

Exploiting Data Representation for Fault Tolerance

J. Elliott^{a,b,*}, M. Hoemmen^b, F. Mueller^a

^aNorth Carolina State University, Dept. of Computer Science, Raleigh, N.C.

^bCenter for Computing Research, Sandia National Laboratories, Albuquerque, N.M.

Abstract

Incorrect computer hardware behavior may corrupt intermediate computations in numerical algorithms, possibly resulting in incorrect answers. Prior work models misbehaving hardware by randomly flipping bits in memory. We start by accepting this premise, and present an analytic model for the error introduced by a bit flip in an IEEE 754 floating-point number. We then relate this finding to the linear algebra concepts of normalization and matrix equilibration. In particular, we present a case study illustrating that normalizing both vector inputs of a dot product minimizes the probability of a single bit flip causing a large error in the dot product's result. Furthermore, the absolute error is either less than one or very large, which allows detection of large errors. Then, we apply this to the GMRES iterative solver. We count all possible errors that can be introduced through faults in arithmetic in the computationally intensive orthogonalization phase of GMRES, and show that when the matrix is equilibrated, the absolute error is bounded above by one.

Keywords: Algorithm-based fault tolerance, resilient algorithms, numerical methods

1. Introduction

In the field of high-end computing (HEC) the notion of reliability has tended to focus on keeping thousands of physical nodes operating cooperatively for extended periods of time. As chip manufacturing and power requirements continue to advance, soft errors are becoming more apparent [1]. This implies that reliability research must address the case that the machine does not crash, but that outputs during computation may be silently incorrect. There have been many studies into hardening numerical kernels against soft errors, that is, the researchers attempt to preserve the illusion of a reliable machine by detecting and correcting all soft errors. We take a more analytical approach. Instead of focusing on detection/correction, we seek to study how the data operated on impacts the errors that we can observe given soft errors in data — called silent data corruption (SDC).

The driving motivation behind our work is the uncertainty surrounding the reliability of an

exascale-class machine [2–4]. We attempt to avoid speculation over what hardware may be used in future (or present) HEC deployments, and instead analyze how a single soft error in an IEEE-754 floating-point number behaves. It has already been shown that existing and decommissioned HEC deployments have suffered from SDC [1, 5]. For the prior reasons, we seek to study the link between the data operated on and soft errors. We intentionally perform our research subject to the IEEE 754 specification, which we believe will be used regardless of the architecture. We also restrict our analysis to single bit flips. This gives us a base line from which to draw higher-level conclusions related to multiple bit flips, and lets us isolate the impact of a bit flip.

IEEE 754 both defines the binary *representation* of data, and bounds the *rounding error* committed by arithmetic operations. This work focuses on data representation. The effects of rounding error on numerical algorithms, including those studied in this paper, have been extensively studied (see, e.g., [6]). However, these results generally only apply to *small* errors, such as those resulting from rounding. The error from bit flips can be huge and thus require different methods of analysis, like those presented in this paper.

*Corresponding author

Email addresses: jjellio3@ncsu.edu (J. Elliott),
mhoemme@sandia.gov (M. Hoemmen), mueller@cs.ncsu.edu
(F. Mueller)

2. Related Work

Researchers have approached the problem of SDC in numerical algorithms in various ways. Many take the approach of treating an algorithm as a black box and observing the behavior of these codes when run with soft errors injected. Recently, Howle et al. [7] analyzed the behavior of various Krylov methods and observed the variance in iteration count based on the data structure that experiences the bit flip. Shantharam et al. [8] analyzed how bit flips in a sparse matrix-vector multiply (SpMV) impact the L^2 norm and observed the error as CG is run. Bronevetsky et al. and Sloan et al. [9, 10] analyzed several iterative methods documenting the impact of randomly injected bit flips into specific data structures in the algorithms and evaluated several detection/correction schemes in terms of overhead and accuracy. Exemplifying the concept of black-box analysis, BIFIT [11] characterizes applications based on their vulnerability to bit flips. A similar tool, S-FETI [12] injects soft errors through an emulation layer, allowing arbitrary codes to be run in a faulty virtual environment. Rather than focusing on how to preserve the illusion of a reliable machine or devising a scheme to inject soft errors, we investigate an avenue mostly ignored, which is how the data in the algorithm can be used to mitigate the impact of a bit flip.

Hoemmen and Heroux proposed a radically different approach. Rather than attempt to detect and correct soft errors, they use a “selective reliability” programming model to make the algorithm converge *through* soft errors [13]. Sao and Vuduc showed that reliably restarting iterative solvers enables convergence in the presence of soft errors [14]. In the same vein, Elliott et al. showed that bounding the error introduced in the orthogonalization phase of GMRES lets FT-GMRES converge with minimal impact on time to solution [15]. Boley et al. apply backward error analysis to linear systems in order to distinguish small errors due to rounding from inacceptably large errors due to transient hardware faults [16]. In general, our work complements this line of research. While Elliott, Hoemmen, and Sao have investigated algorithms that can converge through error, we show that in certain numerical kernels the data itself can have a “bounding” effect. For example, coupled with the work of Elliott et al. [15], we improve the likelihood that errors fall within the derived bound.

3. Motivation

Bit flips are the motivation for resilient algorithms. Algorithms run on digital machines, so any corruption is due to a fault at the hardware level or radiation from space. There are many layers and abstractions between what a user sees and what hardware actually does. Caches for data and instructions are an example of these layers.

Many works in the field of algorithmic fault tolerance have tailored methods for a bit flip fault model, e.g., [10, 17–21], but their fault model rarely considers faulty hardware. Instead, researchers simulate hardware faults by injecting bit flips into values before or after an operation. A standard approach, e.g., [9, 14, 22], is to flip one or more bits in the input to an operation and observe the effect on algorithms.

We use this same fault injection methodology, but relate our findings to the analytic model of IEEE-754 floating point representation. Our findings demonstrate the close relationship between data and a fault model. Moreover, our findings illustrate the shortcomings of evaluating algorithms using this type of fault injection. As we will demonstrate, the expected value obtained by sampling is highly dependent on the characteristics of the data operated on. While bit flips are the root cause of errors, it is not clear that we need to assess algorithmic fault tolerance approaches using a bit flip fault methodology.

4. Overview

To explore the relation between data representation and soft errors, we first construct an analytic model of a soft error in an IEEE-754 floating-point scalar, and then extend this to a dot product. We uncover through analysis that the binary pattern of the exponent can be exploited for fault tolerance. We show this graphically via a case study using Monte Carlo sampling of random vectors, and then extend the idea of data scaling to matrices by using sparse matrix equilibration. To demonstrate the feasibility and utility of our work, we analyze the GMRES algorithm and instrument the computationally intensive orthogonalization phase. We count the possible absolute errors that can be introduced via a bit flip in a dot product, and show that scaling data lowers the likelihood of observing large, undetectable errors.

This paper is organized as follows:

1. In § 5, we construct an analytic model of the absolute error for single bit upsets in IEEE-754 floating-point numbers.
2. In § 6, we extend our model of faults in IEEE-754 scalars to vectors of arbitrary values, and present examples of how data scaling impacts the binary representation and absolute error we can observe.
3. In § 7, we sample random dot products and compute the probability of having an error larger than one using Monte Carlo.
4. In § 8, we link data scaling to sparse matrix equilibration, and instrument and evaluate the impact of a soft error in the computationally intensive orthogonalization phase of GMRES.

