TOWARDS A MORE FAULT RESILIENT MULITGRID SOLVER

JON CALHOUN*, LUKE OLSON*, AND MARC SNIR*

Abstract. Much is known about properties of linear solvers with regard to their stability, convergence rates, complexity, and efficiency, but little is known about their ability to handle bit-flips that can lead to silent data corruptions (SDCs). As supercomputers continue to add more cores to increase the performance of the machine, they are becoming more susceptible to SDCs. Going forward it is paramount that studies on the impact of SDCs on algorithms and applications in widespread use be conducted. This paper looks at the linear solver Algebraic Multigrid in a environment where bit-flips are possible. We propose an algorithmic based detection and recovery scheme that maintains the numerical properties of AMG while maintaining near perfect convergence rates in faulty environments.

Key words. Algebric Multigird, Resilience, Fault Tolerance, Silent Data Corruptions

1. Introduction. Many scientific applications from modeling blood flow to electromagnetics depend upon sparse matrix structures, which are central to the underlying linear algebra computations. These types of computations comprise a sizable percent of high performance computing (HPC) workloads. One of the most crucial computations is solving a linear system. Much is known about the efficiency and scalability of linear solvers, but their behavior in the presence of faults is still unclear. Current and emerging HPC architectures are expected to experience higher levels of faults than before. In order to efficiently utilize these resources, we need to elevate fault consideration to a first class priority. Resiliency techniques need to be developed and analyzed to allow linear solvers the ability to remain efficient and scalable on emerging HPC architectures.

Modern scientific and engineering computing relies on solving large systems of linear equations. The matrices that arise from modeling real world phenomena often have sparse structures. Sophisticated solvers can be employed to take advantage of this sparsity. One popular solver that has shown its flexibility across a range of different architectures is Algebraic Multigrid (AMG) [11] [21] [5]. AMG continues to gain interest from many researchers in both industry and academia due to its potential scalability, robustness, and efficiency as a $\mathcal{O}(n)$ linear solver. As computing capabilities progress over then next several years demands on the solver will increase.

As machines are built using a higher number of cores the individual cores themselves are not becoming more reliable, therefore, as the number of cores in a system increases, the mean time between interruption decreases. Because of this, fault tolerance and resilience are receiving increased attention. Most of this attention is devoted to developing traditional checkpoint-restart libraries [17] [4], however, some approaches are based upon a reworking of the algorithm itself to reduce its susceptibility to faults [8].

Main deterrents for checkpoint-restart schemes are increases in time and energy for a run to complete. Modern approaches attempt to address these issues, but more can be done via the use of algorithmic approaches. The use of algorithmic based fault tolerance will increase the resiliency of the application while at the same time lowering the time and energy required. The focus of this paper is on utilizing algorithmic based fault tolerance to improve resiliency of AMG. Understanding the level of resiliency that can be provided by AMG on emerging architectures is vital if AMG is to become the solver of choice for current and future machines.

To motivate the need for fault detection and recovery in AMG, let's look at the residual history, Figure 1.1, for the solution of a 2D anisotropic diffusion problem, the problem used in Section 4. A single fault is inject into the residual calculation before restriction during the second iteration on the second coarsest level. This fault is a single bit-flip in the first element of the residual vector. This fault doesn't lead to a segmentation fault and is thus potently masked. Depending upon what bit was flipped, the number of extra iterations required to achieve the convergence tolerance of $1e^{-}11$ ranges from 0 to 136.

