove in place updates.

The first option, though possibly more elegant, would imply taking rather considerable distances from the original algorithm. It is also arguable that the $\beta q + Ag$ operation might need to be protected, since a matrix-vector multiplication is the most computationally intensive and longest operation in a CG iteration. We thus opted for the latter option, and unrolled the loop to use two d vectors alternately, as illustrated in Listing 2. Code unchanged by this transformation is skipped. This solution adds redundancy to the method at the cost of some minimal memory overhead.

3.1.2 Bi-Conjugate Gradient Stabilized

BiCGStab is one of the generalizations of CG to matrices that are non SPD. The pseudo-code for this method and the relations that may be used to make it resilient are presented in Listing 3, similarly to what has been done for CG. r is constant, along with the usual A and b. BiCGStab exhibits more redundancies than CG, and only an example set of relations that can be used is shown.

With $q = Ad$, $s = g - \alpha q$ and $t = As$, updating g can be rewritten $g \Leftarrow g - \alpha Ad - \omega As$. Thus we have another way of computing g if for example q is faulty, but we also verified that the algorithm still conserves $g = b - Ax$.

Note that other assignments can also be expressed as slightly more complicated updates. We have e.g. $s \Leftarrow s - \omega t - \alpha q$. The reverse also holds true, from the update of x we may get a direct relation such as $x = A^{-1}(b - s + \omega t)$

3.1.3 Generalized Minimal RESidual

The code for GMRES is available in Listing 4. Each iteration of GMRES consists of running the Arnoldi method - the part creating an orthogonal basis of vectors spanning $(g, Ag, ..., A^{(m-1)}g)$ and an associated upper-Hessenberg matrix H - followed by a QR decomposition of this matrix H through Givens rotations. We may then increment the iterate by the solution y of $min_y||g - Hy||$.

Protecting the biggest part of the data, which is the v^k vectors, is pretty straightforward thanks to the Hessenberg matrix. At any time, we have at step t,

$$l > 0 \text{ and } l < t \Rightarrow v^l = \frac{1}{h_{l,l-1}}\left(Av^{l-1} - \sum_{k=0}^{l-1} h_{k,l-1} v^k\right)$$

Thus the redundancy kept in the Hessenberg matrix' elements allows us to recover any Arnoldi vector under our error model.

Note that it is possible (and usual) to build the QR decomposition of the Hessenberg matrix H as the Arnoldi method goes, by computing the Givens rotation that corresponds to each new vector of the Arnoldi method. Q is thus computed as the set of Givens rotations, and $Q^T||g||_2 e_1$ is also updated at every step. Thus we could use the redundancy $H = QR$ by keeping a copy of H even while we build R:

- Givens rotations are easily deducible from H, thus Q and R are recoverable from H
- Givens rotations are easily invertible, since inverting a rotation means rotating by the opposite angle. Thus H is recoverable from Q and R.

Even though space is a limiting factor in GMRES, the H and R matrices are respectively upper Hessenberg and upper triangular of size $m(m+1)$, thus much smaller than the set of Arnoldi vectors of size mn (with $m << n$). Agullo et al. consider H to be stored (and solved) redundantly [1], which would then need no further protection. This also indicates that keeping a copy of the matrix H has a reasonable cost.

Listing 5: Preconditioned CG

```
1  ε_old ⇐ +∞                          
2  g ⇐ b − Ax                          
3  for t in 0..t_max                   
4      solve Mz = g        g = b − Ax  
5      ρ ⇐< z, g >         Mz = g      
6      β ⇐ ρ/ρ_old                     
7      d ⇐ βd + z          Mz = g      
8      q ⇐ Ad                          
9      α ⇐ ε/ < q, d >                 
10     x ⇐ x + αd                      
11     g ⇐ g − αq                      
12     ρ_old ⇐ ρ                       
```

Listing 6: Preconditioned BiCGStab

```
1  g, r, d ⇐ b − Ax                                
2  ρ ⇐< g, r >                                     
3  for t in 0..t_max                               
4      solve Mp = d          d double-buffered     
5      q ⇐ Ap                Mp = d               
6      α ⇐ ρ/ < q, r >       r = g + ωt           
7      r ⇐ g − αq                                 
8      solve Ms = r                               
9      t ⇐ As                Ms = r               
10     ω =< t, r > / < t, t > r = g − αq          
11     x ⇐ x + αp + ωs       p = A^{-1}q, Ms = r  
12     g ⇐ r − ωt            r = g − αq           
13     ρ_old ⇐ ρ                                  
14     ρ ⇐< g, r >           r = g + ωt           
15     β ⇐ ρ/ρ_old ∗ α/ω                          
16     d ⇐ g + β(d − ωq)     d double-buffered    
```

Listing 7: Preconditioned GMRES

```
1  for t ln 0..t_max                                  
2    g ⇐ b − Ax                                       
3    solve Mz = g                  g = b − Ax         
4    v^0 ⇐ z/||z||_2               Mz = g             
5    for l in 0..m − 1                                
6      u ⇐ Av^l                                       
7      solve Mw = u                                   
8      for k in 0..l                                  
9        h_{k,l} ⇐< w, v^k >                          
10       w ⇐ w − h_{k,l} v^k                          
11     h_{l+1,l} ⇐ ||w||_2                            
12     v^{l+1} ⇐ w/h_{l+1,l}                          
13   solve H = QR                                     
14   y ⇐ R^{-1} Q^T ||z||_2 e_1                       
15   x ⇐ x + Σ_{l=0}^{m−1} y_l v^l   x = A^{-1}(g − b)
```

3.2 Preconditioned algorithms

The described recovery techniques can be straightforwardly applied to the same algorithms with a preconditioner. To preserve the generality of our approach, and to avoid preconditioners specifics, we consider a generic preconditioning operation "solve $Mu = v$", M being the preconditioning matrix. To derive protected versions of the preconditioned algorithms we have to protect all the linear operations involving the preconditioned vectors. Protecting the execution of the preconditioner itself is beyond the scope of this paper, but a topic of complementary work, describing for example how to effectively protect multi-grid preconditioning [9].