5. Fault Model

The premise of our work is that a silent, transient bit flip impacts data. Before we can perform any analysis or experimental work, we must define how such a bit flip would impact an algorithm, and how we enforce that the bit flip was transient. To achieve this goal, we build our model around the basic concept that when an algorithm uses data, this translates into some set of operations being performed on the data. Should a bit flip perturb our data, some operation will use a corrupt value, rather than the correct value. The output of this single operation will then contain a tainted value, and this tainted value could cause the solution to be incorrect. Note that a transient bit flip may cause a persistent error in the output depending on how the value is used.

A side benefit of an operation-centric model is that we naturally avoid a pitfall to which arbitrary memory fault injection succumbs, namely that if a bit flip impacts data (or memory) that is never used (read) then this fault *cannot lead to a failure*. Our fault model allows a bit flip to perturb the input to an operation performed on the data, while not persistently tainting the storage of the inputs. This mimics how a transient bit flip would manifest itself, e.g., during ALU activities. As a result, the data that experiences the bit flip need not show signs that it was perturbed. This model allows us to observe the impact of transient flips on the inputs, which results in sticky or persistent error in the result. We then utilize mathematical analysis to model how this persistent error propagates through the algorithm.

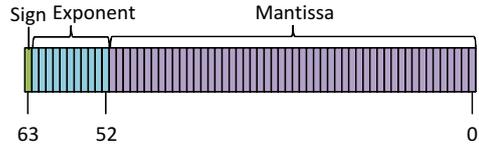


Figure 1: Graphical representation of data layout in the IEEE-754 Binary64 specification.

5.1. Fault Characterization

To derive a fault model we must first understand what a fault is. Since floating-point numbers approximate real numbers and most numerical algorithms use real values, we start from the definition of a real-valued scalar $\gamma \in \mathbb{R}$. The range of *possible* values that γ can take is $\gamma \in [-\infty, +\infty]$. We assume that the IEEE-754 specification for double-precision numbers, called *Binary64*, is used to represent these numbers. This means that γ can take a fixed set of numeric values. The range of $|\gamma|$, excluding 0 and denormalized numbers, is

$$|\gamma| \in [1.0 \times 2^{-1022}, 1.\bar{9} \times 2^{1023}]; \quad (1)$$

where $1.\bar{9}$ indicates the largest possible fractional component, and 1.0 indicates the smallest fractional component.

To approximate real numbers, *Binary64* uses 64 bits, of which 11 are devoted to the exponent, 52 for the fractional component (we refer to as the mantissa), and one bit for the sign. Figure 1 shows how these bits are laid out. In addition to numeric values, Binary64 includes two non-numeric values, Not-a-Number (NaN) and Infinity (Inf), which may be signed to account for infinity and values that result in undefined operations, e.g., division by zero. The range of values in Equation (1) is not continuous and has non-uniform gaps due to the discrete precision, which is a consequence of having a fixed number of bits in the fractional component.

We can further discretize the range of possible values by recognizing that there is a finite number of exponents that are possible given IEEE-754 double precision, e.g.,

$$\gamma \in \{0, \pm\text{Inf}, \pm\text{NaN}, \pm 2^{-1022}x, \pm 2^{-1021}x, \dots, \pm 2^0x, \dots, \pm 2^{1023}x\},$$

where $x \in [1, 2)$ is some fractional value.

Analytically, this is expressed as

$$\gamma = (-1)^{\text{sign}} \left(1 + \sum_{i=0}^{51} b_i 2^{i-52} \right) \times 2^{e-1023}, \quad (2)$$

for IEEE-754 *Binary64*. Note, the specification does not include a sign bit for the exponent. Rather, IEEE floating point numbers utilize a *bias* to allow the exponent to be stored without a sign bit, which we will later exploit for fault-resilience. Another important characteristic that stems from the general approach of expressing numbers in exponential notation is that we can characterize numbers by their order of magnitude. Of particular interest is the following relation:

$$|2^{-1022}| \leq |2^{-1022}x| < \dots < |2^{1023}| \leq |2^{1023}x| \quad (3)$$

This means that we can use the next order of magnitude as an upper bound for errors in the fractional component of a number — which is practically achieved by incrementing the exponent or multiplying by two. We can also analytically model the number of fractional bits that could contribute an error larger than some tolerance, since the error that *could* arise from each mantissa bit is relative to the exponent of the number. This final step is necessary since the fractional term can take values in the range $[1, 2)$, where the left parenthesis indicates that 2 is not a member of this interval. We can also characterize the error that a perturbed sign bit can contribute, and, like the fractional component, this error is relative to the exponent of the number. Suppose the sign is perturbed in a scalar γ , then we have $\tilde{\gamma} = -\gamma$, the absolute error is $|\gamma - \tilde{\gamma}| = |\gamma - (-\gamma)| = 2\gamma$. This means we can bound the error from a sign bit perturbation by incrementing the exponent of the resulting value.

In summary, we have demonstrated that errors in IEEE-754 floating point numbers can be characterized using the exponent of the numbers. This property allows us to reduce the number of bits we need to consider in a fault model, since we know that a large number of errors are bounded by the relatively small set of possible exponents.

5.2. Fault Characteristics of Exponents

In the context of IEEE-754 double precision numbers and silent data corruption, we do not model the exponents directly. Instead, we model the biased exponents, as they are the interesting portion of the *data* that allows us to characterize the errors that the majority of the bits present in the data can produce. For instance, in double precision data we can characterize the errors from 53 of the 64 bits using our approach. This type of fault characterization is impossible if bit flips are injected randomly

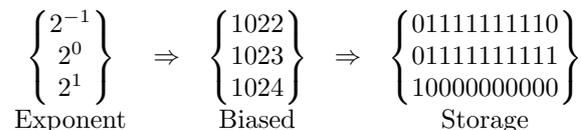


Figure 2: Relation of exponent, IEEE-754 double precision bias, and what data are actually stored.

into the data’s memory, as that approach loses the semantic information that is implicitly present in the data.

The Binary64 specification does not store exponents directly, instead it uses a bias of 1023. From § 5.1, this means we can characterize *all* faults in double precision data by analyzing perturbations to the possible biased exponents

$$\{0, 1, 2, \dots, 1023, \dots, 2046\}.$$

Note that zero is not a biased exponent and has a special meaning. In IEEE-754, a zero pattern in the exponent with zeros in the mantissa is used to represent the scalar zero, while a non-zero pattern in the mantissa is used to represent subnormal numbers. We also assume the user does not perform computation on the two non-numeric values NaN and Inf, which are represented using the biased exponent 2047 (all ones). We do include zero in our analysis because it is a valid real number.

Since we are concerned with bit perturbations in the exponent, we express the biased exponents in their binary form, e.g., 11-bit unsigned integers presented in binary. We can further expand Figure 2 to show the potential change to the original exponent should a bit flip occur, which will form the basis for our fault model and analytic models.

In the context of bit flips, we can view a bit flip as adding or subtracting from the biased exponent, which in turn translates to multiplying or dividing the number by some power of two. We model the impact of a bit flip in the exponent as the original scalar being magnified or minimized by a specific power of two. The biased exponent translates to a discrete binary pattern. We consider all single bit flips in this binary pattern and compute all possible perturbed values.