*Department of Computer Science, University of Illinois at Urbana-Champaign

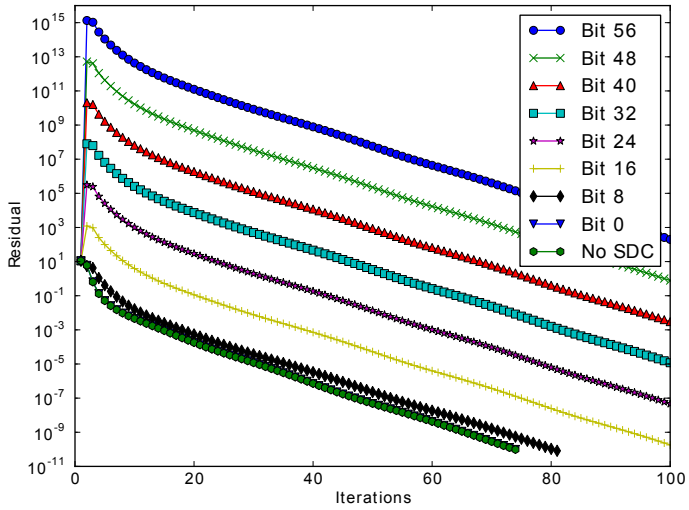

Fig. 1.1: Motivation for fault detection and correction.

To alleviate these extreme convergence times, this paper makes the following contributions:

- Low overhead algorithmic based recovery scheme for AMG
- Low cost SDCs detectors for iterative linear solvers
- AMG specific SDCs detectors

2. Background.

2.1. Algebraic Multigrid. Sparse matrix computation is a fundamental kernel in numerical computing. Solving the linear system $A\mathbf{x} = \mathbf{b}$ by direct methods incurs increasingly long execution times as the number of unknowns increases. These direct methods are ill-suited for the sparse nature of many science and engineering applications. Instead, we can create iterative algorithms that take advantage of these sparse systems. One such iterative algorithm is Algebraic Multigrid (AMG) [11] [21] [5].

AMG initially proceeds by refining the solution x_0 via the use of a smoother until its convergence slows. At this point all high frequency error has been removed, but low frequency error remains. This error can be smoothed out but with considerable work. The observation that this low frequency error will appear oscillatory if sampled on a coarser gird motivates the idea behind AMG. On this coarser grid AMG chooses to solve the residual equation $A\mathbf{e} = \mathbf{r}$, but since this system has less unknowns that the original the residual vector must be restricted via the use of a linear operator defined when constructing the coarser version of A. The residual equation is used because assuming that $\mathbf{e} = \mathbf{0}$ is a good initial guess. Solution to this newly formed linear system is accomplished by a smoother with the option to recurse and create another level in the hierarchy, or if the system is small or dense enough a direct solve for the error can be computed. Once a solution to an equation on a coarser gird is found, it must be interpolated to the finer grids and added into the solution on that level as a correction. However other traversal strategies are possible; traversing thought the levels from finest to coarsest and back to finest is know as a V-cycle and is the predominate way to traverse the AMG hierarchy.

2.2. Resilience. Resilience is not a new problem in computing. The first vacuum tube computers suffered a high failure rate, on the order of a few days. Once the problem was discovered, it took on the order of an hour to find the correct tube and replace the faulty tube. With the advent of transistors and integrated circuits, the reliability of the underlying hardware increased, but with the small feature sizes of today and computing in low power environments, fault awareness is making a resurgence.

Not every hardware component has the same level of reliability. DRAM and SRAM are susceptible to bit flips. This is why modern DRAM and SRAM have error correction codes (ECC) that guard against bit-flips.

The version of ECC most common on memory chips is single error correction - double error detection (SEC-DED). In this version, parity information is calculated for a bit field, and for every read of that location, the parity information is recomputed to verify that a bit-flip has not occurred. With this scheme, single bit errors will be detected and corrected, double bit errors will be detected, but not corrected. Every write to this location will cause the parity information to be recomputed.

The traditional way to handle transient and fail-stop errors is checkpoint-restart. In this approach, the application will run for some time and then save its state (in full or in part) to some permanent storage. Upon detection of a failure, the checkpoint file is read, data structures are rebuilt, and computation is restarted. Various variations of checkpoint-restart have been designed, with varying levels of required programmer intervention and checkpoint size and frequency [12] [18] [17] [4]. With certain fault considerations and applications, checkpointing is not even required [8] [1].

