Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2012) 1–10
Procedia Computer Science

International Conference on Computational Science, ICCS 2012

High Performance Dense Linear System Solver with Resilience to Multiple Soft Errors

Peng Du, Piotr Luszczek, Jack Dongarra

Department of Electrical Engineering and Computer Science

University of Tennessee, Knoxville

Abstract

In the multi-peta-flop era for supercomputers, the number of computing cores is growing exponentially. However, as integrated circuit technology scales below 65 nm, the critical charge required to flip a gate or a memory cell has been dangerously reduced, causing higher cosmic-radiations-induced soft error rate. Soft error threatens computing system by producing silently data corruption which is hard to detect and correct. Current research of soft errors resilience for dense linear solver offers limited capability when facing large scale computing systems, and suffers from both soft error and round-off error due to floating point arithmetic. This work proposes a fault tolerant algorithm that can recover the solution of a dense linear system $Ax = b$ from multiple spatial and temporal soft errors. Experimental results on the Kraken Supercomputer confirm scalable performance of the proposed fault tolerance functionality and negligible overhead in solution recovery.

Keywords: soft error, fault tolerance, multiple errors, dense linear system solver

1. Introduction

Soft errors, normally in the form of bit flips, are events in microelectronic circuit that result in transient modification without permanently damaging the device. They corrupt computed data and produce erroneous results without leaving a trace. High-end computer systems are especially susceptible to such errors due to the ever increasing chip density and shrinking assembly scale. Between 2003 and 2004, the 2048-node ASC Q supercomputer for scientific computing in Los Alamos National Laboratorys experienced failures from extensive soft errors [1]. By comparing the error logs with a radiation experiment conducted in a lab, the cause was soon identified to be the cosmic ray striking the parity-protected cache tag array. A similar incident has also appeared in a commercial computing system from Sun Microsystems that caused outages for many of its customers due to cosmic ray soft errors [2]. These incidents signify that soft error is a real issue that both hardware and software developers must face. Soft error rate (SER) in memory is usually quantified using FIT (failure in time) per MB, 1 FIT is 1 failure per 10^9 operation hours per 10^6 bits. Google has reported between 778 and 25,000 FIT from errors in the DRAMs of their server fleet, an order of magnitude higher than previously expected [3].

Among HPC applications that could benefit from resilience capability to soft errors, dense linear algebra applications such as the HPL benchmark for the TOP500 competition [4] and the AORSA fusion energy simulation program [5], are representative examples. These applications normally involve solving a dense system of equations of the form $Ax = b$ on large scale HPC systems with matrix sizes of A being as large as 500,000. Soft errors that occur

during such long running applications produce incorrect solution with no apparent trace. This lowers productivity by wasting valuable time and power in error tracing with little chance of ever locating the error.

To mitigate the impact of soft errors, hardware and software methods have been developed. For example, ECC (error correcting code) [6] is a commonly used hardware technique which is although limited to a small number of soft errors due to the overhead of encoding/decoding. In software, until now most of the soft error resilience techniques for dense linear solvers are limited to small scale computing installations, such as on systolic arrays, assuming that encoding/decoding can be carried out with exact arithmetic [7, 8]. Unfortunately, this assumption does not hold for today's Pflop/s supercomputer systems. In previous work [9], we have demonstrated the first attempt to take on the challenge of recovering the solution from a dense linear system solver of $Ax = b$ with a single error occurrence in both L and U of the LU factorization. This work further extends that effort into multiple soft errors resilience as a more performance friendly alternative to the complex hardware ECC. The proposed algorithms consider both the temporal and spatial distribution of multiple errors. Temporal soft errors occur at different time, whereas spatial soft errors manifest as simultaneous multiple bit flips in disparate locations. The proposed method may also be extended to other one-sided factorizations for the recovery of linear system solution and factorization matrices.

The rest of the paper is organized as follows. Section 2 introduces an LU based dense linear solver. The impact of soft error on the linear solver is then analyzed and the general workflow of the proposed soft error resilience algorithm is shown in Section 3. Sections 4 and 5 develop the protection method for both the left factor L and right factor U. Section 6 proposes a segmented encoding method to reduce the computational complexity of the error detection and locating. Finally, the recovery algorithm is discussed in Section 7 and the experimental results are shown in Section 8. Related work is described in Section 9 while Section 10 concludes the paper.