We summarize the absolute error and change in order of magnitude of a scalar given a single bit flip in Table 1. The key observation is the 4th column, that is, the order of magnitude does not increase

Table 1: Bit flip absolute error for a scalar λ represented using IEEE-754 double precision, with λ_{exp} as the exponent 2^x .

Bit Location	Absolute Error	Bit Range (j)	Change in Order of Magnitude
Mantissa	$ \lambda_{\text{exp}}(1 + 2^{j-52}) $	$0, \dots, 51$	0
Exponent $_{1 \rightarrow 0}$	$ \lambda(1 - 2^{-2^j}) $	$0, \dots, 10$ and $\text{bit}_{j+52} = 1$	-2^j
Exponent $_{0 \rightarrow 1}$	$ \lambda(1 - 2^{2^j}) $	$0, \dots, 10$ and $\text{bit}_{j+52} = 0$	$+2^j$
Sign	$ 2\lambda $		1

given flips in the mantissa and *always* decreases if the bit being flipped is a one. Consider Figure 2, suppose a bit flip impacts the least significant bit in 2^{-1} (a zero). This will clearly increase the order of magnitude, but the resulting perturbed exponent can at worst be 2^0 . The only bit flip that can introduce an error larger than 2^0 is the case that the most significant bit (0) is flipped. In this case, the error introduced is very large, e.g., 2^{-1} becomes 2^{+1023} .

The goal of this work is to uncover the characteristics of the data being operated on, and then exploit these properties to improve fault tolerance given a bit flip. This section has explored the impact of a bit flip and shown that values in the range $[0, 2^0]$ have a binary pattern (thanks to the bias) that will minimize the error introduced most of the time. For values less than 2^1 , bit flips in any bit position $0, \dots, 9$ will never produce an error larger than 1. We now extend these findings to dot products, and then show how we can exploit the concept of all values being less than or equal to one.

5.3. Operation Centric Fault Model

We now describe a realization of our fault model that describes the error that could be injected if an operation in a dot product experiences a single bit upset. We choose the dot product because it is a common operation, and because we will use this model in § 8 to model the worst-case errors that could be injected into a phase of the GMRES algorithm.

Given two real-valued n -dimensional vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, the dot product is defined as

$$c = \sum_{i=1}^n c_i, \text{ where } c_i = a_i b_i. \quad (4)$$

If we allow a single bit flip to impact the i -th element of the dot product, then we have a perturbed

solution \tilde{c} , which is the result of a perturbation to either a_i , b_i , or c_i . In the context of our fault model, this captures a bit upset impacting the inputs to the multiplication operator, and it captures a bit upset in the intermediate value, c_i , which is the input to the addition operator.

Using Table 1, we have all of the tools necessary to compose an absolute error model for a dot product, i.e., addition is modeled by a fault in a scalar $|\alpha + \beta - (\tilde{\alpha} + \beta)| = |\alpha - \tilde{\alpha}|$. The potential change in order of magnitude is paramount. Consider an exponent flip from $1 \rightarrow 0$. These types of exponent bit flips produce an error that is bounded above by the original magnitude of the result, which can be viewed as “zeroing out” the term if a perturbation occurs. Similar to a perturbed scalar, the mantissa can contribute either no change in the order of magnitude, or in the worst case a bit flip causes a carry, which will increment the order of magnitude by one. The order of magnitude for a sign bit flip is exactly the same as that of a perturbed scalar, which introduces an error one order of magnitude larger than the result. These error models can be thought of as the largest additive error that we can inject into a dot product from a bit flip, e.g.,

$$\tilde{c} = \sum_{i=1}^n a_i b_i + (\text{error term}). \quad (5)$$

In summary, we have composed analytic models for the the absolute error that could be introduced into a dot product. Our models are initially constructed from the IEEE-754 Binary64 model, which we extended to express how a bit upset impacts a singular double precision scalar. We then composed a model for the multiplication operator, and analytically expressed the absolute error. Using the absolute error, we have a model that explains *how wrong* a dot product can be, assuming a bit flip in one of the input vectors or in an intermediate value. Next, we refine these models to construct strict up-

per bounds on the error introduced by a bit flip in a dot product.

5.4. Error Bounds for a Bit Flip in a Dot Product

The models presented in Table 1 make no assumptions about the bits present in the mantissa of the operands. This is problematic if we want to consider *all* possible errors that could be introduced into a dot product. To account for the mantissa, and to create strict upper bounds on the error, we will use the relation presented in Eq. 3. From this relation, we know that $\alpha\beta < 2^{\alpha_{\text{exponent}}+1}2^{\beta_{\text{exponent}}+1}$. We can write this as

$$\alpha\beta < 4\alpha_{\text{exp}}\beta_{\text{exp}}, \quad (6)$$

where $\alpha_{\text{exp}} = 2^{\alpha_{\text{exponent}}}$. Using Eq (6), we are able to account for the mantissa bits, but we can also show that a bit flip in the sign is bounded by Eq. (6). The sign bit introduces an absolute error equivalent to incrementing the exponent of the result

$$\alpha\beta < 2\alpha\beta < 4\alpha_{\text{exp}}\beta_{\text{exp}}, \quad (7)$$

where $2\alpha\beta$ is the potential error introduced should the sign bit be perturbed, which must be smaller than the bound constructed for the mantissa.

By utilizing Eq. (6), we are able to account for all possible mantissas and their potential faults, as well as a perturbation to the sign bit. We will now discuss how to use this model to understand the relationship between the data in an algorithm and the distribution of potential errors that could occur should a bit flip in the data.

6. Fault Model Evaluation

In § 5, we proposed analytic models for errors should a bit flip occur in IEEE-754 double precision data. We now illustrate how data can impact the size of errors that a bit flip can create. Consider the following sample vectors

$$\mathbf{u}_{\text{small}} = \begin{bmatrix} 0.5 \\ 0.25 \end{bmatrix}, \quad \mathbf{v}_{\text{small}} = \begin{bmatrix} 0.25 \\ 0.5 \end{bmatrix}, \quad \text{and}$$

$$\mathbf{u}_{\text{large}} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}, \quad \mathbf{v}_{\text{large}} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}.$$

If we compute the dot product $\lambda = \mathbf{u}_{\text{large}} \cdot \mathbf{v}_{\text{large}}$, we have a finite number of potential errors should a bit flip in the data of $\mathbf{u}_{\text{large}}$, $\mathbf{v}_{\text{large}}$, or in an intermediate value in the summation. We can experience either

$$\tilde{2} = \begin{Bmatrix} 2^2 \\ 2^3 \\ 2^5 \\ 2^9 \\ 2^{17} \\ 2^{33} \\ 2^{65} \\ 2^{129} \\ 2^{257} \\ 2^{513} \\ \text{Zero} \end{Bmatrix}, \quad \tilde{4} = \begin{Bmatrix} 2^1 \\ 2^4 \\ 2^6 \\ 2^{10} \\ 2^{18} \\ 2^{34} \\ 2^{66} \\ 2^{130} \\ 2^{258} \\ 2^{514} \\ 2^{-1020} \end{Bmatrix}, \quad \tilde{8} = \begin{Bmatrix} 2^4 \\ 2^1 \\ 2^7 \\ 2^{11} \\ 2^{19} \\ 2^{35} \\ 2^{67} \\ 2^{131} \\ 2^{259} \\ 2^{515} \\ 2^{-1018} \end{Bmatrix}$$

Figure 3: Example of perturbed values greater than one.

$$\widetilde{0.5} = \begin{Bmatrix} 2^0 \\ 2^{-3} \\ 2^{-5} \\ 2^{-9} \\ 2^{-17} \\ 2^{-33} \\ 2^{-65} \\ 2^{-129} \\ 2^{-257} \\ 2^{-513} \\ 2^{1022} \end{Bmatrix}, \quad \widetilde{0.25} = \begin{Bmatrix} 2^{-3} \\ 2^0 \\ 2^{-6} \\ 2^{-10} \\ 2^{-18} \\ 2^{-34} \\ 2^{-66} \\ 2^{-130} \\ 2^{-258} \\ 2^{-514} \\ 2^{1019} \end{Bmatrix}, \quad \widetilde{0.125} = \begin{Bmatrix} 2^{-2} \\ 2^{-1} \\ 2^{-7} \\ 2^{-11} \\ 2^{-19} \\ 2^{-35} \\ 2^{-67} \\ 2^{-131} \\ 2^{-259} \\ 2^{-515} \\ 2^{1017} \end{Bmatrix}$$

Figure 4: Example of perturbed values less than one.