Going forward, detection of silent data corruptions (SDCs) will become a major focus in the fault tolerance/resilience community [6]. Extensive work has been done for SDC detection in numeric computations [2] [9]. General methods for detection and correction for SDCs have also been proposed [14] [19], but have yet to gain mass adoption.

2.3. AMG Resilience. The resiliency of AMG to SDCs has been studied previously [16], in which checksums were used to detect SDCs. This approach was limited to dense matrices, but it was fairly robust, having checks for matrix-vector multiplications, relaxation, and interpolation.

More recent work [7] has shown that AMG is resilient to SDCs due to its iterative and multi-level nature. Provided that AMG does not segmentation fault, an SDC will appear to AMG as error in the solution vector. This error will be removed at the cost of more work. This induced error can be removed by working on a coarser level, where this error will be more pronounced. Their recovery scheme addressed the segmentation fault issue by triplicating key pointers, and used a voting scheme among the three instances when accessed, accepting the pointer with the most votes. This voting is done for every access to the protected arrays, therefore, this resiliency scheme has a large ever present overhead, but it can decrease the number of segmentation faults for AMG in a faulty environment.

Our recovery scheme induces less overhead, even when AMG does not suffer a segmentation fault. In addition, our scheme provides checks that alert AMG to the presence of an SDC and attempts to remove it instead of blindly allowing AMG to remove the error via more iterations.

3. Contributions.

3.1. Recovery Scheme. As faults occur inside AMG, their effects can be classified as the following:

- Decreases convergence time
- Increases convergence time
- Converge to wrong solution (solving perturbed problem)
- Never converge (segmentation fault)

Of the possible outcomes, never converging due to a segmentation fault is a clear indication that an error has occurred. The other possible results from an injection will lead to an SDC, but the algorithm can continue to function. Our first concern is to handle segmentation faults in a way that will allow AMG to continue execution with minimal impact to convergence.

Segmentation faults are traditionally handled by allowing the application to crash and be restarted from a checkpoint or the beginning. Upon investigation of how AMG traverses its hierarchy, we can see that each level can be thought of as an implicit checkpoint, since the data for that level will be computed via the previous level. It is possible to build a recovery scheme that takes advantage of this implicit checkpoint. In order to exploit this facet of AMG, we need to define a segmentation fault handler that, upon invocation, will restart AMG on the previous level, or, if deemed serious enough, restart the cycle. This same method can be used to catch and recover from other signals that can arise due to experiencing a fault. The pseudocode executed by our segmentation fault handler can be found in Algorithm 1.

In order for our restart routine to execute correct we created global state information (direction, level, iteration) that the solver updates as it traverses through the hierarchy. Saving and restarting at some point in the hierarchy uses the `C` functions `sigsetjmp` and `siglongjmp` placed as to allow level or cycle restarts. The solution vector is periodically checkpointed in-memory on the finest level at the end of the V-cycle to limit the amount of roll back required. These minor augmentations were designed to be generic enough to be added to any current AMG implementation with minimal effort.

Algorithm 1: Multi-level Restart

```
if retrying on same level in cycle then
    x_k ← in-memory checkpointed vector on finest level
    restartLevel(finest)
if downpass == TRUE then
    if level ≠ finest_level then
        restartLevel(level − 1)
    if iteration > 0 then
        uppass ← TRUE
        downpass ← FALSE
        restartLevel(level + 1)
    else
        restartLevel(finest)
if uppass == TRUE then
    if level ≠ coarsest_level then
        restartLevel(level + 1)
    else
        uppass ← FALSE
        downpass ← TRUE
        restartLevel(level − 1)
```

3.2. Fault Detectors. Faults that do not lead to a segmentation fault will become silent data corruptions (SDCs) and will go unnoticed by the standard AMG algorithm. Simple augmentations to AMG can be added to allow the detection of these SDCs. These augmentations vary in their overhead, applicability to other codes, coverage, and recovery cost. Although application specific detectors are not portable, they offer the best chance of detecting SDCs.

