Towards a More Fault Resilient Multigrid Solver

Jon Calhoun jccalho2@illinois.edu
Luke Olson lukeo@illinois.edu
Marc Snir snir@illinois.edu
William D. Gropp wgropp@illinois.edu

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

ABSTRACT

The effectiveness of sparse, linear solvers is typically studied in terms of their convergence properties and computational complexity, while their ability to handle transient hardware errors, such as bit-flips that lead to silent data corruption (SDC), has received less attention. As supercomputers continue to add more cores to increase performance, they are also becoming more susceptible to SDC. Consequently, understanding the impact of SDC on algorithms and common applications is an important component of solver analysis. In this paper, we investigate algebraic multigrid (AMG) in an environment exposed to corruptions through bit-flips. We propose an algorithmic based detection and recovery scheme that maintains the numerical properties of AMG, while maintaining high convergence rates in this environment. We also introduce a performance model and numerical results in support of the methodology.

Author Keywords

Algebraic Multigrid, Silent Data Corruption, Fault Tolerance, Resilience

INTRODUCTION

Many scientific applications, from modeling blood flow to electromagnetics, depend on sparse matrix structures and linear algebra computations. These types of computations often consume a sizable percent of a high performance computing (HPC) workload. In particular, one crucial operation is the sparse, linear solve. Scalability and convergence of such methods are well studied, but their behavior in the presence of hardware faults is less developed. Current and emerging HPC architectures are expected to experience higher levels of faults than previous architectures and understanding the impact of fault(s) on the *algorithm* is an important component in fully utilizing their resources. Consequently, 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 computing relies on solving large, sparse systems of linear equations. Sophisticated solvers are employed to take advantage of this sparsity, and as computing capabilities continue to progress so do the demands on the linear solver. One solver that has shown to be flexible across a range of different architectures is algebraic multigrid (AMG), due to its potential scalability, robustness, and efficiency as an $\mathcal{O}(n)$ complexity method.

HPC 2015, April 12-15, 2015, Alexandria, VA.

As machines are built using a higher number of cores the individual cores themselves are not becoming more reliable [3]. Therefore, as the number of cores in a system increases, the mean time between interruptions decreases. As a result, fault tolerance and resilience are receiving increased attention. Most of this attention is devoted to developing traditional checkpoint-restart libraries [13, 16]. Checkpoint-restart is designed to correct fail-stop errors that do not allow the application to proceed once the error manifests. A main disadvantage of a typical checkpoint-restart scheme is the increase in time and energy to complete a run. Advanced approaches attempt to address these issues, but techniques at the *algorithm* level offer an opportunity to further enhance resilience. Some approaches use mathematical theory and numerical methods in-order to re-construct the missing data due to a fail-stop error [7, 1].

The increase in the number of circuits, coupled with the decrease in feature size, and the the use of low power techniques are also likely to increase the frequency of silent data corruption (SDC) and poses a major problem to the future of large scale scientific computing [5, 20]. General methods for SDC detection leverage redundancy [12], but algorithm based fault tolerance (ABFT) has the potential to reduce the overhead for handling SDC by leveraging algorithm dependent heuristics or invariants to detect and recover from SDC [8, 10]. Furthermore, the use of ABFT increases the resiliency of the application while at the same time lowering the time to solution in faulty environments.

The focus of this paper is on utilizing ABFT to improve resiliency of AMG. Understanding the level of resiliency that is provided by AMG on emerging architectures is important for AMG and other sparse, linear solvers.

To motivate the need for silent data corruption (SDC) detection and recovery in the sparse, linear solve, in Figure 1 we consider the residual history for the solution of a diffusion problem in 2D (cf. Section 5). In this example, a single fault is injected into the residual calculation before restriction during the second iteration on the finest level. This fault is a single bit-flip in the first element of the residual vector, a IEEE 754 double precision floating point number, on process 0 of a 16 process job. The fault does not lead to a segmentation fault or *segfault*, and is thus potently silent, but influential in its impact on the algorithm. Depending on *which* bit is flipped, the number of extra iterations required to achieve the convergence tolerance of $1e^{-7}$ ranges from 0 (bits in the mantissa) to double the original amount (bits in the exponent), thus motivating the need for SDC detection and recovery.

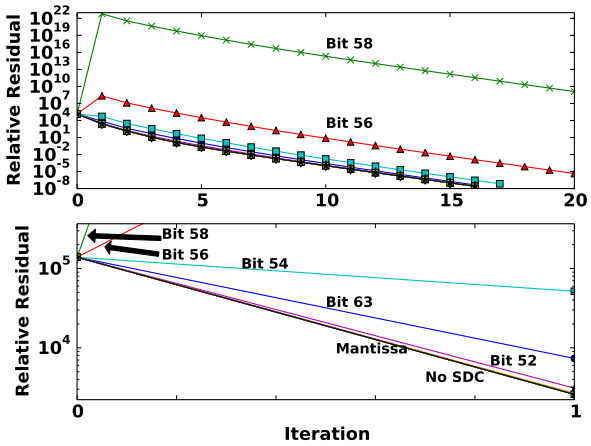

Figure 1. A single bit-flip in a IEEE 754 double precision floating point element in the residual vector's effect on the iterations needed to converge. In this location, a flipped bit in the exponent causes an increase in the iterations till convergence; while a flipped bit in the mantissa or the sign bit, bit 63, has no effect.