2. High Performance Linear System Solver

For dense matrix A, the LU factorization produces $PA = LU$ (or $P = ALU$), where P is a pivoting matrix, and L and U are unit lower and upper triangular matrix respectively. LU factorization is popular for solving system of linear equations. With L and U, the linear system $Ax = b$ is solved by $Ly = b$ and then $Ux = y$. ScaLAPACK[10] implements the right-looking version of LU with partial pivoting based on a block algorithm and 2D block cyclic data distribution. Without loss of generality, this block algorithm is described with an $N \times N$ matrix (or submatrix) A.

Split A into 2×2 blocks with block size NB. A_{11} has size $NB \times NB$, A_{12} is $NB \times (N - NB)$, A_{21} is $(N - NB) \times NB$, and A_{22} is $(N - NB) \times (N - NB)$, which is also known as the "trailing matrix". Decompose A as

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & L_{22} \end{bmatrix}$$

and therefore

$$\begin{cases} \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11} & \rightarrow PDGETF2 \\ A_{12} = L_{11} U_{12} & \rightarrow PDTRSM \\ L_{22} U_{22} = A_{22} - L_{21} U_{12} & \rightarrow PDGEMM \end{cases} \tag{1}$$

This poses as one iteration (step) of the factorization, and pivoting is applied to the left and right side of the current panel. The routines names in the ScaLAPACK LU are listed after "→". For description, we use $\bar{U}$ to represent the area of U_{12} modified by PDTRSM, and $\tilde{U}$ for A_{22} after PDGEMM. Note that the size of $\bar{U}$ and $\tilde{U}$ changes as the LU algorithm proceeds. Block algorithms offer good granularity to benefit from high performance BLAS routines, while 2D block-cyclic distribution ensures scalability with load balancing.

3. Soft Error Resilience Framework

Since soft errors occur at times and locations unknown to the host algorithm, different methodologies are required to provide resilience to different part of the matrix. In this section, the error propagation in LU factorization is discussed and a general workflow of error detection and recovery is given.

3.1. Error Pattern in the Block LU Algorithm

During LU factorization, the left factor L and right factor U have different "dynamics" with regard to the frequency of data change. For L, once a panel is factorized, the resulted data stored under the diagonal comes to the final form without undergoing any further changes except pivoting. Therefore Soft errors occurred in this area do not propagate. This offers an opportunity to use traditional diskless checkpointing method to protect these data. Algorithm based fault tolerance (ABFT) cannot be applied to the panel factorization since otherwise checksum rows for the panel could be moved into data by pivoting which causes erroneous result. In LU, partial pivoting that swaps rows of both L and U is adopted for better stability, but this pivoting operation could break the static feature of the L data as explained in [9], and therefore in this work the pivoting to the factorized L is delayed to the end of factorization. Since soft errors could strike at any moment, checkpointing frequency as high as once per panel factorization is necessary, but this also potentially leads to high performance overhead and therefore checkpointing should be used to the minimum. For example, even though the factorized $\bar{U}$ (result of PDTRSM) also stays static once produced, it can be protected by ABFT checksum with much less overhead.

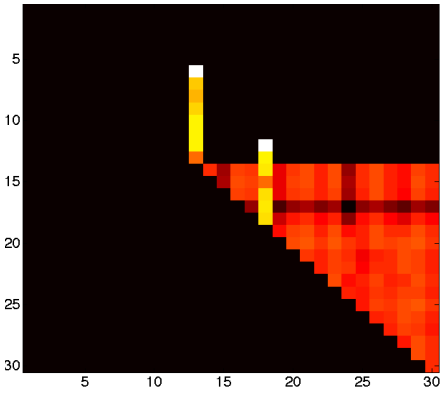


Figure 1: Example of error propagation in the U result of a 30×30 matrix

$\tilde{U}$ differs from L and $\bar{U}$ in that it undergoes changes constantly from trailing matrix update. If soft errors alter data within $\tilde{U}$, and the erroneous data are carried along with computation to update the $\tilde{U}$, even a single-bit soft error could propagate into large area of $\tilde{U}$, let alone multiple errors at different time of the factorization.

Figure 1 shows an example of error propagation in U in a small matrix. Gaussian elimination is applied to a 30×30 matrix. To simplify the illustration, no pivoting nor block algorithm is used. Each step of the Gaussian elimination zeros out elements below the diagonal in one column. The color in the figure is related to the difference between the correct and incorrect upper triangular results. Higher brightness means larger absolute difference in value and black means no difference. During the factorization, Two soft errors were injected at step 1 and 3 at location (6,13) and (12, 18) by adding random values. Since both errors occurred below the row 3, these errors fell in the $\tilde{U}$ area of steps 1 to 3. The two white dots at (6,13) and (12, 18) are the initial injection locations. Starting from step 4, the trailing matrix update, which is a GEMM (matrix-matrix multiplication), picked up the erroneous data for computation. As the iteration continued, the errors grew downward into the trailing matrix (in yellow). When they reached the diagonal, the erroneous data started to participate in the vertical scaling of zeroing out values below diagonals, and as a result the entire trailing matrix was contaminated immediately, shown in red dots. Both of the two errors followed the same propagation pattern. This example shows that soft error propagation could result in large area of erroneous data.