3.2.1. Residual Check. The stopping criterion for AMG is the residual being less than a provided tolerance. If SDCs occur during a cycle, their effect will be shown in the residual calculated for that cycle, Figure 1.1, and every cycle thereafter until this added error is removed. Unlike some iterative linear solvers, AMG makes no guarantee that the residual or relative residual will monotonically decrease from cycle to cycle. Heuristics have shown that for a large class of problems, the residual does decrease almost monotonically, therefore, we can devise a low cost check that examines the newly calculated residual and compares it to the previous residual. Due to the non-monotonically decreasing nature of the residual, we must use a region of plausibility for the new residual. In this paper we consider anything less than the previous residual scaled by an order of magnitude to be an acceptable residual. The scaling factor of the previous residual in our check can be modified depending upon the type of problem being solved to enhance its ability to detect SDCs. This SDC detector is not just AMG specific; it can be applied to most iterative linear solvers.

3.2.2. Loop Checks. Important work on low cost methods for SDC detection can be found in [13]. Application agnostic SDC checks can be employed to improve the resiliency of AMG. For example, one check exploits the property that a significant number of fault injections are into loop control variables because of their high usage. Due to the regular nature of linear algebra computations, loops are highly structured and deterministic in their execution. These loops often iterate over a vector or through a row or column of a matrix. Since the stopping point for such loops is deterministic, we can verify that these loops terminate correctly as shown in Figure 3.1.

Upon closer examination, many loops in numerical linear algebra are idempotent. This key property allows the recovery for such loops to be attempted locally.

3.2.3. Energy Check. AMG does not guarantee that the residual will decrease monotonically at the end of every cycle. However, it does guarantee that the error will decrease monotonically with every iteration. Although we cannot measure error directly, we can use AMG's sense of energy, Equation 3.1, to check for

```
for (i = 0; i < n; i++)
{ ... }
if (i !=  n)
    handleSDC();
```

Fig. 3.1: Loop SDC Check

SDCs.

$$< A\mathbf{x}, \mathbf{x} > -2 < \mathbf{x}, \mathbf{b} > \tag{3.1}$$

This internal sense of energy is valid on each level. That is, energy at level i in the down-pass will be greater than the energy calculated at level i in the up-pass in the same V-cycle. We utilize the energy check as a guarantee that the results calculated on a given level are correct before we continue to the next level. This allows recovery to proceed by the multi-level restart algorithm, Algorithm 1.

4. Experimental Results. To test our recovery scheme and SDC detectors, a clean basic version of AMG was developed that allowed the SDC detectors and recovery scheme to be easily implemented and evaluated. All functions with available source code used during the AMG solve phase were compiled using our fault injector, with each instruction having the same probability of faulting, $1e^{-}9$. It is important to note that we do not inject faults into the creation of the hierarchy, but only during the solve phase.

To evaluate the effectiveness of our multi-level recovery scheme we chose to solve a 2D anisotropic diffusion problem. We chose this type of problem because historically it has been a challenge for AMG to solve efficiently. Because of the high number of iterations required, it has more innate susceptibility to transient faults.

This 2D anisotropic diffusion problem has 10000 unknowns on the finest level, and 7 levels in the hierarchy. The hierarchy is traversed using V-cycles with weighted Jacobi as the smoother and Gaussian Elimination for the direct solve. Averages and probabilities presented are based upon 1000 runs.

4.1. Fault Injector. Our fault model considers transient hardware errors that result from the CPU producing incorrect results. An LLVM [15] based fault injector was used in order to have tight control over where faults were injected, the ability to collect detailed statistics, and usability in both parallel and sequential environments.

Our fault injector is based off of [20], and is structured as an LLVM compiler pass. This compiler pass selectively examines the source code of a parameterized set of modules. For these modules, each instruction is surrounded by code that probabilistically injects a fault into an operand or the result of the LLVM instruction.

4.2. Overheads and Fault Characteristics. By their nature, transient faults are infrequent events, this implies that any resiliency scheme should limit the overhead introduced. To determine the practicality of our detectors; their ability to detect is often offset by their cost, overhead. Table 4.1 details the overhead of each SDC detector used. For every detector typed used we also enable the multi-level restart scheme. Because we are determining the overhead in a fault free case the multi-level restart code is never executed, therefore, we don't include the time to restart the AMG solve in the reported times.