To this end, we improve and analyze the resilience of AMG in the presence of silent faults. In particular, in this paper we make the following contributions:

- low overhead algorithmic based recovery for AMG;
- low cost SDC detectors for iterative linear solvers; and
- AMG specific SDC detectors.

BACKGROUND

Algebraic Multigrid

Consider the sparse matrix problem $A\mathbf{x} = \mathbf{b}$. In a parallel setting, iterative solution techniques are preferred due to the memory and complexity requirements to solve. One such approach is algebraic multigrid (AMG) [18], which is often used as a preconditioner for a Krylov method such as conjugate gradient (CG) or generalized minimum residual (GMRES).

AMG constructs a hierarchy of successively coarser problems that are used to iteratively refine the error in the solution. Figure 2 outlines a *solve* cycle of AMG, where an initial guess x^0 is refined using two compatible operations: relaxation, such as weighted Jacobi, and coarse-grid correction, which is a subspace projection method. Relaxation is responsible for reducing high energy error, while coarse grid correction targets *algebraically* smooth error, or error that is invariant to relaxation. The cycle proceeds by successively coarsening the error equations, $A\mathbf{e} = \mathbf{r}$, until a coarse level problem is effectively solved in a few steps of relaxation or is efficiently processed with a direct method. Errors on coarse levels are then interpolated to correct solutions. Figure 2 represents the well-known V-cycle and is a common approach to traverse the AMG hierarchy in parallel.

Resilience

Resilience has been a challenge since the early days of computing. Vacuum tube based machines had a mean-time-between-failure (MTBF) of a few days. With the introduction

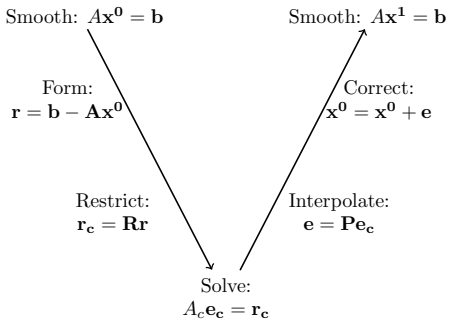

Figure 2. AMG V-cycle process used to solve $A\mathbf{x} = \mathbf{b}$

of transistors and integrated circuits, the reliability of the underlying hardware increased, but width the small feature sizes and low-power environments in emerging architectures, fault awareness is making a resurgence [3].

All hardware components do not exhibit the same level of reliability. DRAM and SRAM are susceptible to bit-flips, which has led to error correction codes (ECC) and so-called *chipkill* that protect against bit-flips in memory [21]. These methods, at the cost of area and power on chip, seek to recover the data by stored parity information. In the case of chipkill, a multi-bit errors and in some cases the entire memory chip can be detected and repaired.

The traditional way to handle fail-stop errors is checkpoint-restart [5]. In this approach, the application executes for a given time and saves its state (in full or in part) to permanent storage. In the event of a detected failure, a checkpoint file is read, data structures are rebuilt, and computation is restarted. A variety of checkpoint-restart schemes have been designed, with varying levels of intervention required by the programmer, checkpoint sizes, and checkpoint frequencies considered [16, 13]. In addition, for certain fault considerations and applications, checkpointing is not required [7, 1].

Emerging architectures are demanding attention to silent data corruption (SDC). Extensive work has been done for SDC detection in numerical computations [14, 2, 11, 8] , whereas general methods for detection and correction for SDC have also been proposed [19, 9, 12], but have yet to gain wide adoption.

AMG Resilience

The resiliency of AMG to SDC has been studied previously [15], where checksums are used to detect SDCs. The approach is limited to dense matrices, but is robust, incorporating checks for matrix-vector multiplications, relaxation, and interpolation.

More recent work [6] reports that AMG is resilient to SDC due to its iterative and multilevel nature. Provided that AMG does not segfault, a SDC with high probability emerges as an error in the solution vector. This error is subsequently removed at the cost of more work. Moreover, the error is reduced on a coarser level, where this error is more pronounced. The recovery scheme addresses a segfault by triplicating key pointers and ranks the three instances when accessed, accept-

ing the pointer with the most credibility. This is done for every access to the protected arrays, therefore yielding resiliency at a large overhead. Regardless, the approach decreases the number of segfaults for AMG in a faulty environment, yet the resiliency overhead is present even in the case when AMG is not exposed to faults.