To recover part of a preconditioned vector, there is no general way to avoid re-applying the preconditioner. Therefore, the prerequisite for the recovery to be cheap is the ability to perform a partial application of the preconditioner, that is, to apply the preconditioner to a small subset of v such that all lost data in u is recovered. If M is a block-diagonal matrix, solving $Mu = v$ only on the set of blocks that supersedes the lost data achieves this. If M is a fixed point method's matrix, the sparse set of elements in v that contribute to the lost portion of u is sufficient. If M denotes a multigrid method, we consider the nodes of the coarsest grid that participate to producing lost data, then we only need the inputs that contribute to these nodes for recovery. In any case, re-running the preconditioner completely is a viable, though slow, forward recovery for u. Finally, a corrupted v after a "solve $Mu = v$" operation is always recoverable without using the equation $Mu = v$. This is an important point since M is not always explicitly formed.

This can be made explicit by looking at the preconditioned versions of CG, BiCGStab, and GMRES, which are shown in Listings 5, 6 and 7. We can easily observe that in both CG and BiCGStab the preconditioned vectors z, p and s always exist at the same time as their non-preconditioned counterparts, g, d and r, because the latter are still used in the solver. Thus we can always recover the preconditioned ones as discussed in the previous paragraph. All the relations protecting operations that involve z or g in CG, and p, s, d or r in BiCGStab are detailed next to the code of the preconditioned versions. For preconditioned GMRES, shown in Listing 7, the main redundancy relation from its non-preconditioned counterpart linking all the v^k is still valid. The only addition is the need for g to be conserved for the possible recovery of x.

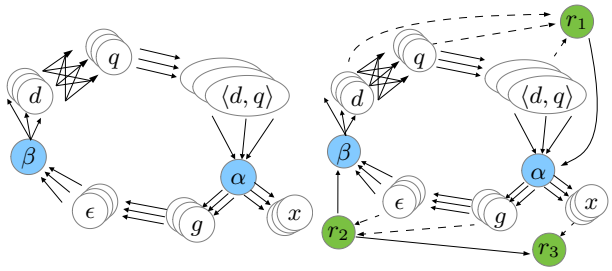
3.3 Implementing Recovery with Asynchrony

CG and BiCGStab are harder to protect, as they require both redundancy relations and double buffering approaches to be fully protected, while GMRES just requires redundancies. For this reason, as well as because CG is a very popular method for solving SPD matrix equations in the HPC context, we select it to test our approach. We implement two versions of CG, one without a preconditioner and a second one using a block-Jacobi preconditioner. Any conclusion obtained from our experiments with CG can be trivially extended to the other two since they constitute a similar and simpler use-cases respectively, and to their preconditioned versions as explained in Section 3.2.

3.3.1 *Conjugate Gradient's Parallel Decomposition*

The pseudo-code for CG is given in Listing 1, and its parallelization in tasks is done by strip-mining as shown in Figure 1(a), with each set of tasks being named after the value or vector it outputs. Dependencies between tasks are generated from annotations to the sequential code, and represented by arrows on this graph. Tasks are then scheduled asynchronously by the runtime according to this dataflow. Some dependencies that do not affect the ordering or scheduling of tasks are not drawn for the sake of clarity.

Sets of tasks depicted in white show an operation that is strip-mined into as many parallel tasks as available threads. Blue tasks (with converging arrows) are depending on all the previous tasks (because of a reduction operation) and represent a single task producing a scalar value. They are thus de facto synchronization points. The lattice-like arrows describe the fact that each following task depends on each previous task, as the block-row matrix-vector multiplication takes a whole vector as input for each single block as output.



(a) textbook version (b) with recovery tasks

Figure 1: Task decomposition of CG

3.3.2 Packing Recovery Tasks out of the Critical Path

We divide those relations in blocks as described in Section 2.2 and maintain an atomic bitmask (e.g. an `int`) per block of failure granularity, thus per memory page. Each data vector and task output is represented by a bit in this mask. Thus, if a task T works on a page p of a vector, it can check whether (one of) its inputs(s) was corrupted or skipped, and if so skip the computation while marking the bitmask with the bit representing T's output. This is necessary as to keep track of when errors happen, and works especially well with linear relations (which are the majority of considered relations). There is a memory overhead directly proportional to the size of the linear system n to store this information.

Skipping computations is critical for reductions, because a floating point accumulation can be irremediably corrupted by adding (or multiplying by) $+/-inf$ or nan. Through a thread-private `sig_atomic_t` variable, each task is made aware of interruptions, and only contributes a page-level accumulation to the task-level one when no errors were reported. For this page based division to be valid, the reduction needs to be associative, which is always guaranteed since it is already required for the strip-mining into tasks.

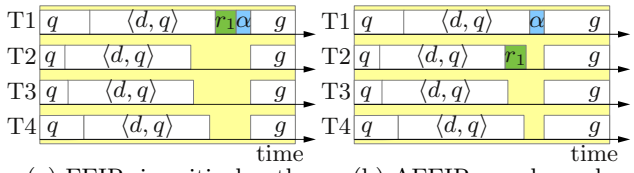
While errors are not corrected, the skipping of computations that depend on not-produced data propagates through the different tasks. When reaching a scalar task, skipping dependent computations would mean stop progressing completely. Thus we have to recover errors before the said scalar tasks. The graph in Figure 1(b) shows the modified cycle with the green tasks where recoveries take place: replacing lost data and recomputing skipped computations.

Each recovery task recovers the inputs and outputs of normal tasks which point to it with dashed lines. Recovery tasks are always added to the execution flow of the program and check the global variables for signalled errors. If none occurred, the tasks do nothing.