$\tilde{2} \times 4 + 4 \times 2$, $2 \times \tilde{4} + 4 \times 2$, or $\tilde{8} + 8$. For clarity we state what the perturbed values could be in Figure 3 and Figure 4.

By inspection it is clear that substituting any of the above perturbed scalars into the dot product will produce an absolute error greater than one in all cases, and in the event one chooses to substitute the near zero perturbed values, the absolute error of the dot product still has magnitude 8, e.g., $|16 - (0 + 8)|$.

Alternatively, consider the vectors $\mathbf{u}_{\text{small}}$ and $\mathbf{v}_{\text{small}}$. Computing the dot product, $\lambda = \mathbf{u}_{\text{small}} \cdot \mathbf{v}_{\text{small}} = 0.25$, the possible values to perturb are: $\widetilde{0.5}$, $\widetilde{0.25}$, and $\widetilde{0.125}$. We construct these from our model of a perturbed scalar, and present the perturbed variants in Figure 4. By inspection, $\widetilde{0.5}$ can contribute an absolute error to

the dot product larger than one only once, e.g., $|0.25 - (2^{1022} \times 0.25 + 0.125)|$. Likewise, $\widetilde{0.25}$ and $\widetilde{0.125}$ can perturb the result of the dot product with error greater than one only once, and for all 3 cases the perturbation will change the result by hundreds of orders of magnitude.

6.1. Faults in the Mantissa or Sign

The error generated by the mantissa or sign bits is relative to the exponent of the number that the flip occurred in. If the exponent is larger than one, then clearly the mantissa or sign bits can generate an error larger than one. Alternatively, if the values all are less than one, then mantissa errors will produce errors less than one because $2^{-1} \times 1.x \leq 2^0$. The errors from the sign bit cannot exceed 2 since $2 \times 2^{-1} \times 1.x < 2^1$.

It is reasonable to consider that the mantissa generates a carry, as discussed in § 5.4. To account for this we construct a strict upper bound by incrementing the exponent of each element of the vectors analyzed similar to Eq. (6). For example,

$$\mathbf{u}_{\text{original}} = \begin{bmatrix} 2.12332 \\ 1.24568 \end{bmatrix} \Rightarrow \mathbf{u}_{\text{upper bound}} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}. \quad (8)$$

We then can evaluate our models on these vectors to determine a strict upper bound on the errors we can experience in a dot product.

6.2. Modeling Large Vectors

We have shown how to exhaustively examine each element in a vector, and from this analysis we can determine precisely which absolute errors we could experience. Given large vectors, where the dimension n may have millions or billions of elements, exhaustively searching each element would be time consuming, but it would also be a waste of time. As stated previously, there is a discrete number of exponents supported by the IEEE-754 Binary64 specification. As we have previously shown, the exponent characterizes the faults we can observe, so we only need to consider the 2046 possible biased exponents and the special case of zero. The perturbations that are possible can be determined independent of concrete data values, e.g., we can precompute the perturbations and absolute error because we know the relation stated in Eq. (3) and Eq. (6).

To analyze arbitrarily large vectors, we construct a lookup table for the absolute error in whatever operation we choose to model (we have chosen products and addition). The table size is 2047×2047 ,

and allows us to consider the error introduced by performing an operation on two exponents, which will map to a unique i, j location.

For example, consider the vectors

$$\mathbf{u} = \begin{bmatrix} 1.0 \\ 1.2 \\ 8.0 \\ 0.125 \end{bmatrix}, \text{ and } \mathbf{v} = \begin{bmatrix} 0.125 \\ 0.125001 \\ 0.125002 \\ 1.0 \end{bmatrix}. \quad (9)$$

We first extract the biased exponents from the vectors

$$\mathbf{u} \Rightarrow \mathbf{u}_{\text{exponent}} = \begin{bmatrix} 2^0 \times 1.0 \\ 2^0 \times 1.x \\ 2^3 \times 1.0 \\ 2^{-3} \times 1.0 \end{bmatrix} \Rightarrow \mathbf{u}_{\text{biased}} = \begin{bmatrix} 1023 \\ 1023 \\ 1026 \\ 1020 \end{bmatrix} \quad (10)$$

Now, we determine an interval of possible values, and account for the mantissa values that may have been truncated

$$u_i \in [1020, 1026] \subseteq [1020, 1027] \text{ for } i = 1, \dots, 4. \quad (11)$$

The range of biased exponents $[1020, 1027]$ will contain all possible values that the original vector contained, and include one value that was larger than any in the vector, the number corresponding to the biased exponent 1027. Similarly, we can compute the interval for \mathbf{v}

$$\mathbf{v} = \begin{bmatrix} 0.125 \\ 0.125001 \\ 0.125002 \\ 0.25 \end{bmatrix} \Rightarrow \begin{bmatrix} 2^{-3} \times 1.0 \\ 2^{-3} \times 1.x \\ 2^{-3} \times 1.x \\ 2^{-2} \times 1.0 \end{bmatrix} \Rightarrow \begin{bmatrix} 1020 \\ 1020 \\ 1020 \\ 1021 \end{bmatrix}, \quad (12)$$

which leads to the interval we consider errors

$$v_i \in [1020, 1021] \subseteq [1020, 1022] \text{ for } i = 1, \dots, 4. \quad (13)$$

To allow us to analyze intervals efficiently, we create a lookup table, where each entry computes the relevant perturbations and absolute errors for the operations being modeled. In the case of multiplication, the table has symmetry because multiplication is commutative. In practice, computing the full table $(0, \dots, 2046)$ is simple and allows the modeling of errors for arbitrary vectors.

A caveat of the above approach is that we must know the range of values that the vector contains. This can be achieved by directly computing the min and max values for each vector. Alternatively, an approximate range can be determined if the “length” of the vector is known, e.g., the two-norm

or if we know that the data is normalized, i.e., the two-norm is one. One weakness to the proposed approach is that we do not consider a flip in the accumulating sum, which we have left to future work. We also leave to future work analysis that shows how many of these modeled errors lie within the rounding error bound for pairwise sums.

6.3. Summary

We have shown that the range of values used in the dot product has a direct impact on the size of the errors that can be observed. A general rule in floating point algorithms has been to perform operations on numbers as close to the same magnitude as possible, as doing so minimizes the loss of precision. We have now shown that following this rule-of-thumb also gives the benefit of making bit upsets generate a relatively small error when the numbers are no larger than one. Next we present a motivating case study that focuses exclusively on dot products, and then in § 8 we show how to exploit data scaling in an iterative solver.