In contrast, our recovery scheme induces less overhead, even when AMG does not suffer a segfault. In addition, our scheme provides checks that alert the AMG solver of the presence of SDC and attempts to avoid the impact of the fault instead of allowing AMG to remove the error through more iterations.

TRANSIENT ERROR DETECTION

We first discuss our error model and assumptions. For this method, we assume that memories are sufficiently protected with ECC and chipkill; as such, we do not model SDC arising in memory. Further, we assume that errors emerge during instruction execution in the solve phase of AMG and that corruptions manifest themselves as bit perturbations is the result of instructions. In addition, during the solve phase the operators A, P, and R on each level as well as the right hand side $\mathbf{b}$ are never written only read. As a result, these structures are not checkpointed for SDC recovery since they are only modified through errant stores. To protect these data structures from such writes, one approach is to use the system call `mprotect` to force the associated memory pages to be read-only. In the case of a write to a protected page, a segfault is raised triggering recovery. This function is used in some asynchronous checkpoint-restart approaches [17] to exploit copy on write capabilities when scheduling pages to be checkpointed.

MultiLevel Recovery Scheme

As faults occur inside AMG, we classify them as the following. A fault,

- decreases the convergence time;
- increases convergence time;
- leads to convergence to wrong solution; or
- unexpected program termination (crash).

In the case of divergence due to an unexpected termination — e.g., segfault — this is a clear indication that an error has occurred. The other possible results from faults lead to silent data corruption (SDC), but the algorithm still continues to function. SDC may also lead to longer runtimes, or to converging to the wrong solution. We rely on mathematical theory and heuristics of AMG to detect these errors as discussed below.

Checkpoint-restart is the defacto method to recover from failure by allowing the application to crash and restart from a checkpoint or beginning state. We utilize this idea in our recovery from detected errors. By observing the traversal of the AMG hierarchy, we consider each level an *implicit* checkpoint, since the data for that level is computed via the previous level in the hierarchy. It is possible to build a recovery scheme that takes advantage of this implicit checkpoint. In order to exploit this, we define a recovery function to to restart AMG on the previous level, or if the error is deemed severe, restart the cycle. The code executed by our multilevel recovery scheme is found in Algorithm 1. If a fault is detected, we first determine its severity. Provided the fault is minor we recover from the previous level in the AMG hierarchy. Should a SDC be flagged in the same location again, or the residual check, discussed below, is triggered we rollback to our previous *explicit* checkpoint and restart the solve on the finest level.

Algorithm 1: Multilevel Restart

```
1  if retry-same-level or residual-check-failed then
2      restoreFromCheckpoint (x^k)                    {in-memory, fine level}
3      restartLevel (lev_finest)
4  if down-pass then
5      if lev ≠ lev_finest then
6          restartLevel(lev − 1)
7      if iteration > 0 then
8          up-pass ← TRUE
9          down-pass ← FALSE
10         restartLevel(lev + 1)
11     else
12         restartLevel(lev_finest)
13 if up-pass then
14     if lev ≠ lev_coarsest then
15         restartLevel(lev + 1)
16     else
17         up-pass ← FALSE
18         down-pass ← TRUE
19         restartLevel(lev − 1)
```

In order for our restart routine to execute correctly, we create global state information comprised of direction, level, and iteration, that the solver updates as it traverses the hierarchy. Saving and restarting at some point in the hierarchy utilizes the C-language functions `sigsetjmp` to mark level or cycle restart locations and `siglongjmp` to facilitate the jumps. These functions store and restore the register state of the program at the point of the call respectively. These functions can be thought of as a label and a non-local `goto`. In addition, the solution vector periodically creates an in-memory checkpoint at the end of the V-cycle to limit the amount of roll back required. These minor augmentations are designed to be generic to allow them to be added to any sequential or parallel AMG implementation with minimal effort.

Silent Error Detectors

Faults that do not lead to a unexpected termination become SDC and go unnoticed by the standard AMG algorithm. Here, we add simple augmentations to AMG that allow the detection of SDC. These augmentations vary in their overhead, applicability to other codes, coverage, and recovery cost, as we detail below.

Residual Check

The typical stopping criterion for AMG is verifying that the relative residual is less than a given tolerance. If SDC occurs during a cycle, its effect is shown in the residual calculated for that cycle and every cycle thereafter until the error is removed. This is illustrated in our motivating example, Figure 1. Unlike other iterative linear solvers, AMG

solvers do not guarantee that the residual or relative residual monotonically decreases from cycle to cycle. Heuristics show that for a large class of problems, the residual is expected to decrease *nearly* monotonically; therefore, we devise a low cost error 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 use a region of plausibility for the new residual. In this paper, we consider a single order of magnitude in difference; while this is subjective, it is important to note that we are attempting to save the solver from a total restart as we detail in the next section. In addition, the scaling factor in our residual check may be modified depending on the type of problem being solved to enhance its ability to detect SDC and to prevent infinite loops. We also note that this SDC detector extends to other iterative solvers and is not specific to only AMG.