Because dashed lines represent communication through atomic global bitmasks rather than dependencies, recovery tasks can execute concurrently to CG tasks, which do not touch the memory pages to be recovered, either skipping them or working on unrelated data. This allows to overlap computations and recoveries, thereby reducing overheads; however errors discovered between recovery tasks and the following scalar task are not recoverable. Thus we execute r_1 and r_2 asynchronously as late as the scheduler allows us to schedule them, which means concurrently with $< d, q >$ and ϵ respectively, and with a lower priority as to start all reduction tasks first, see Figure 2(b). This technique is the Asynchronous Forward Exact Interpolation Recovery (AFEIR).

A more conservative approach consists in allowing the recovery tasks to execute in the critical path, that is, waiting for all computations that do not need lost data to finish and only then run the recovery, as illustrated in Figure 2(a). Note that this latter option has a slightly better coverage of faults, since all the tasks (thus potential error discoveries) have finished executing when the recovery starts, as we will see with high error injection rates in Section 5.4. We call this technique Forward Exact Interpolation Recovery (FEIR).

The parallelization strategy for the preconditioned CG is exactly the same as for the non-preconditioned CG. The only added recovery technique is the partial "solve $Mu = v$", as explained in Section 3.2, which is easy since block-Jacobi is a blocked preconditioner.



(a) FEIR, in critical path (b) AFEIR, overlapped

Figure 2: Traces illustrating scheduling of recovery tasks

3.4 Recovery on Distributed Memory Systems

The recovery methods described so far apply to shared memory models, in a single node or in a partitioned global address space for example. We list here the few modifications needed to extend our resilient methods to distributed memory programming models like MPI, that we will use to evaluate how the recovery techniques impact the scaling of the application in highly parallel scenarios.

More specifically, we use a hybrid implementation where the node level parallelism is levered using MPI and the intra-socket parallelism is expressed in terms of the asynchronous task-based data-flow programming model, OmpSs. Our CG solver only requires the following additions:

- Global MPI reductions after the local reductions
- A task to exchange local parts of the vector p with neighbouring nodes depending on it, at every iteration.

This new exchange task takes place instead of the lattice-like dependencies between tasks d and q, while the MPI reductions occur during the α and β tasks (see Figure 1).

For our FEIR and AFEIR methods, we instantiate a second r_1 recovery task to be executed before our new exchange task, to avoid sending potentially failed and non-corrected data. Finally, MPI communications added inside the task recovering the x vector, r_3, request and perform exchanges of parts of x when needed for recovery, since this vector is not exchanged at every iteration.

4 Other Recovery Approaches

4.1 Trivial Forward Recovery

The trivial forward recovery consists in simply keeping the program running, by allocating new (blank) memory for corrupt or lost data. No other actions are taken. While an error in a part of the data that is not reused later would be masked, we lose all guarantees on convergence.

4.2 Rollback Recovery

Our checkpointing method is only applied to dynamic data, to be fair in our comparison of methods, and to be consistent with the assumptions on which the DUE recovery relies. Each Processing Element (PE) periodically writes to its local disk the values of the iterate and search direction vectors it has at that given moment (x and p), which is the minimum to allow rolling back; frequency of checkpointing is expressed in terms of iterations of the solver.

There is no need to use a parallel file system, since we assume that the program will not crash (as we catch the errors). At rollback, each PE will restore the vector portions that were saved to its local disk at the last checkpoint.

Checkpoints and rollbacks are global, that is, they involve all the PEs of a parallel run. In a distributed memory scenario, we perform a global MPI reduction once per iteration to decide whether a rollback is needed. This global communication is executed simultaneously with the α global MPI reduction to avoid further synchronization overheads.

4.3 Lossy Restart

Langou et al. [27] present an interesting forward recovery method for the fail-stop model of an MPI process, the Lossy Approach. This recovery also applies to all Krylov-subspace methods. To compensate for the loss of a part of the iterate x, a step of the block-Jacobi is used, which relies only on constant data and the remaining parts of x, and provides an immediate reduction of the norm of the residual.

This operation is similar to our recovery for the iterate, while discarding the residual in the block relation. After such an interpolation, a restart is necessary since the residual g is outdated and not easily deducible.

We adapt this Lossy Approach into a recovery for our error model, that we name the Lossy Restart:

1. If part of the iterate is lost we use the interpolation from the Lossy Approach. With i the failed block:
$$A_{ii}x_i = b_i - \sum_{j \neq i} A_{ij}x_j$$
2. We restart the method with either the intact or the newly interpolated iterate as initial guess.

Before comparing these methods, let us consider theoretical results presented on this interpolation, by noting x^* the solution of the system $b = Ax^*$, x the iterate, x^I the newly interpolated iterate, $e = x^* - x$ and $e^I = x^* - x^I$ the respective errors, and $g = b - Ax$ and $g^I = b - Ax^I$ the respective residuals. Langou et al. show the following:

THEOREM 1. *The interpolation is contracting for a constant* $c_i = (1 + ||A_{ii}^{-1}|| \sum_{j \neq i} ||A_{ij}||)^{1/2}$, *thus* $||e^I|| \leq c_i||e||$.

This result has been improved for A SPD [1]:

THEOREM 2. *With A symmetric positive definite, the interpolation diminishes the A-norm of the error:* $||e^I||_A \leq ||e||_A$.

Both theorems grant the block-Jacobi's fixed point property: if $x = x^*$, then $x^I = x^*$, since $e = 0$.

From here on, we will restrict ourselves to SPD matrices and show that the block-Jacobi step does not just give better replacement data, but the best possible – in the short run.