Table 4.1: Fault Detector Overheads

Measure	Time (s)	Percent Overhead
No Resiliency	11.6753	-
Residual	11.7526	0.66%
Loop	12.1600	4.15%
Energy	13.2110	13.15%
All	14.1223	20.96%

We can see that the residual and loop checks are much cheaper than the energy check. This is offset by the fact that the energy check is more powerful, able to check for SDCs at each level. Preforming the energy check on each level will limit the amount of the V-cycle that needs to be recomputed, thus offsetting the high overhead.

In order to devise effective resiliency schemes we need to determine where faults are injected into AMG. To determine this we ran 1000 runs of our AMG solver with various SDC detectors enabled. *No Resiliency* represents a pure AMG with no added resiliency, *Low Cost* consists of the multi-level recovery scheme combined with the loop and residual checks, and *All* is the same as *Low Cost* expect the addition of the energy check. The percentages of where faults are injected can be found in Table 4.2, and the break down of what types of faults (*Pointer*, *Control*, *Arithmetic*) per function is shown in Table 4.3. The classification *Pointer* refers to all calculations directly related to use of a pointer (loads, stores, address calculation). *Control* refers to all calculations of branching and control flow (comparisons for branches and loop control variables modification). The final category of injected faults, *Arithmetic*, refers to the pure mathematical operations.

Table 4.2: Injected faults into AMG with various detectors

Function Name	No Resiliency Percentage	Low Cost Percentage	All Percentage
`jacobi`	82.0	81.3	76.1
`csr_matvec`	16.2	16.6	22.3
`residual`	0.3	0.9	0.6
`residua_norm`	0.5	0.2	0.3
`solve`	1.0	1.0	0.7

Table 4.3: Fault Characteristics with various detectors

Function Name	Fault Category	No Resiliency Percentage	Low Cost Percentage	All Percentage
`jacobi`	Pointer	55.8	54.6	53.6
	Control	25.9	28.7	27.7
	Arithmetic	18.3	16.7	18.7
`csr_matvec`	Pointer	68.5	67.8	62.5
	Control	17.5	17.8	21.7
	Arithmetic	14.0	14.4	15.8
`residual`	Pointer	85.7	58.6	73.7
	Control	0.0	37.9	15.8
	Arithmetic	14.3	3.5	10.5
`residual_norm`	Pointer	54.5	33.3	62.5
	Control	9.1	50.0	25
	Arithmetic	36.4	16.7	12.5
`solve`	Pointer	82.6	64.5	61.9
	Control	17.4	19.4	33.3
	Arithmetic	0.0	16.1	4.8

From the tables we can see that most of the faults are injected into the `jacobi` function. This is due to the code spending most of its time in that function. As we add the energy check that requires a matrix-vector multiply we see the number of faults injected into the `csr_matvec` routine increases. It is interesting to see that the majority of faults injected were into *Pointer* instructions. This is due impart to using sparse matrix data structures which requires several operations before the data is used. This high level of injected also signals that segmentation faults are likely, thus a recovery scheme to handle them is needed. We can see a corresponding increase in the *Control* classification when the loop check is added.

Adding our SDC detectors can increase resilience, but can also suffer from false positives. The loop and

residual checks are structured such that a bit-flip in the comparison will still evaluate correctly in all cases when more than 2 bits are set in the correct comparison result. A false positive in the energy is handled as if it was a true positive.

4.3. Convergence Analysis. With iterative linear solvers convergence can vary dramatically depending upon the initial guess and right hand side. To eliminate this variability we use the same initial guess and right hand side. Since an injected fault can cause an increased time to converge, we set the maximum number of iterations at 80. This allows 6 more iterations to converge in than the fault free 74 iterations. Table 4.4 analyzes the convergence of AMG with just a single injection, while Table 4.5 is reserved for multiple injection per run. In both tables *Converged* refers to runs that converged the specified tolerance within the 80 alloted iterations, *Didn't Converge* refers to those runs that didn't crash but were unable to converge within the alloted iteration count, and *Segfault* is used to classify those runs that fail to converge due to segmentation faults that causes the program to crash and not terminate correctly.