Energy Check

AMG does not guarantee that the residual decreases monotonically at the end of every cycle. However, it does guarantee a decrease in the A-norm of the error every cycle. Although we are unable to measure the error directly, we use AMG's sense of energy to check for SDCs, termed the *energetic stability*,

$$E = \langle A\mathbf{x},\, \mathbf{x}\rangle - 2\langle \mathbf{x},\, \mathbf{b}\rangle \tag{1}$$

The energetic stability is valid on each level. That is, energy at level i in the down-pass of the V-cycle is greater than the energy calculated at level i in the down-pass in the next 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 multilevel restart algorithm shown in, Algorithm 1.

In its current form, (1) is costly as it requires one sparse matrix-vector multiply (SpMV) and two inner products. The latter limits scalability at large core counts. On the down-pass and not on the coarsest level, the residual is formed. This operation provides us with $A\mathbf{x}$ at no cost. On the up-pass, the computation is more expensive, but we rewrite (1) in the following form

$$E = \langle \mathbf{r},\, \mathbf{b}\rangle - 2\langle \mathbf{x}-\mathbf{b},\, \mathbf{b}\rangle. \tag{2}$$

If we preform the local operations for both inner products and issue a single reduce that operates on the two local inner products in the same call, issues with scalability are minimized.

Segfault Recovery

Unlike the results of the energy or residual check where there is a global view of the result of each check once the operation completes, segfaults are local to the process on which they occur. One approach to recovery is to elevate a local segfault to a global segfault, thus allowing recovery via Algorithm 1, however, such a scheme can cause high overheads and will increase the amount of work redone. A parallel application is considered as a decomposition into two phases: communication and computation. AMG is composed of basic linear algebra operations such as SpMV operations and vector operations. All of these operations have an idempotent form at the expense of a temporary vector. For example, $y = Ax$ produces the same result independent of the number of times the operation is executed. Due to there idempotent nature, recovery from a segfault is straightforward: restart the operation by the use of the same `C` functions required for Algorithm 1, but use new recovery points for the idempotent operations. The error is transient and does not manifest itself as a segfault again. If the segfault occurs during a non-idempotent operation such as communication, recovery by restart using Algorithm 1 is possible, but important state information, communication library, may be corrupted – e.g. redundantly sent messages. By message logging we are able to eliminate redundant messages, but still face possible corruption in the communication library. As the results in Section 5 show, segfaults in non-idempotent regions represent a small portion of segfaults ($< 3\%$) for the results in Table 2 and Table 3; therefore, we do not add this extra overhead to our recovery scheme.

PERFORMANCE MODEL

In this section, we construct a performance model in order to highlight the scalability of our approach and the impact on computational complexity in the scheme. In particular, we develop this model for our algorithmic based detectors and multilevel recovery scheme in order to understand the impact on the convergence properties of the AMG solver and to see the potential benefits of the methodology. In Section 5.3, we present results from fault injection experiments on solves of AMG with various error rates and detectors enabled.

Basic Model

AMG is divided into two phases: setup and solve. These phases do not overlap leading to the basic performance model, for the runtime of the solution to $Ax = b$:

$$T_{\text{total}} = T_{\text{setup}} + T_{\text{solve}} \tag{3}$$

During the solve phase of AMG, there are several choices that influence the performance of the solver such as depth of the cycle, type of cycle, smoother, and etc. In response, our performance model below (see (4)) is designed to abstract many of these choices yielding a practical tool, while at the same time providing enough detail to draw firm conclusions. T_{cycle} is the time for a single cycle (iteration) in AMG, and n_{cycles} is the number of cycles until convergence. In our model, c_i and t_i are the number of visits and time on level i, respectively. Since we consider faults only in the solve phase, we introduce a term β that represents the percent increase in iteration count due to reduced convergence as result of a SDC. β is more accurately modeled by T_{repeat}, where r_i is the number of times a level is repeated due to a SDC.

$$\begin{aligned} T_{\text{solve}} &= T_{\text{cycle}} n_{\text{cycle}} (1+\beta) \\ &= T_{cycle} n_{cycle} + T_{repeat} \\ &= \textstyle\sum_{i=0}^{n_{\text{level}}} c_i t_i n_{\text{cycle}} + \sum_{i=0}^{n_{\text{level}}} r_i t_i \end{aligned} \tag{4}$$

The number of cycles (iterations) required for AMG to converge to d digits of accuracy is given by $\frac{d}{-\log_{10}(\rho)}$, where ρ is the convergence factor, the spectral radius of the iteration matrix used during the relaxation step. If faults occur during the solve phase, convergence *potentially* deteriorates to $\hat{\rho} = \alpha\rho$

such that $\rho < \hat{\rho} < 1$. As convergence deteriorates, additional cycles are required to achieve convergence to the same accuracy. In some situations, the increase is dramatic as outlined in Figure 1. Our detection and recovery schemes limit the number of extra cycles required by maintaining the average convergence factor of the fault free problem in most cases, or a few more iterations if the average convergence factor is only slightly affected.