THEOREM 3. *For A SPD, the interpolation* minimizes *the A-norm of the error* $||e^I||_A$ *over all possible values for* x_i^I

The proof of our theorem relies on the transformation of the error implied by the linear interpolation, $p_i^I : e \to e^I$ being a linear projection, orthogonal for the norm $|| \cdot ||_A$

PROOF. By construction, the residual at x^I for the block i is $g_i^I = b_i - \sum_{j=0}^{n-1} A_{ij}x_j^I = 0$. Let us also notice that $g = b - Ax = A(x^* - x) = Ae$ and similarly $g^I = Ae^I$.

Now let us show that the kernel and image of p_i^I are orthogonal for A:

$$\begin{aligned} \forall e \in \Re, < p_i^I(e), e - p_i^I(e) >_A &=< e^I, e - e^I >_A \\ &=< Ae^I, e - e^I > \\ &= \sum_{j=0}^{n-1} < g_j^I, e_j - e_j^I > \end{aligned}$$

This is always zero, since for $j = i$, $g_j^I = 0$ and for $j \neq i$, $x_j^I = x_j$ thus $e_j = e_j^I$. It then comes clearly that:

$$||e||_A = ||p_i^I(e)||_A + ||(Id - p_i^I)(e)||_A$$

where $p_i^I(e)$ depends solely on the e_j (thus x_j) with $j \neq i$. Hence the minimum of this norm for all possible x_i, or e_i, is reached in $p_i^I(e)$. □

Table 2: Resilience methods' overheads, no errors

method	Lossy	Trivial	AFEIR	FEIR	ckpt 1K	ckpt 200
overhead	0.00%	0.00%	0.23%	2.73%	17.62%	46.20%

Table 3: Increase of time spent per state for FEIR methods

	imbalance	runtime	useful
AFEIR	4.30%	8.11%	1.90%
FEIR	25.06%	7.84%	2.78%

We can also deduce this from Theorem 2 and the fact that the unknown part of x is in the kernel of p_i^I. $||e^I||_A \leq ||e||_A$ then holds for any x_i, hence the *min* relation of our theorem.

Restarting the solver, with a good or unmodified initial guess, still harms the superlinear convergence of CG, which relies on the fact that the sequence of iterates x minimizes at each iteration the norm $||x^* - x||_A$ on a sequence of increasing subspaces. However, this disturbance may be beneficial to methods who have a tendency to stagnate (such as GMRES).

All recoveries based on restarting are identical as long as the iterate is untouched, and trade in convergence properties for simplicity of recovery in the same way. It is to be expected that such methods would behave very similarly to the Lossy Restart, though always worse in the short run, hence it is the only restart method against which to compare.

5 Performance Evaluation

5.1 Experimental Set-up

We measure solving 9 matrices selected from the University of Florida sparse matrix collection [13]. They are well-conditioned matrices for CG selected among the biggest of each family of SPD matrices. We ran experiments on Intel® Xeon® E5-2670, with one thread on each of its 8 cores.

Our evaluation is done on two versions of CG, a non-preconditioned version to show the hardest case possible, and one using a block-Jacobi preconditioner. Due to the wide variety of preconditioners available for CG, it is impossible for us to evaluate every single one. Our remarks in Section 3.2 comment the desirable properties of preconditioners for an efficient recovery. The block-Jacobi is simple to implement, and trivially applicable to a subset of a vector. We select it also because, if its block size coincides with the memory page size, the factorization of diagonal blocks for the recovery of single errors is already computed. Thus we will use diagonal blocks of 512 by 512 elements.

We compare the following methods: our Forward Exact Interpolation Recovery (FEIR) without asynchrony (recovery tasks in critical path), our Asynchronous Recovery (AFEIR), the Lossy Restart, checkpointing-rollback to local disk, and Trivial Forward Recovery. The optimal checkpointing rate is used whenever errors are injected, and no fallback is used for FEIR or AFEIR: simultaneous errors on related data are simply ignored (see Section 2.4).

5.2 Techniques Overheads

From here on, the "ideal" CG will refer to our version of CG with no resilience mechanisms nor error injections.

We present in Table 2 the harmonic means of overheads for all methods in absence of faults, compared to the ideal CG. The Lossy Restart and Trivial techniques have no overhead when no errors are injected, since catching the error, replacing memory pages and ordering a restart is done in a signal handler which is never called. To give a sense of checkpointing cost we arbitrarily consider checkpointing periods of 200 and 1000 CG iterations. The corresponding

overhead raises from 17.62% to 46.20% as the checkpointing frequency increases, which constitutes a significant cost.

The overheads associated to the AFEIR and FEIR techniques are much smaller since they are associated to activities like task creation or scheduling, that are much cheaper than writing data to disk. The asynchronous nature of the AFEIR technique allows to compensate much of the overhead incurred by the FEIR technique. We can see in Table 3 a detailed breakdown of what is involved in the overheads of the FEIR and AFEIR methods, expressed as the increase of the proportion of time spent in each state while the solver is running: either idle, thus suffering load imbalance, or performing runtime work, such as creating and scheduling tasks, or finally executing tasks, thus doing computations for the solver. Executing the recovery tasks in the critical path obviously increases the load imbalance.

Most of the runtime overhead of FEIR and AFEIR techniques could be removed if application-level resilience were supported by the runtime, instantiating recovery tasks only when DUE are signalled.

5.3 Error Injection

We consider the most common DUE to test the considered recovery techniques: the corruption of a memory page. However, this is generalizable to more types of errors, since a DUE very often ends up being a data corruption. DUE can also bring changes in the control flow of the programs, which typically ends up with a data corruption or an execution failure. Our model covers the first scenario, while the second lies beyond the scope of this paper.

Errors are injected from a separate thread at times defined by an exponential distribution parametrized by the Mean Time Between Errors (MTBE). To account for the wide range of convergence times across different matrices (from 1 to 100 seconds), we normalize the MTBE to the ideal convergence time. Affected memory pages are selected at random with uniform distribution.

