A Numerical Soft Fault Model for Iterative Linear Solvers

James Elliott[1,2] jjellio3@ncsu.edu
Mark Hoemmen[2] mhoemme@sandia.gov
Frank Mueller[1] mueller@cs.ncsu.edu

[1] Department of Computer Science, North Carolina State University, Raleigh, NC
[2] Center for Computing Research, Sandia National Laboratories, Albuquerque, NM

ABSTRACT

We present a fault model designed to bring out the "worst" in iterative solvers based on mathematical properties. Our model introduces substantially higher overhead, but smaller variance, than a fault model based on random bit flips. We also relate the statistics from our experiments back to the solvers' configuration, and briefly address the computational effort that each model requires. Our approach requires significantly fewer resources, while punishing our solvers with undetectable errors that require notable overhead for recovery. This work also illustrates the robustness of our resilient algorithms: Not only do we make forward progress in the presence of pathological faults, we always obtain the correct answer.

Categories and Subject Descriptors

G.4 [**MATHEMATICAL SOFTWARE**]: Reliability and robustness; G.1.3 [**NUMERICAL ANALYSIS**]: Numerical Linear Algebra—*Linear systems (direct and iterative methods)*

Keywords

Algorithm-Based Fault Tolerance; Sparse Iterative Methods; Pessimistic Fault Modeling

1. INTRODUCTION

Recent studies indicate that large parallel computers will continue to become less reliable as energy constraints tighten, component counts increase, and manufacturing sizes decrease [8, 5]. This unreliability may manifest in two different ways: either as "hard" faults, which cause the loss of one or more parallel processes, or as "soft" faults, which cause incorrect arithmetic or storage, but do not kill the running application.

This paper focuses on soft faults. Specifically, we consider those that corrupt data or computations, without the hardware or system detecting them and notifying the application that a fault occurred. We call this type of soft fault Silent Data Corruption (SDC). SDC is much less frequent than process failures, but much more threatening, since the application may silently return an incorrect answer. In physical simulations, the wrong answer could have costly and even life-threatening consequences. Users' trust in the results of numerical simulations can lead to disaster if those results are wrong, as for example in the 1991 collapse of the Sleipner A oil platform [11]. Unlike with hard faults, applications currently have few recovery strategies. Hardware *detection* without correction may cost nearly as much as full hardware correction. Hardware vendors can harden chips against soft faults, but doing so will increase chip complexity and likely either increase energy usage or decrease performance. An open field of research and the focus of this paper is designing algorithms that can tolerate SDC.

HPDC'15, June 15–20, 2015, Portland, Oregon, USA.

ACM 978-1-4503-3550-8/15/06 ...$15.00.
http://dx.doi.org/10.1145/2749246.2749254.

2. PRECONDITIONED LINEAR SOLVERS

Our fault model assesses iterative, possibly preconditioned, linear solvers under faults that are not detectable in standard implementations, and that can remain undetectable using low overhead detectors. The model introduces a *pessimistic* fault. This accounts for our lack of knowledge of exactly which physical events can lead to the worst case for a particular problem and solver combination. We argue that if a numerical approach can tolerate these types of perturbations, then it should be able to tolerate transient arithmetic errors. This minimizes fruitless speculation about how faults manifest in real hardware, and instead asks whether an algorithm can handle challenging numerical faults. The latter presents a fault model that we show produces a much worse case than random bit flips. If it is true that future hardware will allow some transient soft errors, we should assess fault tolerance in algorithms based on a worst-case scenario, rather than the extremely biased case of random bit flips.

2.1 Soft Faults and Iterative Methods

Given a direct solver, if a soft fault corrupts arithmetic, the method will reach a (possibly unacceptable) solution in a bounded, known number of steps. Iterative methods behave differently. They may 1) "converge through" the error, taking no more iterations than in the error-free case; 2) converge but take more iterations; 3) *stagnate* — reach the maximum iteration count without improving the initial approximation; or 4) become divergent — oscillate wildly or have rapid error growth such that the solver "explodes" toward infinity. In the latter two cases, the solver fails to produce an acceptable solution. Stagnation relates to the *maximal attainable ac-*

curacy, which bounds below the accuracy an iterative solver can reach in finite-precision arithmetic. If a soft error introduces error sufficient to damage the maximal attainable accuracy, then the solver may stagnate.

Pessimistic faults have mathematical interpretations. For example, they may introduce a fictitious, abnormally large eigenvalue to the matrix A. Iterative solvers approximate the solution as a linear combination of basis vectors that are weighted by the largest eigenvalues in the system. The fault will thus make the solver converge to a bogus solution dominated by the fictitious eigenvalue. Also, iterative solvers are often used for solving discretized versions of elliptic partial differential equations (PDEs). Their solutions must satisfy the *maximum principle*: their maximum must be found on the boundary. One may view a soft fault as a (transient) violation of this principle. Alternatively, a "bad" soft error may make the problem appear to have a nonmathematical discontinuity.