Segfault Recovery

To implement the segfault recovery outlined in Section 3.3, we need to issue calls to `sigsetjmp` and `siglongjmp`. These calls, along with subsequent retry of the regions after a segfault, add time not reflected in our model (4). To extend our model we modify T_{cycle} in (4) with the addition of a $T_{\text{resiliency}}$ term that is the overhead of the added resiliency measures.

$$T_{\text{resiliency}} = T_{\text{seg}} \tag{5}$$

Here T_{seg} is the overhead of segfault recovery. Recovery from a segfault has us restarting the local operation. This operation time is less than the level time. Our model provides an upper bound on the restart time by defining r_i as the sum of the maximum number of segfaults experienced on a single rank on level i during each cycle.

Residual and Energy Check

To add the residual and energy checks to our performance model in (6), we make an extension to (5), where $T_{\text{checkpoint}}$ is the time to checkpoint the solution vector on the fine level, T_{residual} is the overhead of the residual calculation, and $T_{\text{energyDown}}$ and T_{energyUp} are the overhead of doing the energy check on each level on the down and up-pass respectively. This yields

$$T_{\text{resiliency}} = T_{\text{seg}} + T_{\text{checkpoint}} + T_{\text{residual}} + T_{\text{energyDown}} + T_{\text{energyUp}}. \tag{6}$$

Comparison of Time to Solution

Faults that occur during the solve phase that require extra iterations to converge, increase the average convergence factor. As the average convergence factor approaches 1.0 the number of iterations required to converge grows exponentially. Ultimately, the same *asymptotic* convergence factor is likely achieved, yet the *effective* convergence factor for the run increases. In Figure 3, we see the relative overhead is equal to β. Each trend is associated with a different value for β, which represents the percent increase in iteration count over the fault free case. That is $\beta = 1.0$ corresponds to taking twice as many iterations to solve the problem when compared to the fault free case, or a relative overhead of 1. The overhead due to a SDC is proportional to the amount of work that is redone. For example, if a problem's average convergence factor changes from 0.2 to 0.3, this is equivalent to $\beta = 0.3$ or a 1.3x increase in the number of iterations of the original problem. This is only compounded as the average convergence factor approaches 1.0 where the number of iterations required grows exponentially. Our methods keep β small which limits extra computation.

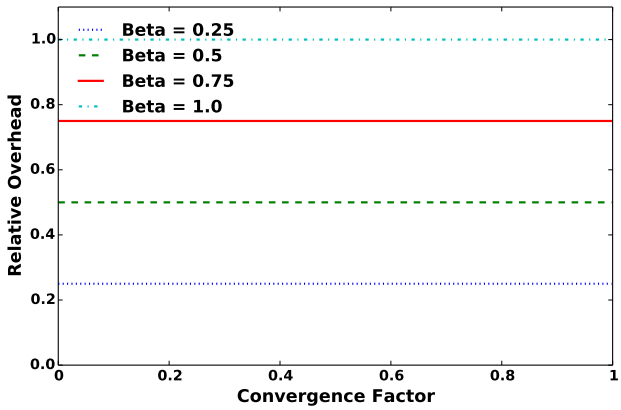

Figure 3. Relative overhead in converging to a fixed tolerance for various values of β.

EXPERIMENTAL RESULTS

To better understand AMG in a faulty environment, we use the fault injector FlipIt [4] to perturb with a bit-flip the result of a single dynamic instruction during the solve phase of the AMG solver of Hypre[1]. In the following (except Section 5.1), 1000 fault injection trials are conducted on Blue Waters where we solve a 2D Laplacian with zero on the boundaries using 16 processes and $16,384$ unknowns per process. Because of its rapid convergence and lightweight hierarchy, this problem highlights the overhead of our detectors and recovery scheme with more severity than a more difficult problem where the solver itself is more computationally heavy. Moreover, due to the rapid convergence for this model problem, the errors should be quickly removed by the efficient lightweight hierarchy. When a fault is injected it is classified by the fault injector in one of three main types: *pointer* refers to all calculations directly related to use of a pointers (loads, stores, and address calculation), *control* refers to all calculations of branching and control flow (comparisons for branches and modification of loop control variables), *arithmetic* refers to pure mathematical operations.