7. Case Study: Vector Dot Products

To begin our investigation, we assess the susceptibility of the dot product of two N -dimensional vectors to a silent bit flip in arithmetic. We make this choice since many linear algebra operations can be decomposed into dot products, for example, Gram-Schmidt orthogonalization or matrix-vector multiplications.

7.1. Computational Challenges

Given a single double-precision number, there are 64 bits that may be flipped. Extending this to an N -dimensional vector, we have $64N$ bits that are candidates for flipping. Accounting for different numbers results in a very large sample space, and, therefore, we utilized a hybrid CPU-GPU cluster and created a parallel code that farms out specific Monte Carlo trials to various nodes. In this context, a trial consists of creating two vectors, which is discussed in the follow section, and then determining pass/fail statistics given a bit flip on the input to the dot-product kernel. We utilized the BLAS `ddot()` routine, and aggregated the output for post-processing in MATLAB. Ensuring a sampling error of less than 0.001, which is discussed in Section 7.5, required nearly 400,000 CPU hours parallelized over the processors of a 1700 core cluster of

AMD 6128 Opterons. The large search space coupled with ensuring statistically significant results highlights why an analytic approach is not only more efficient, but may be required for more advanced methods and data structures, e.g., matrices and linear solvers.

7.2. Monte Carlo Sampling

We next develop a better understanding of how vector magnitudes impact the expected absolute error should a bit perturb a dot product. To conduct Monte Carlo sampling, we must first determine a mechanism for tallying success, and we must define *success and failure*.

- **Vector Creation**

- 1) Mantissa generated randomly using C `stdlib rand()`.
- 2) For each vector, we fix each element's magnitude to the bit pattern 2^{-50} to 2^{50} (101 bit patterns). This corresponds to the base ten numbers in the range 8.8×10^{-16} to $1.1 \times 10^{+15}$. This range was chosen because 2^{-50} roughly is the machine precision. The numbers in this range are utilizing the highest *precision* that Binary64 offers.

- **Sample definition and Error Calculation**

- 1) A random sample is defined by generating two random N length vectors and computing the absolute error considering all possible $2 \times 64 \times N$ bit flips.
- 2) A tally is defined by failure, which we define to be any absolute error that is greater than 1.
- 3) An empirical estimate of the expected absolute error is computed by dividing the number of failures by the number of bits considered times the vector length times 2 times the number of random samples (M) taken for a given magnitude combination, i.e., $failures / (2 \times 64 \times N \times M)$.

- **Visualization**

- 1) To visualize the expected absolute error, we construct tallies for each magnitude combination, i.e., 101×101 unique combinations, and each combination is sampled M times.
- 2) We summarize this information in a surface plot, where the x- and y-axes denote the \log_2 of the relative magnitude of the vector \mathbf{u} and \mathbf{v} , respectively. The height of the surface plot indicates the probability of seeing an absolute error larger than 1.

Figure 5 presents a surface plot as described in the Visualization bullet. To interpret this graph,

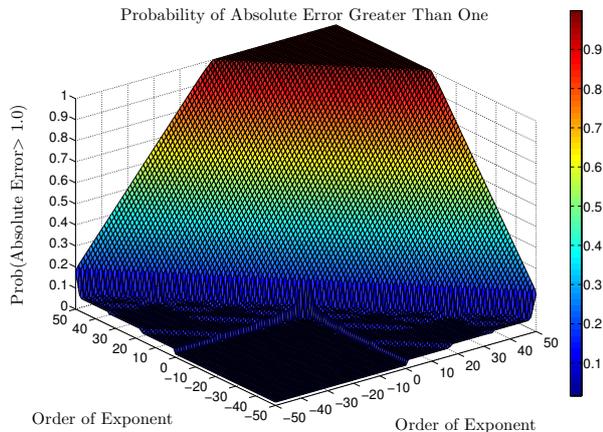


Figure 5: Probability of observing an absolute error larger than 1, given dot products with vectors of various magnitudes. Failure is defined to be an absolute error larger than 1.

the x-axis indicates the magnitude that all elements of the vector \mathbf{u} were forced to have while the mantissa was randomly generated. Likewise, the y-axis indicates the magnitude that all elements of the vector \mathbf{v} were forced to have. Each x-y intersection represents 1,000,000 random vector samples, where the dot product was computed and failures tallied. The height of the surface at an (x, y) location indicates the probability of observing an absolute error larger than 1 given a single bit flip. From this surface, one may immediately recognize the unusual structure of this graph: When both vectors have magnitudes larger than 2^0 , the probability of failure is noticeably higher; yet, when both vectors have magnitudes less than or equal to 2^0 , the probability of failure is approaching zero.

The key finding presented in Figure 5 is that when we operate on vectors that are normalized, e.g., values in the range $[0, 1]$, we have a very low probability of seeing a large error should a bit flip occur. The lowest probability, i.e., the flat region in the quadrant $[0, -50] \times [0, -50]$, is precisely $\text{Prob}(\text{Abs Error} > 1) = 0.015625$, which is $1/64$. The single bit that can introduce an absolute error larger than one is the most significant exponent bit. Also, should the most significant exponent bit flip, the error is quite large and can be detected [15].

7.3. Scaling and Error

We defined a failure to be an absolute error larger than one. This is not an arbitrary value. It can

be shown that the expected absolute error behaves in two asymptotically different ways given single bit flips in the exponent of IEEE-754 scalars [23, Figure 9]. We have empirically discovered this behavior, which motivates the theoretical analyses presented in [23]. Operating on normalized values has a substantial impact on the absolute error. Beyond the scope of this work, it can also be shown that the relative error will be less than one most of the time, because mantissa errors are considerably more likely than exponent or sign errors. Furthermore, the relative error in the exponent will be small approximately 50% of the time, if bit flips from one to zero and zero to one are equally likely. This work represents an empirical study of bit flips in double precision, following an error injection methodology that is consistent with other resilience works, i.e., flip a bit in the input and observe the effect on the output.

7.4. Per Bit Analysis of Surface Plot

To better understand the structure of the surface plot, we take two slices of the surface and look at the per-bit probability of a failure (Figures 6 and 7). The slices chosen feature dot products of vectors with *similar* relative magnitudes and dot products of vectors of many magnitudes (the x-axis) with a vector that contains magnitudes up to 2^3 . Intuitively, these figures slice from the back-most corner of Figure 5 to the front for similar magnitudes (Figure 6), and they slice from the left to right for Figure 7.

We have shown why this shape should be expected in § 5.2, and in the example presented in § 6. This feature is an artifact of how the exponent is implemented in the IEEE-754 specification, i.e., a biased exponent. The lowest probability presented in the surface plot is $0.015625 = 1/64$. We graphically show this in Figure 6, where one can see that bit #62 (2^{nd} from the top) is the only bit that can contribute a large error. We also show that the sign and mantissa bits cannot introduce a large error when values are in the range $[0, 1]$.