Table 4.4: Convergence with single injection

Results	No Resiliency Percentage	Low Cost Percentage	All Percentage
Converged	66.4	99.8	100.0
Didn't Converge	1.2	0.2	0
Segfault	32.4	0	0

Even just a single injection has drastic impact upon convergence, with almost one-third of all runs experience a segmentation fault before converging. A high rate of segmentation faults is expected because most of our faults are being injected into *Pointer* instructions. Our multi-level recovery scheme is able to recover from these segmentation faults and with the aid of our SDC detectors converge in almost all cases. The cases that didn't converge were very near convergence, and would have converged within a few extra iterations.

Table 4.5: Convergence with multiple injections (avg. 3 per run)

Results	No Resiliency Percentage	Low Cost Percentage	All Percentage
Converged	15.1	100.0	99.8
Didn't Converge	2.7	0	0.2
Segfault	82.2	0	0

With multiple injections convergence for the non resilient AMG implementation decreases dramatically, however, our augmented AMG remains at near perfect convergence. It is interesting to note that upon examining the residual history for runs in which the multi-level recovery scheme is active, convergence occurs in fewer iterations than in the fault free cause. This seemingly strange behavior is possible because recovery starts on a level that has already be executed, and upon the restart on this level further refinement occurs on its solution vector.

4.3.1. Time to Convergence. A valid and often used recovery scheme to handle the program produces incorrect output due to faults is to rerun program until correct output is produced. With this recovery scheme no detectors are used with the assumption that the incorrect run is the exception rather than the norm. We can compare our multi-recovery scheme to this approach by calculating an approximate convergence time based upon the ratio of the average time to converge to the probability that it will converge. Table 4.6 compares these two recovery schemes with our normal probability of injecting a fault of $1e^{-}9$, and an order of magnitude higher injection rate, $1e^{-}8$, is presented in Table 4.7. Because we are running until convergence, this process may be infinite. We therefore impose a time limit, 2 minutes, that allows several attempt at solving the linear system.

Even with out lowest injection rate the rerun recovery scheme fails to provide sufficient convergence. The reason for this is most of the injected faults lead to segmentation faults that the rerun recovery scheme

Table 4.6: Approximate time to converge (Injection rate $1e^{-}9$)

Detector(s)	Prob of convergence	Avg time to converge (sec)	Approx converge time	Speedup
No Resiliency	0.202	34.32	169.90	1.0
Low Cost	0.998	29.13	29.19	5.82
All	1.00	32.00	32.00	5.31

Table 4.7: Approximate time to converge (Injection rate $1e^{-}8$)

Detector(s)	Prob of convergence	Average time to converge	Approx converge time	Speedup
No Resiliency	0.0	—	—	—
Low Cost	0.984	27.01	27.45	—
All	1.00	29.17	29.17	—

can not recovery from without high cost. From these tables we can conclude, that if AMG is used inside a faulty environment our multi-level resiliency provides excellent convergence and a lower time to convergence even with an order of magnitude higher injection rate. To see what level of injection is required to cause our most resilient version, *All*, to fail to converge acceptably, we repeat the previous runs, but we don't retry if convergence isn't achieved on the first try. In Figure 4.1, we increase the level of injection until the probability of convergence drops below 0.5.