Our fault model evolves from these mathematical interpretations. We model faults as a specific MPI process returning a bad vector from its preconditioner application. We generate faults in two ways: a fault may 1) scale its contribution to the global vector, or 2) permute its local portion of the global vector. Permutations preserve the vector's norm, while making its contents incorrect. This models discontinuity. Scaling increases or decreases the norm of the vector predictably. This, or directly corrupting inner product or norm results, perturbs the eigenvalue approximations. Corrupting the basis vectors also makes the algorithm search for a solution in the wrong direction. Our numerical fault model suffices to cause stagnation or divergence in non-restarted solvers.

2.2 Selective Reliability

Our fault-tolerance strategy rests on relating numerical methods that naturally correct errors to system-level fault tolerance. In particular, we assume a *selective reliability* or "sandboxing" programming model [6] that lets algorithm developers isolate faults to certain parts of the algorithm in a coarse-grained way. In our scheme, we enforce that the outer solver be reliable, while letting the inner solver run in an unreliable mode. We aim to spend most of our computation time in cheap "unreliable" computations, while minimizing the time we spend in the presumably expensive outer solve.

Analytically, any faults that occur in the inner solver manifest as a "different preconditioner" to the outer solver. We choose Flexible GMRES [9] as the outer solver, since it can tolerate a preconditioner that changes between iterations. As an inner solver, we use the Generalized Minimal Residual Method (GMRES) from Saad and Schultz [10]. We show results for this inner/outer solver system, called FT-GMRES, that uses a multigrid preconditioner (MueLu) and solves a Poisson problem. Multigrid is the preferred preconditioner for Poisson problems.

2.3 Implementation

We implemented our solvers using the Tpetra [1] sparse linear algebra package in the Trilinos framework [7] and validated them against both MATLAB and the solvers in Trilinos' Belos package [2]. Implementing our solvers using Trilinos lets us benefit from the scalability and performance of its sparse matrices and dense vectors.

3. RESULTS

3.1 Methodology

We previously described how we corrupt the preconditioner's output. To evaluate the impact of our preconditioned solvers in the presence of SDC, we perform the following steps:

1. Solve the problem injecting no SDC, and compute the number of times, K, the preconditioner was applied.
2. For all j in $[1, K]$, reattempt the solve, introducing SDC at the j-th preconditioner application. This results in K total solves.
3. For all K solves with SDC, compute the relative percent of *additional* preconditioner applies over the SDC-free solve[1], e.g., $\frac{Applies_{observed} - Applies_{FailureFree}}{Applies_{FailureFree}} \times 100$
4. Repeat Steps 2 and 3, letting various numbers of MPI processes participate in the SDC injection.
5. Repeat Steps 2-4, varying the scaling factor applied to the SDC.
6. For each combination of scaling factor and number of faulty processes, plot the *average* number of additional preconditioner applies as a percentage. 0% means no additional applies; 100% means twice as many.

3.2 Model Comparisons

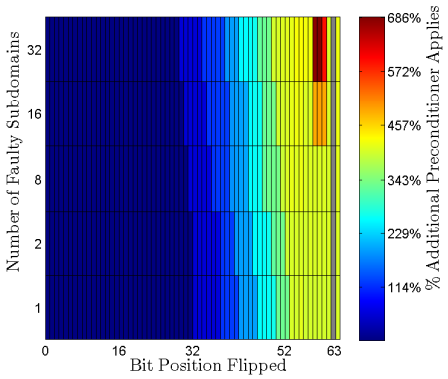
Fig. 1 shows a side-by-side comparison of the overhead introduced from random bit flips and our numerical fault model. Here, we use no detection mechanism and force our solver to roll through all errors.

Each plot represents a different fault model, so the results cannot be compared geometrically. The intent of the figure is to illustrate the overhead we observe given faults from each model. Recognize the overheads have roughly the same range, yet the variance in Fig. 1a is considerably higher than our model (Fig. 1b). We address this in greater detail in § 3.4.