Conversely, Figure 7 shows that when mixing large and small values, we expect to see large errors for faults. The green shading in Figure 7 (upper left quadrant) indicates a roughly 50% chance that we see an absolute error larger than one. The reason for this is that values larger than 2 have a binary pattern that introduces a large error most of the time (recall Figure 3). The increased likelihood of a large error from the large numbers, coupled

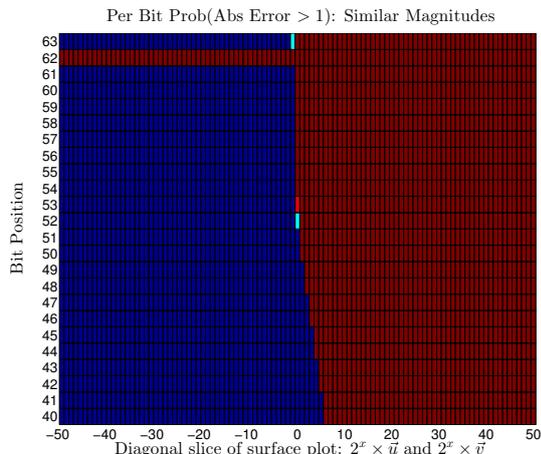


Figure 6: Probability of observing an absolute error larger than 1, with exponents of similar magnitudes (back corner to front corner of Figure 5)

with the low chance from small numbers, creates a scenario where it is equally likely to see both large and small absolute errors. The more we deviate from operating on numbers in the range $[0, 1]$, the closer we get to having a 50/50 chance of seeing a large error.

7.5. Comparison of Analytic Model and Monte Carlo Sampling

In Figure 8, we compare the error observed while performing Monte Carlo sampling with the expected error computed from our model. We sampled up to $M = 1$ million random vectors per data point, which implies a Monte Carlo error of $error_{MC} = 1/\sqrt{M} \approx 0.001$. We observe a perfect fit, which is to be expected because we have analytically shown that the exponent bits dictate the size of the absolute error we will observe. Even with random sign and mantissa bits evaluated, we see that the likelihood of experiencing a large error is entirely determined by the exponent bits.

8. Matrices and Iterative Solvers

Having recognized that dot products on numbers less than one can produce errors less than one, we will relate this idea to matrix equilibration. We then provide an example of how to use this concept in a sparse iterative solver (GMRES), while exhaustively counting the possible errors that can be introduced.

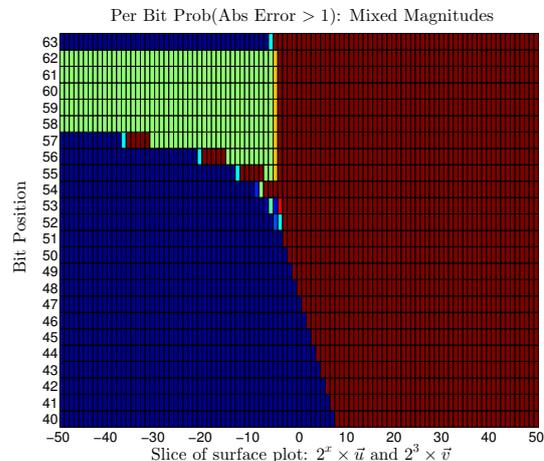


Figure 7: Probability of observing an absolute error larger than 1, with exponents of mixed magnitudes (magnitude 2^3 sliced front to back of Figure 5)

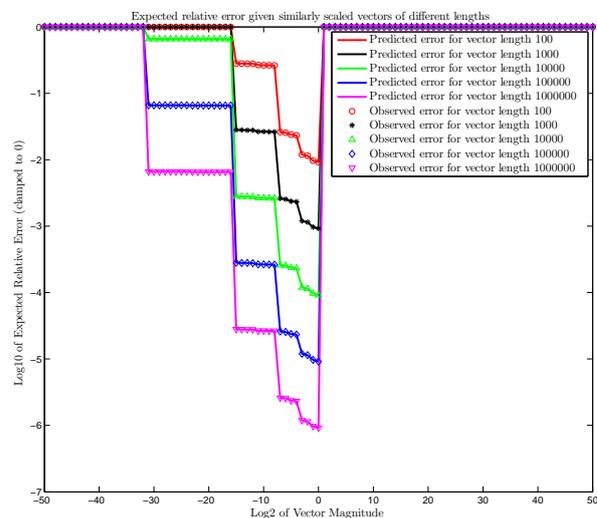


Figure 8: Comparison of observed error caused by a flip in the exponent, excluding the most significant bit, for sampled vector sizes having similar relative magnitudes.

8.1. Matrix Equilibration

The idea of scaled vectors is analogous to vector normalization, i.e., $\|\mathbf{u}\|_2 = 1$. Applied to matrices in the context of solving linear systems, scaling takes the form of *matrix equilibration*: for a matrix \mathbf{A} , scale the rows and columns such that $\|\mathbf{A}\|_\infty = 1$. Scaling can also be performed before a matrix is created. For example, the equations leading to the matrix can be scaled prior to assembling a matrix. To scale a sparse matrix after its creation, we use a sparse matrix implementation of LAPACK’s equilibration routine DGEEQU [24]. Equilibration does not cause fill, i.e., it will not increase the number of non-zeros. In general, equilibrating a matrix is only beneficial, but equilibration may not be practical in all cases, e.g., see [25, p. 126].

8.2. GMRES

The Generalized Minimum Residual method (GMRES) of Saad and Schultz [26] is a Krylov subspace method for solving large, sparse, possibly non-symmetric linear systems $\mathbf{Ax} = \mathbf{b}$. GMRES is based on the Arnoldi process [27], which uses orthogonal projections and basis vectors normalized to length one. Arnoldi and GMRES relate to this work because the orthogonalization phase of Arnoldi is often Modified Gram-Schmidt or Classical Gram-Schmidt, which are dot product heavy kernels.

We present the GMRES algorithm in Algorithm 1. The Arnoldi process is expressed on Lines 3–14 in Algorithm 1. At its core is the Modified Gram-Schmidt (MGS) process, which constructs a vector orthogonal to all previous basis vectors \mathbf{q}_i . The MGS process begins on Line 5 and completes on Line 8. We now describe how we instrument the orthogonalization phase and count the absolute errors that *could* be injected.

8.3. Instrumentation and Evaluation

To demonstrate the benefit of data scaling we have chosen 2 test matrices. We instrument the code and for each dot product in the orthogonalization phase we determine an interval that describes the range of values possible in the vectors. Using our fault model, we compute the absolute errors that are possible. We know the basis vectors (\mathbf{q}_i) are normalized, hence the intervals for the values in the vectors are $[0,1]$. We compute the min and max for the unknown vector \mathbf{v} , and this determines the interval for the values in \mathbf{v} . We use the intervals