To simulate errors we use the `mprotect` system call available in Linux kernels to change the authorizations of the targeted memory page. This is more practical than triggering a real hardware poisoning of a memory page, and behaves identically: the program receives a signal at the time of access to the memory page. We recover in the same way as we would from a real error: in a signal handler, we request a new memory page at the same virtual address through means of the `mmap` system command. All the recoveries operate exactly in the same way as they would if a real DUE took place. For the solver, there is no difference between real hardware DUE and our error injection mechanism.

Errors are injected in the memory pages of the Krylov vectors, which we cover with our resilience techniques, and not in the pages that contain constant data, program instructions, scalar or control values of the algorithm, or our bitmasks used for resilience. The amount of non-constant data that is not covered by our error injections is very small compared to the targeted memory space, and relatively constant across the different resilience techniques.

5.4 Convergence and Performance

Figure 3 illustrates the convergence of CG for a sample scenario consisting of a single error injection. The x-axis represents the time and the y-axis shows the execution progress in terms of the logarithm of the residual norm defined as $||Ax - b||/||b||$, updated at each iteration. The ideal CG is

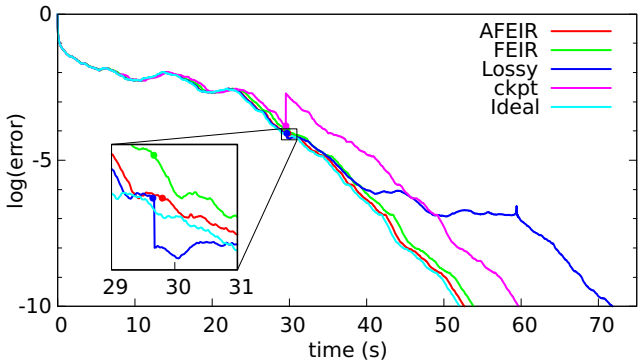


Figure 3: CG convergence for different resilience methods with matrix thermal2 and same single error injection in x

represented by the cyan line; all the other experiments have a single error injected 30 seconds after the beginning of the execution at a certain memory page that contains a portion of the iterate x. Before the error is injected, each resilience method pays its typical overhead in absence of faults. The purple line corresponding to the checkpointing mechanism is the one with more overhead, which is consistent with the analysis presented in Section 5.2, as we use a checkpointing frequency of 1000 iterations. At the time of the error, it already incurs a 9.12% slowdown. Once the error is injected, the checkpointing mechanism rolls back a certain number of iterations and resumes progress from there. The Lossy Restart, represented by the blue line, has an immediate reduction in the error thanks to its block-Jacobi step interpolation, but converges slower afterwards because restarting harms CG's superlinear convergence. The FEIR and AFEIR methods recover the lost data by using an exact interpolation and keep progressing. The overhead paid by the AFEIR technique is significantly smaller than the one paid by FEIR, since asynchrony allows most of the recovery work to be overlapped with other computations.

Figure 4 shows an exhaustive evaluation of the performance slowdown associated to the 5 resilience mechanisms listed in Section 5.1: Trivial, checkpointing-rollback, Lossy Restart, FEIR and AFEIR. The checkpointing rate is computed for each experiment to minimize execution time, taking into account the time to write and read checkpoints, the MTBE, and no downtime [5]. We consider the same 9 input matrices as for the overhead measures, and 6 error injection scenarios per matrix and method, which means that we provide an evaluation of 270 different experiments. Each experiment has been run over 50 times and Figure 4 reports their harmonic mean, and standard deviation as error bars. In each repetition, the errors have been injected randomly at different times and memory pages. On the x-axis of Figure 4 we display the name of the considered matrices and, for each matrix, the error injection frequency normalized to the ideal CG's convergence time τ for that matrix. A value n means an error frequency of $\frac{n}{\tau}$, thus an MTBE of $\frac{\tau}{n}$. In other words, n is the expected number of errors injected during the ideal convergence time τ. The y-axis is displayed in logarithmic scale and shows the measured performance slowdown in percentage for each experiment, with respect to the ideal CG. A slowdown close to 0 means the resilient CG converges at a speed close to that of the ideal one, whereas a bigger slowdown means its convergence is slower. The convergence threshold is 10^{-10}.