3.2. General Workflow

We proposed an ABFT based method to protect the LU factorization based linear solver. This method can tolerate multiple occurrences of soft error in the whole area of factorization result and recover the correct solution x to the linear system of equations $Ax = b$. The general workflow of error detection and recovery is in Algorithm 1. Checksum that is generated before solving the system is used to check and locate the soft errors and eventually recover the solution. The verification process is performed with no "online checking" interruption.

4. Encoding for Multiple Errors in L

The first step of the workflow in Algorithm 1 is to checkpoint the input matrix A with a generator matrix G. For the single error case, it has been demonstrated in [9] that generator matrix $G_1 = \begin{bmatrix} 1 & \cdots & 1 \\ w_1 & \cdots & w_n \end{bmatrix}$ and check matrix $H_1 = \begin{bmatrix} 1 & \cdots & 1 & -1 \\ w_1 & \cdots & w_n & -1 \end{bmatrix}$ work for the entire area of factorization result. In this section, we apply this idea to multiple errors in L, and the next section further extends it to protecting U. Only the encoding issue is discussed. For a scalable

Algorithm 1 Fault Tolerant System Workflow

Require: $Ax = b$; Generator matrix G; Check matrix H
Step 1: Checkpoint A by $A_c = \begin{bmatrix} A & A \times G \end{bmatrix}$
Step 2: Perform LU factorization $L_c U_c = P \times A_c$ in block algorithm of block size NB with partial pivoting; Pivoting to L is delayed; Panel factorization result in each step is checkpointed immediately once produced; Errors in L are correct after factorization and then pivoting in L is applied
Step 3: Detect error occurrence by checking $\delta = \|U_c \times H\|$
if Found error(s) by $\delta >> 0$ **then**
Step 3.1: Locate initial error(s) in U using δ
Step 3.2: Calculate $\hat{x}$ by $\hat{x} = \hat{U}(\backslash L \backslash (P \times b))$, and
Step 3.3: Adjust $\hat{x}$ to the correct solution $x = \hat{x} + \Delta$
else
Step 4: Reach the correct solution $x = U \backslash (L \backslash (P \times b))$
end if

implementation, we continues to use the local checkpointing method in [9] where each process checkpoints its local participating blocks in the current panel area.

For any column of the factorized panel in L, $[l_1, l_2, \cdots, l_k]^T$, the vertical checkpointing produces the following three checksums c_1 to c_3:

$$\begin{cases} l_1 + l_2 + \cdots + l_k = c_1 \\ w_1 l_1 + w_2 l_2 + \cdots + w_k l_k = c_2 \\ u_1 l_1 + u_2 l_2 + \cdots + u_k l_k = c_3 \end{cases} \tag{2}$$

Since all computation is carried out in floating point number that has a fixed number of digits for exponent and fraction, the selection of w_i and u_i should avoid causing large contrast between operands that encourages the accumulation of round-off errors. As an opposite example, in [8], the use of Vandermonde matrix where $w_i = j$ and $u_i = j^2$ incurs fast increase of value magnitude in checksum and causes notable precision loss from round-off errors.

To work with round-off errors, we propose to choose w_i and u_i from random numbers between 0 and 1. Suppose soft errors change l_i and l_j to $\hat{l_i}$ and $\hat{l_j}$ respectively, $i < j$. During the error locating step (step 3.1) in Algorithm 1, re-generating the checksum gives:

$$\begin{cases} l_1 + \cdots + \hat{l_i} + \cdots + \hat{l_j} + \cdots + l_k = \hat{c_1} \\ w_1 l_1 + \cdots + w_i \hat{l_i} + \cdots + w_j \hat{l_j} + \cdots + w_k l_k = \hat{c_2} \\ u_1 l_1 + \cdots + u_i \hat{l_i} + \cdots + u_j \hat{l_j} + \cdots + u_k l_k = \hat{c_3} \end{cases} \tag{3}$$

Subtract (3) from (2), we have

$$\begin{cases} \hat{c_1} - c_1 = \hat{l_i} - l_i + \hat{l_j} - l_j \\ \hat{c_2} - c_2 = w_i(\hat{l_i} - l_i) + w_j(\hat{l_j} - l_j) \\ \hat{c_3} - c_3 = u_i(\hat{l_i} - l_i) + u_j(\hat{l_j} - l_j) \end{cases} \tag{4}$$

Define the system of equations in (4) as the "symptom equations". The symptom equations establish the relationship between soft errors and checksum, however it cannot be solved "as is" since the six unknowns $\hat{l_i}$, $\hat{l_j}$, w_i, w_j and u_i, u_j outnumber the available three equations.