Algorithm 1 GMRES

Input: Linear system $\mathbf{Ax} = \mathbf{b}$ and initial guess \mathbf{x}_0
Output: Approximate solution \mathbf{x}_m for some $m \geq 0$

```

1:  $\mathbf{r}_0 := \mathbf{b} - \mathbf{Ax}_0$   $\triangleright$  Initial residual vector
2:  $\beta := \|\mathbf{r}_0\|_2$ ,  $\mathbf{q}_1 := \mathbf{r}_0/\beta$ 
3: for  $j = 1, 2, \dots$  until convergence do
4:    $\mathbf{v}_{j+1} := \mathbf{A}\mathbf{q}_j$   $\triangleright$  Apply the matrix  $A$ 
5:   for  $i = 1, 2, \dots, j$  do  $\triangleright$  Orthogonalize
6:      $h_{i,j} := \mathbf{q}_i \cdot \mathbf{v}_{j+1}$ 
7:      $\mathbf{v}_{j+1} := \mathbf{v}_{j+1} - h_{i,j}\mathbf{q}_i$ 
8:   end for
9:    $h_{j+1,j} := \|\mathbf{v}_{j+1}\|_2$ 
10:  if  $h_{j+1,j} \approx 0$  then
11:    Solution is  $\mathbf{x}_{j-1}$   $\triangleright$  Happy breakdown
12:    return
13:  end if
14:   $\mathbf{q}_{j+1} := \mathbf{v}_{j+1}/h_{j+1,j}$   $\triangleright$  New basis vector
15:   $\mathbf{y}_j := \arg \min_y \|\mathbf{H}(1:j+1, 1:j)\mathbf{y} - \beta\mathbf{e}_1\|_2$ 
16:   $\mathbf{x}_j := \mathbf{x}_0 + [\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_j]\mathbf{y}_j$   $\triangleright$  Solution update
17: end for

```

and our fault model to evaluate all absolute errors that can be introduced from a single bit flip in the input vectors. We classify the absolute error into four classes:

1. Absolute error less than 1.0,
2. Absolute error greater than or equal to 1.0, but less than or equal to $\|\mathbf{v}\|_2$,
3. Absolute error greater $\|\mathbf{v}\|_2$.
4. Error that is non-numeric, e.g., Inf or NaN.

We choose to include the 2nd class of errors due to work by Elliott et al. [15] that demonstrates how to use a norm bound on the Arnoldi process to filter out large errors in orthogonalization. We drop the subscript of \mathbf{v} , because this vector is merely a temporary vector that is normalized to form the next basis vector \mathbf{q}_{j+1} .

Classes 1 and 2 are *undetectable*, while Classes 3 and 4 are detectable. Our goal is to ensure that should a bit flip, the error falls into Classes 1, 3, and 4 while minimizing or eliminating the occurrence of Class 2 errors. We refer to Class 2 errors as the *grey area*, as they are undetectable errors that we consider to be large.

8.3.1. Sample Problems

We have chosen two sample matrices to demonstrate our technique. To ensure reproducibility, we

Table 2: Sample Matrices

Properties	Poisson	CoupCons3D
number of rows	10,000	416,800
number of cols	10,000	416,800
nonzeros	49,600	17,277,420

Table 3: Norms of Sample Matrices

Norm	Poisson Equation		CoupCons3D	
	No Scaling	Scaled	No Scaling	Scaled
$\ \mathbf{A}\ _\infty$	8.0	2.0	1.30×10^6	1.0
$\ \mathbf{A}\ _2$	7.999	1.999	1.20×10^6	1.0
$\ \mathbf{A}\ _F$	446	112	2.75×10^6	291

did not create any of these matrices from scratch, rather we used readily available matrices. The first matrix arises from a second-order centered finite difference discretization of the Poisson equation. We generated this matrix using MATLAB’s built-in Gallery functionality. The second matrix, CoupCons3D, presents a more realistic linear system. It comes from the University of Florida Sparse Matrix Collection [28] and arises from a fully coupled poroelastic problem. The matrix is symmetric in pattern, but not symmetric in values. It is also fairly large, and has explicitly stored zero values. The matrix is poorly scaled, with a mixture of large and small values. We have summarized the characteristics of each matrix in Table 2.

We now scale the Poisson and CoupCons3D matrices and right-hand side vectors such that they are equilibrated. Table 3 summarizes the norms for each of our test matrices. We use the infinity norm ($\|A\|_\infty \approx 1$) to measure whether a matrix is well scaled. The Poisson matrix has infinity norm not much larger than one, while the CoupCons3D matrix is inherently poorly scaled.

8.4. Results

We ran Algorithm 1 for 1000 total iterations, using a restart value of 25. By instrumenting the code, we determined the numerical range of values each vector contained, and then computed the possible absolute error that a bit flip could introduce. We classified the absolute error according to § 8.3, and counted each class of errors for the duration of the algorithm.

Table 4 summarizes the results of evaluating all possible bit flips in the data used in Arnoldi process of GMRES. For each matrix, we exhaustively analyze the data used in each iteration of GMRES, e.g., the mat-vec with a normalized vector and the following dot products used in the Gram-Schmidt orthogonalization. At each iteration of the Gram-Schmidt process we analyze the vectors used and determine the possible errors that could be introduced due to a bit flip in the data. We aggregate this information and classify the errors into four classes.

A large proportion of the absolute errors possible in orthogonalization fall into Class 1 (undetectable and small). We can explain this distribution given that the vectors \mathbf{q}_i are normalized (a side effect of GMRES being derived from the Arnoldi process). Given normalized vectors, we know that of all the dot products in Gram-Schmidt orthogonalization, at least one of the vectors has data in the interval $[0, 1]$. We previously established that the interval $[0, 1]$ aids in minimizing absolute error if a bit perturbs a dot product. Now, we show how equilibrating the input matrices can assist in forcing the non-normalized vector (\mathbf{v}_{j+1}) as close as possible to being in the normalized interval.

The Poisson matrix has relatively good scaling, but still sees benefit from equilibration. That is, the percentage of absolute errors less than one is already high (greater than 90%), but the undetectable errors that lie in the range $[1, \|\mathbf{A}\|_2 \|\mathbf{v}\|_2]$ decrease from 0.066% to 0.015%. Likewise, even the large errors are decreased from 0.348% to 0.047%. The likelihood of seeing a non-numeric value increases, which is beneficial as well, as these errors are detectable.

The CoupCons3D matrix, which has poor scaling benefits greatly from equilibrating. By scaling the matrix, we decrease the percentage of undetectable errors from 9.840% to 0.023%, we also decrease the large detectable errors from 6.993% to 0.120%. Similar to the Poisson matrix, equilibrating results in roughly 9% of errors potentially being non-numeric.

Our results show that scaling tends to produce a distribution of absolute error that is roughly 91% less than or equal to one, while 9% are non-numeric. This is expected when *most* of the numbers are near one. Flipping the most significant exponent bit produces 1111111111, which will generate a non-numeric value. Similarly, the 10 remaining exponent bits will produce an error less than one —

Table 4: Exhaustive analysis of all possible bit flip absolute errors originating from data used in GMRES’s Arnoldi process.

Error Class	Poisson		CoupCons3D	
	No Scaling	Equilibrated	No Scaling	Equilibrated
Absolute error ≤ 1	90.623%	90.847%	76.210%	90.836%
$1 < \text{Absolute error} \leq \ \mathbf{v}\ _2$	0.066%	0.015%	9.840%	0.023%
$\ \mathbf{v}\ _2 < \text{Absolute error}$	0.348%	0.047%	6.993%	0.120%
Non-numeric	8.963%	9.091%	6.957%	9.021%

that is, $1/11 \approx 9\%$ and $10/11 \approx 91\%$. As previously discussed, the mantissa errors are determined entirely by the exponent bits.

9. Conclusion

Our results indicate a clear benefit to good scaling. We have shown that a widely used numerical method (the Arnoldi process coupled with Gram-Schmidt orthogonalization) inherently minimizes absolute error in dot products. Furthermore, standard matrix equilibration algorithms can be used to scale input matrices, which further enhance the inherent robustness of the Arnoldi process. We demonstrated our theoretical finding experimentally by instrumenting the GMRES iterative solver, which is based on the Arnoldi process.