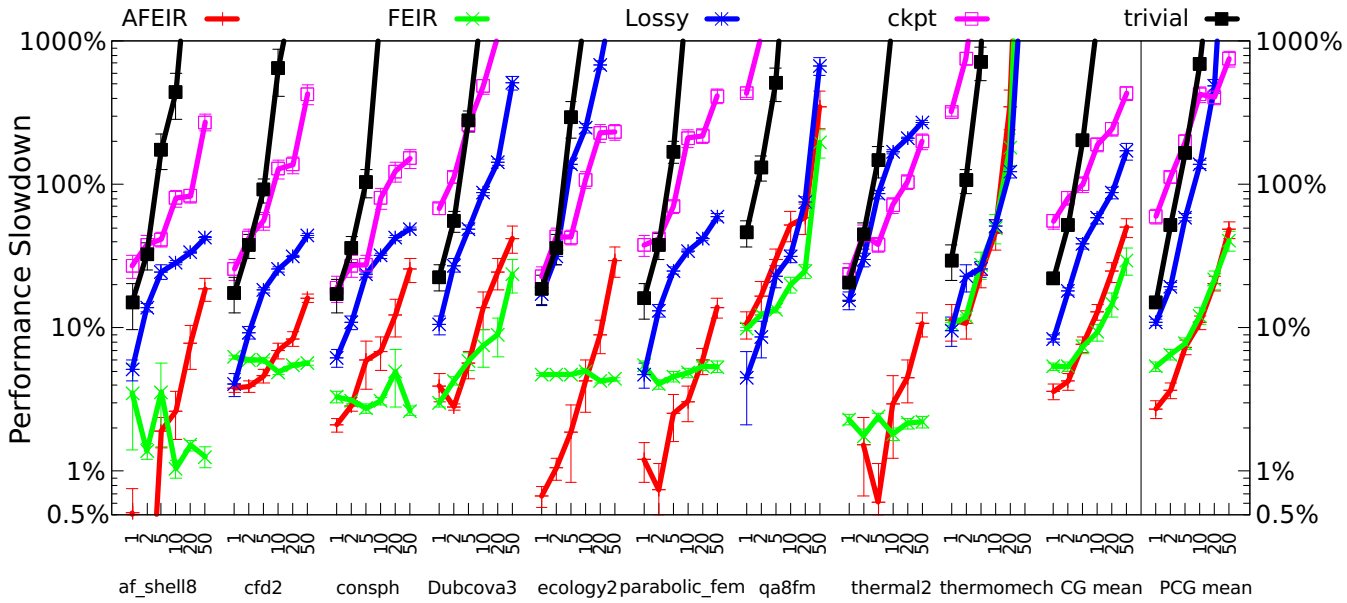


Figure 4: Comparison of the execution time for resilience methods and matrices, varying error injection rates

We have run the exact same 270 experiments with the block-Jacobi Preconditioned CG (PCG), however due to space limitations, we only report the mean of those results, displayed at the right hand side of Figure 4.

The trivial method reacts badly against few errors and its convergence times diverge extremely fast, with overheads over 200% with a normalized error frequency of 5 only. For PCG, the overheads of the trivial recovery reach 50% with a normalized frequency of 2 and become larger than 700% for frequencies of 10 or more. The checkpointing scheme reacts better than the trivial method, with substantial overheads that tend to increase slower, ranging on average from 55% to 433% for CG and from 60% to 752% for PCG. These convergence times are close to the expected values from the checkpointing frequency computation [5]. The Lossy Restart behaves better on average than the trivial method and the checkpointing schemes. Regarding CG, it has an overhead of 8.4% with one expected error per ideal convergence time, reaching up to 87% and 170% against 20 and 50 times higher frequencies respectively, whereas for PCG these overheads are 12.80% and 500% for normalized frequencies of 1 and 20. This better behaviour of the lossy mechanisms with respect to checkpointing and trivial techniques is already reported in the literature [1], and the fact that our experimental framework has reproduced known results demonstrates its accuracy and reliability. This is matrix specific however, as non-ABFT methods are clearly outperformed for af_shell8 or cfd2 but have overheads similar to Lossy for thermal2.

The most important fact of our evaluation is that methods FEIR and AFEIR behave much better than the current state-of-the-art resilience techniques for iterative solvers. When applied to CG, FEIR has an overhead of 5.37% and 29.68% under normalized frequencies of 1 and 50, whereas AFEIR has overheads of 3.59% and 50.47% respectively for the same error rates. On PCG, FEIR and AFEIR have an overhead of 5.36% and 2.72% with the smallest frequency, and reach 40.55% and 48.55% with the largest.

While FEIR has a roughly constant overhead on most matrices, the impact of recoveries in the critical path can be seen where execution times are the shortest, such as for Dubcova3 and especially qa8fm and thermomech. As recoveries run on a per iteration basis, the error rates per iteration determine the chances of encountering errors on related data and during recovery tasks. Under injection frequencies of 20 and 50, both qa8fm and thermomech experience over 0.2 and 0.6 errors per iteration, which significantly pulls up the mean overheads. Such extreme cases could be dealt with by using more recoveries per iteration, or a fallback method for unrecoverable errors. These matrices also show that Lossy Restart is most profitable on fast converging problems.

AFEIR is slower than FEIR for high error injection rates, because errors happening between the end of a recovery task and its following scalar task are unrecoverable. That is the time between the end of $r1$ or $r2$ and the beginning of α and β respectively, as illustrated by Figure 2(b). With very high error injection rates, the probability of an error happening during these time windows may cause the contribution of a memory page to <d,q> or ϵ (see Figure 1(b)) to be ignored. Depending on the matrix and the actual data lost, this might have a significant impact, as matrix ecology2's behaviour shows. The FEIR method is not at risk of discovering an error after a recovery task ran, because these tasks start after all computations are done. However, both methods are still vulnerable during the recovery's execution. There is thus a trade-off between the low overheads of AFEIR at frequencies of 10 and less, and a more conservative approach, FEIR, which trades in some convergence speed for safer recoveries and is thus useful at higher error rates. The same trade-off applies to the PCG results for low error injection rates. The precomputed factorization of diagonal blocks reduces recovery time, thus a block-Jacobi preconditioner weakens this trade-off for high error injection rates. It is to be expected that when using a preconditioner whose partial application is computationally hard (see Section 3.2), the average recovery time will increase and this trade-off will become stronger.

5.5 Scaling Results

In this section, an evaluation of the scalability of our recovery techniques is performed considering a hybrid MPI + OmpSs implementation. Since the previously considered

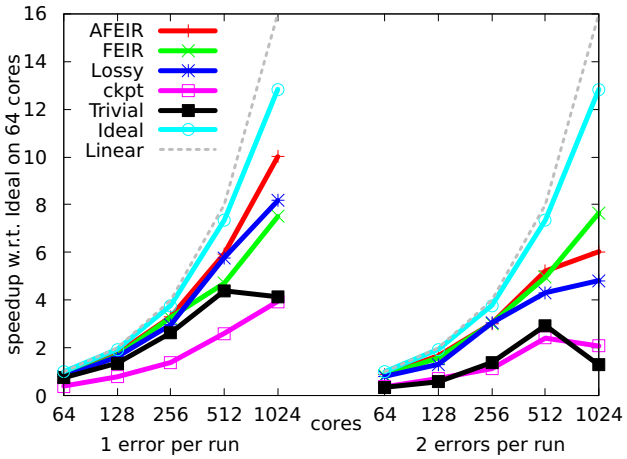


Figure 5: Speedup of the MPI+OmpSs resilient CGs

matrices are not well suited for large scale experiments, we solve Poisson's equation in 3D using a 27 point stencil discretization, which is also used in the HPCG benchmark [19], with a system size of 512^3 unknowns. Experiments are run in the MareNostrum supercomputer, whose nodes contain two Intel Xeon CPU E5-2670 sockets. Each MPI rank is mapped to one 8-core socket, running 1 OmpSs thread per core. We consider runs on 8, 16, 32, 64 and 128 sockets (64, 128, 256, 512 and 1024 cores), since we need at least 8 processors to fit the matrix in memory.