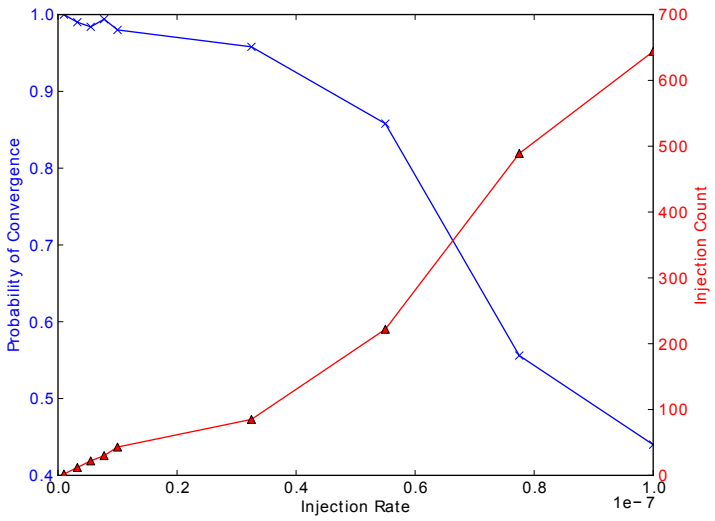

Fig. 4.1: Convergence of AMG with multi-level recovery scheme.

Even with over 200 faults injected per run our most resilient version of AMG converges in over 85% of runs. However after this point the probability of convergence decreases dramatically. This is due the the solver making little, if any forward progress. When a fault is detected we attempt to recover, but as we are in the recovery process we suffer more faults that require a restart. As we increase the injection rate AMG makes no forward progress and is constantly attempting to recover from the inject faults.

5. Conclusions and Future Work. Through a combination of general purpose and application specific detectors we are able to detect SDCs that significantly impact convergence of AMG. With faults detected our proposed multi-level recovery scheme is able to recover and converge with near perfect probability for a

high number of faults. Going forward fault consideration is going to become a driving factor in HPC software development and deployment. With the AMG augmentations presented and evaluated in this paper, AMG has shown it is capable of being used in faulty environments.

Although we've shown that a multi-level recovery scheme for AMG is possible, much can be done to limit the overhead. For example, limiting the locations detectors are placed or not recovering from every detected fault, but only those that would extend convergence time. Other future work includes porting theses ideas into a parallel setting as augmentations to established AMG solvers such as Hypre [10] and PETSC [3].

REFERENCES

[1] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Jean Roman, and Mawussi Zounon. Towards resilient parallel linear Krylov solvers: recover-restart strategies. Rapport de recherche RR-8324, INRIA, July 2013.

[2] Cynthia J. Anfinson and Franklin T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Trans. Computers*, 37(12):1599–1604, 1988.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[4] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ACM.

[5] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial (2Nd Ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[6] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, November 2009.

[7] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 91–100, New York, NY, USA, 2012. ACM.

[8] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 73–84, New York, NY, USA, 2011. ACM.

[9] James Elliott, Frank Mueller, Miroslav Stoyanov, and Clayton Webster. Quantifying the impact of single bit flips on floating point arithmetic. Technical report, Oak Ridge National Laboratory, August 2013.

[10] Robert D. Falgout and Ulrike Meier Yang. hypre: A library of high performance preconditioners. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part III*, pages 632–641, London, UK, 2002. Springer-Verlag.

[11] Wolfgang Hackbusch. *Multi-grid methods and applications*, volume 4 of *Springer series in computational mathematics*. Springer, Berlin [u.a.], 1985.

[12] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[13] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[14] Man lap Li, Pradeep Ramach, Swarup K. Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Swat: An error resilient system, 2008.

[15] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[16] Amitabh Mishra and Prithviraj Banerjee. An algorithm-based error detection scheme for the multigrid method. *IEEE Trans. Comput.*, 52(9):1089–1099, September 2003.

[17] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[18] Bogdan Nicolae and Franck Cappello. Ai-ckpt: Leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 155–166, New York, NY, USA, 2013. ACM.

[19] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. mswat: Low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 122–132, New York, NY, USA, 2009. ACM.

[20] Vishal Chandra Sharma, Arvind Haran, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Towards formal approaches to system resilience. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2013. to appear.

[21] Ulrich Trottenberg and Anton Schuller. *Multigrid*. Academic Press, Inc., Orlando, FL, USA, 2001.