We cannot enforce that data are always normalized. Some linear systems may be inherently poorly scaled, or it may be impractical to equilibrate them. We *can* advocate that scaling, while typically used to improve numerical stability and reduce the loss of precision, can also benefit fault resilience. We have shown that this result has broad applicability, because many iterative solvers are based on orthogonal projections using normalized vectors, i.e., they create an orthonormal basis. While this work does not propose an end-to-end solution to soft errors, it does indicate that data scaling can help mitigate the impact of such errors should they occur.

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.

References

- [1] S. Michalak, A. Dubois, C. Storlie, H. Quinn, W. Rust, D. DuBois, D. Modl, A. Manuzzato, S. Blanchard, Assessment of the impact of cosmic-ray-induced neutrons on hardware in the Roadrunner supercomputer, *Device and Materials Reliability*, IEEE Transactions on 12 (2) (2012) 445–454. doi:10.1109/TDMR.2012.2192736.
- [2] E. N. M. Elnozahy, R. Bianchini, T. El-Ghazawi, A. Fox, F. Godfrey, A. Hoisie, K. McKinley, R. Melhem, J. S. Plank, P. Ranganathan, J. Simons, System resilience at extreme scale, Tech. rep., Defense Advanced Research Project Agency (DARPA) (2008). URL <http://institutes.lanl.gov/resilience/docs/IBMMootazWhitePaperSystemResilience.pdf>
- [3] P. Kogge, et al., ExaScale computing study: Technology challenges in achieving exascale systems, Tech. rep., Defense Advanced Research Project Agency, Information Processing Techniques Office (2008). URL http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf
- [4] F. Cappello, G. A. A. Geist, W. D. B. Gropp, L. V. S. Kale, W. T. C. B. Kramer, M. Snir, Toward exascale resilience, Tech. Rep. TR-JLPC-09-01, University of Illinois at Urbana-Champaign (UIUC) - Institut National de Recherche en Informatique et en Automatique (INRIA) Joint Laboratory on PetaScale Computing (Jun. 2009). URL <http://institutes.lanl.gov/resilience/docs/TowardExascaleResilience.pdf>
- [5] I. S. Haque, V. S. Pande, Hard data on soft errors: A large-scale assessment of real-world error rates in GPGPU, in: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID ’10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 691–696. doi:<http://dx.doi.org/10.1109/CCGRID.2010.84>. URL <http://dx.doi.org/10.1109/CCGRID.2010.84>
- [6] C. C. Paige, M. Rozložník, Z. Strakoš, Modified Gram-Schmidt (MGS), least squares, and backward stability

- of MGS-GMRES, *SIAM J. Matrix Anal. Appl.* 28 (1) (2006) 264–284.
- [7] V. Howle, P. Hough, M. Heroux, E. Durant, Soft errors in linear solvers as integrated components of a simulation, invited talk at the Copper Mountain Conference on Iterative Methods (Apr. 2010).
- [8] M. Shantharam, S. Srinivasmurthy, P. Raghavan, Characterizing the impact of soft errors on iterative methods in scientific computing, in: Proceedings of the 25th International Conference on Supercomputing, ICS '11, ACM, New York, NY, USA, 2011, pp. 152–161. doi:10.1145/1995896.1995922.
- [9] G. Bronevetsky, B. de Supinski, Soft error vulnerability of iterative linear algebra methods, in: Proceedings of the 22nd Annual International Conference on Supercomputing, ICS '08, ACM, New York, NY, USA, 2008, pp. 155–164. doi:10.1145/1375527.1375552.
- [10] J. Sloan, R. Kumar, G. Bronevetsky, Algorithmic approaches to low overhead fault detection for sparse linear algebra, in: Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Washington, DC, USA, 2012, pp. 1–12. URL <http://dl.acm.org/citation.cfm?id=2354410.2355166>
- [11] D. Li, J. Vetter, W. Yu, Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool, in: Supercomputing, 2012.
- [12] Q. Guan, N. DeBardleben, S. Blanchard, S. Fu, F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability, in: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, 2014, pp. 1245–1254. doi:10.1109/IPDPS.2014.128.
- [13] P. G. Bridges, K. B. Ferreira, M. A. Heroux, M. Hoemmen, Fault-tolerant linear solvers via selective reliability, ArXiv e-prints Provided by the SAO/NASA Astrophysics Data System. arXiv:1206.1390.
- [14] P. Sao, R. Vuduc, Self-stabilizing iterative solvers, in: Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ACM, New York, NY, USA, 2013, pp. 4:1–4:8. doi:10.1145/2530268.2530272.
- [15] J. Elliott, M. Hoemmen, F. Mueller, Evaluating the impact of SDC on the GMRES iterative solver, in: 28th IEEE International Parallel & Distributed Processing Symposium, Phoenix, USA, 2014.
- [16] D. Boley, G. H. Golub, S. Makar, N. Saxena, E. J. McCluskey, Floating point fault tolerance with backward error assertions, *IEEE Transactions on Computers* 44 (1995) 302–311.
- [17] M. Shantharam, S. Srinivasmurthy, P. Raghavan, Fault tolerant preconditioned conjugate gradient for sparse linear system solution, in: Proceedings of the 26th ACM International Conference on Supercomputing, ICS '12, ACM, New York, NY, USA, 2012, pp. 69–78. doi:10.1145/2304576.2304588.
- [18] T. Davies, Z. Chen, Correcting soft errors online in LU factorization, in: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing, ACM, New York, NY, USA, 2013, pp. 167–178. doi:10.1145/2462902.2462920.
- [19] T. Davies, C. Karlsson, H. Liu, C. Ding, Z. Chen, High performance LINPACK benchmark: A fault tolerant implementation without checkpointing, in: Proceedings of the 25th Annual International Conference on Supercomputing, 2011, pp. 162–171.
- [20] Y. Chen, Y. Deng, A detailed analysis of communication load balance on BlueGene supercomputer, *Comp. Phys. Comm.* 180 (8) (2009) 1251–1258.
- [21] J. Sloan, A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance, Master's thesis, University of Illinois Urbana-Champaign, Urbana, IL, available online [last accessed 09 Jan 2014]: <https://ideals.illinois.edu/handle/2142/24206> (2011).
- [22] Z. Chen, 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, pp. 167–176. doi:10.1145/2442516.2442533.
- [23] J. Elliott, M. Hoemmen, F. Mueller, Model driven analysis of faulty iee-754 scalars, Tech. Rep. TR 2015-9, Dept. of Computer Science, North Carolina State University (Nov. 2015). URL <http://www4.ncsu.edu/~jjellio3/TR-2015-9.pdf>
- [24] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, 3rd Edition, SIAM, Philadelphia, PA, USA, 1999.
- [25] G. H. Golub, C. F. Van Loan, *Matrix Computations* (3rd Ed.), Johns Hopkins University Press, Baltimore, MD, USA, 1996.
- [26] Y. Saad, M. H. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Stat. Comput.* 7 (3) (1986) 856–869. doi:10.1137/0907058.
- [27] W. E. Arnoldi, The principle of minimized iterations in the solution of the matrix eigenvalue problem, *Quarterly of Applied Mathematics* 9 (1951) 17–29.
- [28] T. A. Davis, Y. Hu, The University of Florida Sparse Matrix Collection, *ACM Transactions on Mathematical Software* 38 (1) (2011) 1:1–1:25.