Overhead

By their nature, transient faults are infrequent events, which implies that any resiliency scheme should limit the overhead introduced in a fault free case. To determine the practicality of our detectors, it is critical to assess both the ability to detect as well as their cost (overhead). Figure 4 details the overhead with respect to solve phase execution time of each SDC detector used. We enable the multilevel recovery scheme for every configuration in which a detector is active. Because we are determining the overhead in a fault free case, the multilevel restart code is never executed; therefore, we do not include the time to restart the AMG solve in the reported times. We enable the energy check on every level to provide a worst case scenario. In addition, we in-memory checkpoint the solution vector at the end of every V-Cycle for the same reason.

[1] `https://computation-rnd.llnl.gov/linear_solvers/software.php`

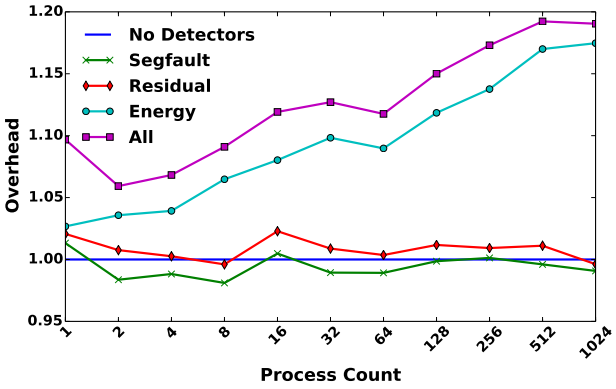

Figure 4. Overhead in solve time of fault detectors compared to the original AMG code.

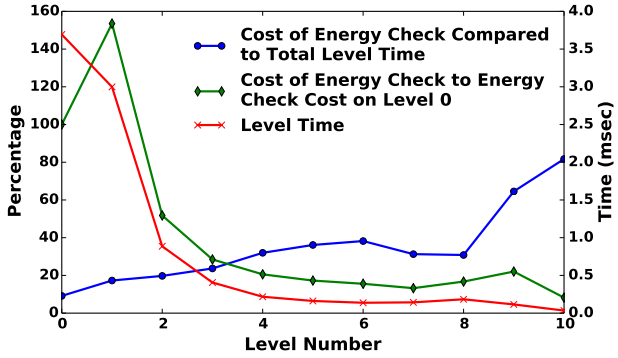

Figure 5. Characterization of cost of the energy check compared to total time on each level and cost relative to the energy check on level 0. Total level time is the sum of original level time and time for the energy check on that level.

We see that segfault (*Segfault*) recovery and the residual check (*Residual*) yield a lower overhead than a energy check (*Energy*). In fact, their runtimes are similar to the original code, differing by $\pm 1\%$ regardless of process count. 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 limits the amount of work that needs to be redone. Because of their relative low combined overhead ($< 1.5\%$), segfault recovery and the residual check, we refer to both of them together as the *Low Cost* configuration. Even with all detectors and our recovery scheme enabled in a worst case scenario, we have an observed maximal overhead of $< 20\%$. In practice, the frequency of a checkpoint, and on which levels the energy check is active would be modified to meet a more strict resiliency budget.

To better characterize the overhead of the energy check, in Figure 5 we see the percentage of level time consumed by the energy check. As we move from the finest level (0) to the coarsest level (10) the problem size decreases, but each level becomes more dense. As the density of the level increases, we see the energy check having a greater influence on level time. Although we have an increase in percentage of level time, the energy check is cheaper on the coarsest levels than on the finest level. The energy check is relatively more expensive on level 1 because although the number of unknowns in the problem is less on level 1, the number of non-zero entries is roughly the same leading to a denser matrix used in the check.

Fault Characteristics

In order to devise effective resiliency schemes, we need to determine where faults are injected into the AMG solve phase. We again investigate the worst case scenarios for both the residual and energy check. Faults are injected randomly throughout the solve phase with each dynamic instruction having a uniform probability of $1e^{-8}$. Seeding the fault injector differently for each trial yields injections in all iterations of the solve.

From Table 1 we see that most of the faults are injected into *Relaxation* and the *SpMV* routines, which includes A for the residual, P, and R. This is expected since most of time is spent in these routines. We classify *Other* routines as all other

Operation	No Detectors	Low Cost	All
Relaxation	50.1	47.4	42.5
SpMV	47	49.1	48.2
Inner product	1.1	1.7	6.7
Other	1.8	1.8	2.6

Table 1. Percentage of injections in AMG components.