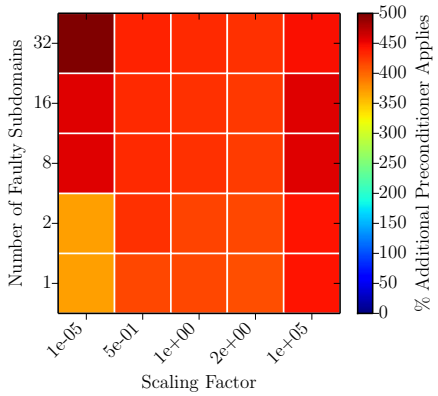
Note, Fig. 1a shows the highest overhead when all 32 subdomains inject the 58th or 59th exponent bit flips. This is not a weakness in our model. Those specific faults introduce very large magnifications into the vector, but not large enough to create an infinite or not-a-number value. Elliott et al. [3] explored exactly this scenario of faults and proposed a low overhead detector that efficiently filters such errors with $O(1)$ cost. For this exact reason our results analyze scaling factors that would slip through such a filter. Only the largest scaling factor, 1×10^5, would be detected by a projection bound.

Next, we enable both explicit residual ($\|Ax - b\|$) and projection bound tests per each inner iteration in Fig. 2. The resulting colorbar bounds are similar for both the numerical model and the bit flip model. That is, both models require a maximum overhead in the range of $100\% - 120\%$. This also exposes the trouble with bit flip injection: bit position does not affect the introduced overhead consistently. For example, exponent bits sometimes introduce high overhead, while mantissa bits can introduce overhead proportional to exponent bits. Notice that the right-most column of the numerical fault model is not the highest overhead — this is

[1]If $Applies_{observed} - Applies_{FailureFree} < 0$, i.e., SDC accelerated convergence, we record zero overhead.

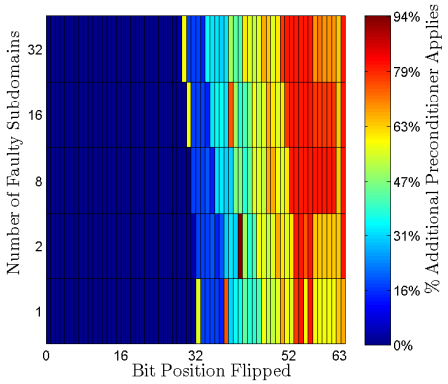


(a) Overhead given no attempt to detect and respond to faults with a random bit flip model.

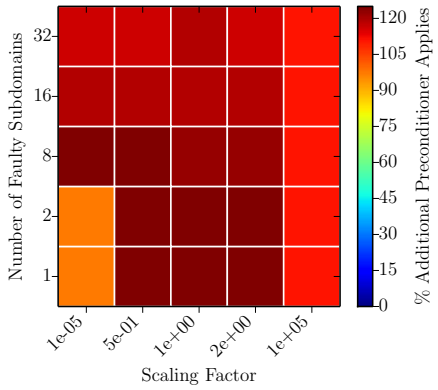


(b) Overhead given no attempt to detect and respond to faults with a numerical fault model.

Figure 1: Overhead comparison with no fault detection, for a numerical fault model (right) and random bit flip injection (left)



(a) Overhead when utilizing detection with a random bit flip model.



(b) Overhead when utilizing detection with a numerical fault model.

Figure 2: Overhead comparison with fault detection on, for a numerical fault model (right) and random bit flip injection (left)

due to a very low overhead detector, whereas checking the explicit residual requires a preconditioner apply.

3.3 Computational Effort

Our fault model captures hard to detect, yet numerically challenging faults. The largest overhead is easily characterized by our model and requires substantially less experimentation. For example, each shaded region constitutes one trial (of which we compute a mean). Clearly evaluating 25 unique experiments is much cheaper than evaluating 64×5 experiments. Moreover, if the experiment is not designed to account for the bias introduced by random bit flips, the mean will approximate optimistic overheads.

3.4 Expected Overhead Comparison

We compute the expected overhead across all experiments, i.e., compute the expected value for each row of these graphs. It becomes clear by inspection that our approach captures a worst-case scenario well beyond that of random bit flipping. That is, we are ensuring our algorithms can tolerate "bad" undetectable errors — error cases that random bit flips fail to expose — since we lack knowledge of exactly which values are the most sensitive. If an algorithm can handle our fault model, it can certainly handle the errors introduced by random bit flips.

We now compute the expected overhead given all samples for a given number of faulty subdomains. This computes the expected value for a "row" of the prior figures. For our numerical model, this entails grouping all scaling factors together, while for the bit flip model this considers an equally likely chance of flipping any of the 64 bits in the IEEE-754 representation.