We present in Figure 5 a complete evaluation in terms of speedup, injecting one and two errors per run. The speedups are computed taking the execution time of an ideal CG on the smallest possible core count, 64, as a reference. We display data concerning the FEIR, AFEIR, Lossy Restart, checkpointing and trivial techniques, and include the ideal CG's and linear speedups for reference. Our MPI+OmpSs CG implementation achieves a parallel efficiency of 80.17% on 1024 cores in a faultless run, which highlights its quality in terms of parallel performance.

AFEIR and FEIR techniques clearly overcome the trivial and the checkpointing techniques, achieving speedups of 10.01 and 7.50 respectively when 1 error is injected and 6.03 and 7.65 against two errors on 1024 cores. The Lossy Restart achieves speedups of 8.17 and 4.82 respectively on 1024 cores. It is worth noting that only a few tens of iterations are required to achieve convergence for the 27-point stencil matrix, which causes any overhead to be important compared to the ideal execution time, but also makes this matrix the ideal workload for a restart method. Even in this case, restarting is as costly if not more than our FEIR and AFEIR methods' overheads. Regarding checkpointing, writing vectors to disk only already causes the checkpointing to perform significantly worse than our baseline, and when injecting errors its speedups stay below a third of the ideal CG, close to that of the Trivial method.

6 Related Work

6.1 Localizing and Detecting Errors

In a multi-node and distributed memory execution, a fail-stop error model for each process yields a localized loss of data error model globally. This naturally leads to the proposal for a Fault-Tolerant Message Passing Interface (FT-MPI), that hands control back due to a node having stopped, in order to recover data from this part of the program [16].

Chen more recently proposed to use two application specific invariant relations to detect Silent Data Corruption (SDC) [11] in CG and its derived method BiCGStab, which are the (bi-)orthogonality of search directions, and the relation between gradient and iterate. The latter consists in checking $||b - Ax - g|| = 0$ (with our notations), which reflects the use of inherent redundancy between vectors of CG, as levered in this paper.

6.2 Checkpointless Algebraic Recoveries

Many ABFT methods add checksum values such that transformations applied by a program to its data are similarly applied to these checksums. This has been accomplished for matrix-vector multiplications by adding a checksum row in a matrix [21], but also for other operations such as QR and LU factorizations [12, 18]. This approach has the advantage of adding little memory space overhead at the price of computational overhead. However, checks on finite precision numbers (as opposed to bitwise checks) are sensible to round-off errors, and they do not cover reduction operations.

While exploiting MPI message logging as an implicit checkpoint, Chen et al. [10] proposed an algebraic recovery method for Krylov solvers with matrix A and right-hand side b, whose iterate x is seldom passed in MPI messages. Once all the other variables are recovered through the implicit checkpoint, the relation between the residual g and x, thus $g = b - Ax$, is used with the lost part of the iterate as unknown. This allows to recover the lost data without checkpointing x, using inherent redundancy of the solver instead.

Langou et al. introduced the Lossy Approach with the block-Jacobi step interpolation [27]. This restart method, designed for an MPI fail-stop error model, is already extensively discussed in Section 4.3. Agullo et al. extended this work by introducing least-squares methods for the interpolation, and further studying strategies to minimize computations and communications in recoveries in the case of multiple simultaneous errors [1]. These trade-offs can naturally be applied to the interpolations we use, if needed.

6.3 Selective Reliability

The Fault Tolerant GMRES [20] is a method consisting of GMRES iterations, run safely, enclosing a preconditioner which may run unreliably and return inexact values. Our protection of GMRES provides the complement to this selective approach tolerating errors in the preconditioner, by protecting the outer iterations – and especially the vector basis which is, according to Hoemmen et al., the most important to guarantee convergence. This work also suggests that we might not need to recover preconditioned vectors exactly, instead replacing lost data with an approximation.

7 Conclusions

This paper demonstrates that hardware DUE reporting can be exploited jointly with redundancy relations to protect iterative solvers paying a very low cost. Our two proposed methods, FEIR and AFEIR, overcome the state-of-the-art techniques in terms of overheads. They are moreover based on very simple relations that do not require deep algorithmic understanding, whereas an algorithmic technique like the Lossy Approach [27] is harder to derive. These straightforward low-overhead recoveries open the door to wide-spread use of algorithmic-based techniques to protect iterative methods when DUE detection is available.

Second, the paper demonstrates that by overlapping recovery with algorithmic computation, overheads can be drastically reduced. Under high error rates, of roughly more than 0.1 errors per second, the overlapping stops paying off since the chances of getting errors on non-protected computations increase, even though this trade-off is largely matrix specific. The FEIR technique provides then better performance. In any case, task-based data-flow programming models have interesting properties for resilience, not only because of inherently splitting programs into tasks, but also because overlapping computations and recoveries is done without explicit programmer intervention. Runtime support for application-level resilience could reduce the overheads by injecting recovery tasks only when errors are encountered – this would also increase AFEIR's coverage by potentially executing the recovery tasks later, and still asynchronously.