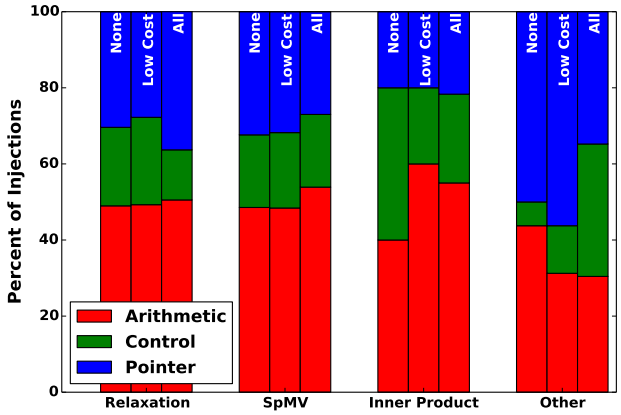

Figure 6. Breakdown of faults injected into AMG components based on the type of instruction executed.

functions used during the Hypre solve phase — e.g. cycling code, vector copy, and scaling routines. As we add the energy check that requires two inner products, we see a higher percentage of faults occurring in this routine.

Notably, in Figure 6, we see that a significant number of faults are injected into *pointer* and *arithmetic* instructions. The high degree of *pointer* injections is due to using sparse matrix data structures which uses indirection to access the data elements. A corrupted pointer often leads to a segfault, and corruption of *arithmetic* computation can produce results as shown in Figure 1, thus motivating the need for an efficient detection and recovery scheme.

Adding our SDC detectors increases resilience, but may also suffer from false positives. The comparisons for the residual and energy check represent a small portion of the dynamic

instructions during the solve phase and are unlikely to experience a SDC. The computation to form the quantities for the comparisons are more likely candidates. Regardless of where SDC occurs, a false positive in the residual or energy check is handled as if it was a true positive.

Convergence Analysis

In the following convergence study, we use the same initial guess and right hand side to limit variability in the results. Moreover, since an injected fault may lead to an increase in the number of iteration, we set the maximum number of iterations at 20. This allows 4 more iterations to converge.

Results	No Detectors	Low Cost	All
Converge	73.5	98.6	99
Did not converge	1	0.4	0
Crashed	25.5	1	1

Table 2. Percentage of trials that converge with a single injection.

In Table 2, we see that a single injection has a large impact on convergence, with 25.5% of all trials segfaulting before converging in the unmodified version of AMG. A high rate of segfaults is expected since one-third of injections are into instructions classified as *pointer*. In configurations where segfault recovery is enabled, our approach successfully recovers, thus allowing the solve to continue. With the aid of our SDC detectors we intercept 12% of injections. If uncaught, these injections would lead to extra iterations.

Results	No Detectors	Low Cost	All
Converge	0	86.2	84.8
Did not converge	0	2.8	0.2
Crashed	100	11	15

Table 3. Percentage of trials that converge with multiple injections. Average of 14 injections per trial for *Low Cost* and *All*. Average of 4 injections per trial for *No Detectors*.

With multiple injections, Table 3 shows a large deterioration in convergence for the unmodified AMG implementation with all trials crashing due to segfaults. However, our augmented versions of AMG remains convergent even in the presence of a high number of faults. Enabling the energy check increases sensitivity to SDC allowing us to detect more of them, but the check also incurs a cost. The energy check flags SDC at a rate that is 1.5x that of the residual check, but recovering from the SDC is more expensive than letting the AMG naturally iterate through the error. Some SDC that just triggers the energy check only slightly increases the energy from the previous iteration triggering the check, but not enough to have a high impact on convergence. In addition, the energy check also increases the parallel communication in routines where idempotent segfault recovery is not possible. Consequently, as the number of faults encountered per solve increases, the energy check becomes less useful.

We now look at how convergence is affected by the number of injections in both the *Low Cost* and *All* configurations. In Figure 7, we increase the average number of injections in each trial until the probability of convergence drops below 0.5. We compute the probability of convergence by dividing the number of trials that converge by the number of trials.

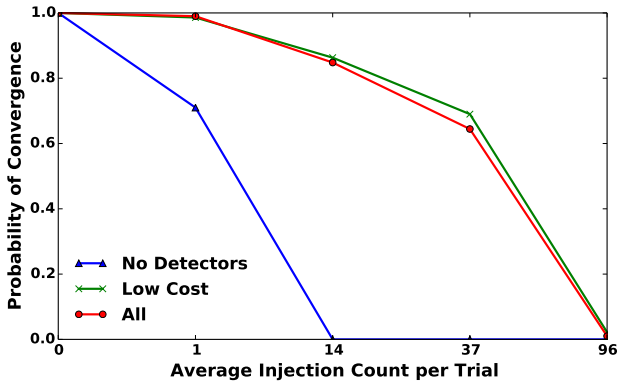

Figure 7. Convergence of AMG with multilevel recovery scheme.