Table 1 summarizes the expectation across all experiments for a given number of faulty subdomains when no reactive fault tolerance is used. This corresponds to Figures 1a and 1b. Clearly, our numerical faults are much worse than bit flips. Our model tends to create roughly **25** additional preconditioner applies, because our inner solver iterates 25 iterations (max) per inner solve. Our faults are sufficient to require an entire inner solve. We then obtain our solution in the next inner iteration, requiring roughly the failure free number of iterations (6). This gives a total iteration (and preconditioner apply) count of approximately 25+6. Our model has significantly smaller variance than what random bit flips

would have introduced. This indicates that our model *consistently* introduces poor behavior, which is our intent.

Table 1: Additional preconditioner applies given no fault detection; percent additional applies in parentheses.

	Additional Preconditioner Applies			
Faulty	Bit Flips		Numerical	
Subdomains	mean	StdDev	mean	StdDev
1	7.28 (121%)	10.94	24.73 (412%)	5.05
2	7.56 (126%)	11.08	24.93 (416%)	5.07
8	8.11 (135%)	11.35	26.40 (440%)	1.90
16	8.56 (143%)	12.08	26.43 (441%)	1.89
32	9.51 (158%)	13.38	26.93 (449%)	2.85

Table 2 analyzes the overhead if we check explicit residuals and projection lengths [3] inside the inner solver (Fig. 2a). Again, we show that random bit flips present very optimistic overheads. The reason for this was rigorously addressed by Elliott et al. [4]. Even with fault detection enabled, our fault model is still sufficient to show high overhead. This is desired. The faults we introduce are not necessarily detectable immediately. This forces our solvers to iterate 2-3 iterations before finally reaching a divergent state that is detectable. These are precisely the events we wish to study — faults that are undetectable, yet cause the solvers to eventually reach an invalid state. This motivates the study of low-overhead detection mechanisms.

Table 2: Additional preconditioner applies **with reactive fault tolerance**; percent additional applies in parentheses.

	Additional Preconditioner Applies			
Faulty	Bit Flips		Numerical	
Subdomains	mean	StdDev	mean	StdDev
1	1.49 (25%)	2.54	7.00 (117%)	2.12
2	1.56 (26%)	2.48	7.00 (117%)	2.12
8	1.69 (28%)	2.35	7.27 (121%)	1.72
16	1.81 (30%)	2.74	7.07 (118%)	1.74
32	1.83 (30%)	2.54	6.97 (116%)	1.83

4. CONCLUSION

We have presented results based on a fault model that allows us to characterize the numerical errors introduced by faults, and have shown that this model encompasses the range of overhead that the random bit flip model can introduce. Our fault model does not aim to predict the actual behavior of SDC. Rather, it shows a case sufficiently "bad" for us to assess how our fault-tolerance strategies behave when presented with very damaging SDC.

Our approach is a very different way of assessing preconditioned iterative linear solvers given an uncertain fault model. Rather than focus on what specifically constitutes a fault, we force our solvers to work through numerically challenging events. We specifically tune our fault model to inject errors that are not necessarily detectable. Our errors live inside the solvers' valid norm bounds, and empirically we observe our errors may cause divergence in latter iterations rather than immediately.

We compare our model to that of random bit flips, showing that random bit flip injection is *not* likely to show worst-case overhead. We support this through a methodical injection of bit flips, and by computing statistics over all experiments, as well as per bit position. Furthermore, we show our approach produces very predictable variance, irrespective of the number of processes that are faulty.

Acknowledgment

This work was supported in part by grants from NSF (awards 1058779 and 0958311) and the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research, under Program Manager Dr. Karen Pao. Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

5. REFERENCES

[1] C. G. Baker and M. A. Heroux. Tpetra, and the use of generic programming in scientific computing. *Scientific Programming*, 20(2):115–128, 2012.

[2] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist. Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems. *Scientific Programming*, 20(3):241–255, 2012.

[3] J. Elliott, M. Hoemmen, and F. Mueller. Evaluating the impact of SDC on the GMRES iterative solver. In *28th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2014)*, Phoenix, USA, May 2014.

[4] J. Elliott, M. Hoemmen, and F. Mueller. Exploiting data representation for fault tolerance. In *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '14, pages 9–16, 2014.

[5] A. Geist. What is the monster in the closet? Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking, Aug. 2011.

[6] M. Heroux. Scalable Computing Challenges: An Overview. Minisymposium talk at SIAM Annual Meeting: Supercomputing Challenges: Petascale and Beyond, July 2009.

[7] M. A. Heroux et al. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[8] P. Kogge et al. ExaScale computing study: Technology challenges in achieving exascale systems. Technical report, Defense Advanced Research Project Agency, Information Processing Techniques Office, 2008.

[9] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[10] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986.

[11] J. Schlaich and K.-H. Raineck. Die Ursache für den Totalverlust der Betonplattform Sleipner A. *Beton- und Stahlbetonbau*, 88:1–4, 1993.