Our resilience method opens the door to interesting trade-offs when SDC comes into play. Since we cover with very low overhead nearly all memory page failures, an ECC that focuses more on detecting than correcting errors would reduce SDC [25], while delegating some correction to the application level. This work will hopefully encourage future architectural, OS and runtime features to expose errors at the application level whenever lower level recoveries fail, allowing resilience aware applications to resist significantly more to adverse conditions than applications oblivious to resilience.

Acknowledgements

This work has been partially supported by the European Research Council under the European Union's 7th FP, ERC Advanced Grant 321253, and by the Spanish Ministry of Science and Innovation under grant TIN2012-34557. L. Jaulmes has been partially supported by the Spanish Ministry of Education, Culture and Sports under grant FPU2013/06982. M. Moreto has been partially supported by the Spanish Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship JCI-2012-15047. M. Casas has been partially supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Co-fund programme of the Marie Curie Actions of the European Union's 7th FP (contract 2013 BP_B 00243).

References

[1] E. Agullo, L. Giraud, A. Guermouche, J. Roman, and M. Zounon. Towards resilient parallel linear Krylov solvers: recover-restart strategies. Research Report RR-8324, INRIA, 2013.

[2] *AMD64 Architecture Programmer's Manual*, volume 2: System Programming, chapter 9. AMD, 2011.

[3] M. Berry, R. Barrett, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Society for Industrial Mathematics, 1994.

[4] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Extending the scope of the checkpoint-on-failure protocol for forward recovery in standard mpi. *Concurrency Computat.: Pract. Exper.*, 25(17):2381–2393, 2013.

[5] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *SC*, article no. 1, 2011.

[6] P. Bridges, M. Hoemmen, K. Ferreira, M. Heroux, P. Soltero, and R. Brightwell. Cooperative Application/OS DRAM Fault Recovery. In *Euro-Par*, pages 241–250, 2012.

[7] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *Int. J. High Perform. Comput.*, 23(4):374–388, 2009.

[8] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Towards Exascale Resilience: 2014 update. *Supercomput. front. innov.*, 1(1):5–28, 2014.

[9] M. Casas, B. de Supinski, G. Bronevetsky, and M. Schulz. Fault resilience of the algebraic multi-grid solver. In *ICS*, pages 91–100, 2012.

[10] Z. Chen. Algorithm-based recovery for iterative methods without checkpointing. In *HPDC*, pages 73–84, 2011.

[11] Z. Chen. Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *PPoPP*, pages 167–176, 2013.

[12] T. Davies and Z. Chen. Correcting soft errors online in LU factorization. In *HPDC*, pages 167–178, 2013.

[13] T. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1–25, 2011.

[14] V. Degalahal, R. Ramanarayanan, N. Vijaykrishnan, Y. Xie, and M. Irwin. The effect of threshold voltages on the soft error rate. In *ISQED*, pages 503–508, 2004.

[15] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parall. Proc. Lett*, 21(2):173–193, 2011.

[16] G. Fagg, A. Bukovsky, and J. Dongarra. Fault tolerant MPI for the HARNESS meta-computing system. In *ICCS*, pages 355–366. Springer, 2001.

[17] K. Ferreira, J. Stearley, J. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *SC*, article no. 44, 2011.

[18] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. Stanley. An Overview of the Trilinos Project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[19] M. Heroux, J. Dongarra, and P. Luszczek. HPCG technical specification. Technical Report SAND2013-8752, Sandia National Laboratory, 2013.

[20] M. Hoemmen and M. Heroux. Fault-tolerant iterative methods via selective reliability. Technical Report SAND2011-3915 C, Sandia National Laboratory, 2011.

[21] K.-H. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 100(6):518–528, 1984.

[22] Intel® Xeon® Processor E7 Family: Supporting next generation RAS Servers. White paper, Intel Corporation, 2011.

[23] *Intel® 64 and IA-32 Architectures Software Developer's Manual* rev. 51, volume 3B: System Programming Guide, Part 2, chapter 15–16. Intel Corporation, 2014.

[24] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar. Near-threshold voltage (NTV) design: Opportunities and challenges. In *DAC*, pages 1149–1154, 2012.

[25] J. Kim, M. Sullivan, and M. Erez. Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory. In *HPCA*, pages 101–112, 2015.

[26] A. Kleen. mcelog: memory error handling in user space. In *Linux Kongress*, pages 159–166, 2010.

[27] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. *SIAM J. Scientific Computing*, 30(1):102–116, 2007.

[28] X. Li, M. Huang, K. Shen, and L. Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *USENIX ATC*, pages 6–6, 2010.

[29] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC*, 2010.

[30] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *MICRO*, pages 29–42, 2003.

[31] I. Reed. A class of multiple-error-correcting codes and the decoding scheme. *Trans. IRE Prof. Group Inf. Theory*, 4(4):38–49, 1954.

[32] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS*, pages 193–204, 2009.

[33] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, 1994.

[34] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA*, pages 123–134, 2002.

[35] O. Subasi, J. Arias Moreno, J. Labarta, O. Unsal, and A. Cristal. NanoCheckpoints: A Task-based Asynchronous Dataflow Framework for Efficient and Scalable Checkpoint/Restart. In *PDP*, pages 99–102, 2015.

[36] D. Tang, P. Carruthers, Z. Totari, and M. Shapiro. Assessment of the effect of memory page retirement on system RAS against hardware faults. In *DSN*, pages 365–370, 2006.

[37] M. Wong, M. Klemm, A. Duran, T. Mattson, G. Haab, B. de Supinski, and A. Churbanov. Towards an error model for openMP. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, pages 70–82. Springer, 2010.

[38] X. Yang, Z. Wang, J. Xue, and Y. Zhou. The reliability wall for exascale supercomputing. *IEEE Trans. Comput.*, 61(6):767–779, 2012.

[39] D. H. Yoon and M. Erez. Virtualized and Flexible ECC for Main Memory. In *ASPLOS*, pages 397–408, 2010.