Even with an average of 37 faults injected during each trial, our resilient version of AMG converges in over 69% of trials for *Low Cost* and in over 62% of trials for *All*. Again we see the duality of the energy check. It is able to detect more SDC, even those that do not significantly affect convergence. However, after this point in the testing the probability of convergence decreases dramatically. This is because the solver is making little progress due to high amounts of recovery and because the segfaults are emerging in non-idempotent routines. Indeed, 54% of trials of the *Low Cost* configuration and 75% of the *All* trials terminate due to segfaults in non-idempotent regions. This suggests that if the code were restructured with more idempotent regions, convergence rates would improve. For the remaining faults, when a SDC is detected we attempt to recover, but as we are in the recovery process we suffer more faults that require a restart allowing almost no forward progress.

CONCLUSIONS

Through the use of a combination of application specific detectors we are able to detect SDC that significantly impact convergence. With SDC detected our proposed multilevel recovery scheme is able to recover and converge with a high probability even for a high number of faults. Going forward, fault consideration is going to become a driving factor in design of long running and large scale software. With the AMG augmentations presented and evaluated in this paper, AMG has shown it is capable of being used in faulty environments. Moreover, the residual check and the local segfault recovery scheme are general and likely to be valuable in other numerical libraries.

ACKNOWLEDGMENTS

This work is sponsored by the United States Air Force Office of Scientific Research under grant FA9550–12–1–0478. This work is supported by the Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract DE–AC02–06CH11307. This research is part of the Blue Waters sustained-petascale

computing project, which is supported by the National Science Foundation (awards OCI–0725070 and ACI–1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

REFERENCES

1. Agullo, E., Giraud, L., Guermouche, A., Roman, J., and Zounon, M. Towards resilient parallel linear Krylov solvers: recover-restart strategies. Rapport de recherche RR-8324, INRIA, July 2013.
2. Anfinson, C. J., and Luk, F. T. A linear algebraic model of algorithm-based fault tolerance. *IEEE Trans. Computers 37*, 12 (1988), 1599–1604.
3. Borkar, S. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro 25*, 6 (Nov. 2005), 10–16.
4. Calhoun, J., Olson, L., and Snir, M. FlipIt: An LLVM based fault injector for HPC. In *Proceedings of the 20th International Euro-Par Conference on Parallel Processing (Euro-Par '14)* (2014).
5. Cappello, F., Geist, A., Gropp, W. D., Kale, S., Kramer, B., and Snir, M. Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations 1* (2014), 1–28.
6. Casas, M., de Supinski, B. R., Bronevetsky, G., and Schulz, M. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, ACM (New York, NY, USA, 2012), 91–100.
7. Chen, Z. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, ACM (New York, NY, USA, 2011), 73–84.
8. Chen, Z. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, ACM (New York, NY, USA, 2013), 167–176.
9. de Kruijf, M., Nomura, S., and Sankaralingam, K. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)* (2010).
10. Elliott, J., Hoemmen, M., and Mueller, F. Evaluating the impact of SDC on the GMRES iterative solver. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, IEEE Computer Society (Washington, DC, USA, 2014), 1193–1202.
11. Elliott, J., Mueller, F., Stoyanov, M., and Webster, C. Quantifying the impact of single bit flips on floating point arithmetic. Tech. rep., Oak Ridge National Laboratory, August 2013.
12. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., and Brightwell, R. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, IEEE Computer Society Press (Los Alamitos, CA, USA, 2012), 78:1–78:12.
13. Hargrove, P. H., and Duell, J. C. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series 46*, 1 (2006), 494.
14. Huang, K.-H., and Abraham, J. A. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput. 33*, 6 (June 1984), 518–528.
15. Mishra, A., and Banerjee, P. An algorithm-based error detection scheme for the multigrid method. *IEEE Trans. Comput. 52*, 9 (Sept. 2003), 1089–1099.
16. Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. d. 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, IEEE Computer Society (Washington, DC, USA, 2010), 1–11.
17. Nicolae, B., and Cappello, F. 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, ACM (New York, NY, USA, 2013), 155–166.
18. Ruge, J. W., and Stüben, K. Algebraic multigrid. In *Multigrid methods*, vol. 3 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, PA, 1987, 73–130.
19. Sastry Hari, S. K., Li, M.-L., Ramachandran, P., Choi, B., and Adve, S. V. 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, ACM (New York, NY, USA, 2009), 122–132.
20. Snir, M., Wisniewski, R. W., Abraham, J. A., Adve, S. V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., Chien, A. A., Coteus, P., DeBardeleben, N. A., Diniz, P. C., Engelmann, C., Erez, M., Fazzari, S., Geist, A., Gupta, R., Johnson, F., Krishnamoorthy, S., Leyffer, S., Liberty, D., Mitra, S., Munson, T., Schreiber, R., Stearley, J., and Hensbergen, E. V. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications 28*, 2 (May 2014), 127 171.
21. Sridharan, V., and Liberty, D. A study of dram failures in the field. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, IEEE Computer Society Press (Los Alamitos, CA, USA, 2012), 76:1–76:11.