To reduce the number of unknowns, let $u_i = w_i^2,\ i = 1, \cdots, k$. Combine the first and second equation in (4):

$$\hat{l_j} - l_j = \frac{1}{w_j - w_i}((\hat{c_2} - c_2) - w_i(\hat{c_1} - c_1)) \tag{5}$$

And similarly combine the first and third equation:

$$\hat{l_j} - l_j = \frac{(\hat{c_3} - c_3) - w_i^2(\hat{c_1} - c_1)}{w_j^2 - w_i^2} \tag{6}$$

Eliminate $\hat{l}_j - l_j$ from (5) and (6) by connecting the right hand sides, (4) can be eventually reduced to

$$(\hat{c_3} - c_3) - (w_i + w_j)(\hat{c_2} - c_2) + w_i w_j(\hat{c_1} - c_1) = 0 \tag{7}$$

Define this equation as the "check equation". w_i, w_j can be determined by iterating through all possibilities in w with $O(N^2)$ complexity because $i < j$, and for each i, $N - i$ pairs of w_i w_j are tested in (7).

The error detection and recovery algorithm can be extended to t errors with complexity $O(N^t)$ to determine the locations of up to t errors. For example, when $t = 3$, symptom equation 4 is

$$\begin{cases} \hat{c_1} - c_1 = \hat{l}_i - l_i + \hat{l}_j - l_j + \hat{l}_k - l_k \\ \hat{c_2} - c_2 = w_i(\hat{l}_i - l_i) + w_j(\hat{l}_j - l_j) + w_k(\hat{l}_k - l_k) \\ \hat{c_3} - c_3 = u_i(\hat{l}_i - l_i) + u_j(\hat{l}_j - l_j) + u_k(\hat{l}_k - l_k) \\ \hat{c_4} - c_4 = h_i(\hat{l}_i - l_i) + h_j(\hat{l}_j - l_j) + h_k(\hat{l}_k - l_k) \end{cases} \tag{8}$$

Here i, j and k correspond to the three errors' locations. Similar to the double-error case, the check equation can be derived as

$$\frac{C_4(w_i - w_j) + w_i^3(w_j C_1 - C_2) - w_j^3(w_i C_1 - C_2)}{(w_i - w_j)w_k^3 - (w_i - w_k)w_j^3 + (w_j - w_k)w_i^3} = \frac{C_3(w_i - w_j) + w_i^2(w_j C_1 - C_2) - w_j^2(w_i C_1 - C_2)}{(w_i - w_j)w_k^2 - (w_i - w_k)w_j^2 + (w_j - w_k)w_i^2}$$

Similarly, by iterating through all possible pairs of w_i, w_j and w_k using the check equation, the three error locations can be determined and the error value can be found accordingly.

5. Encoding for Multiple Errors in $\bar{U}$ and $\tilde{U}$

Soft errors in $\bar{U}$ and $\tilde{U}$ differ from those in L because they participate in the computation and cause error propagation. The $t = 2$ case is discussed in detail and is then extended to $t > 2$ errors.

5.1. Soft Errors Modeling

For temporal multiple soft errors, Luk et al. has proposed to cast soft error to a differnt initial matrix to avoid the difficulty of knowing when soft error occurs [7]. The effect of soft error during factorization is treated as rank-one perturbation to the original matrix. Fitzpatrick et al. applied this method to double error modeling for Gaussian elimination [8]. This section extends the encoding method in section 4 to provide protection to U.

LU factorization can be viewed as multiplying a set of triangularization matrices from the left to the input matrix A to get the final triangular form. Let $A_0 = A$, and $A_t = L_{t-1}P_{t-1}A_{t-1}$. P_{t-1} is the partial pivoting matrix at step $t - 1$. At the end of the factorization, $PA_0 = LU$, where U is an upper triangular matrix.

Suppose two soft errors occur in the $\bar{U}$ or $\tilde{U}$ area at locations (i_1, j_1) and (i_2, j_2) in step s_1 and s_2 separately. In the most general case, $s_1 \neq s_2$, $i_1 \neq i_2$ and $j_1 \neq j_2$. Without loss of generality, let $s_1 < s_2$.

At step s_2, express the soft error as a perturbation to the matrix at location (i_2, j_2): $\hat{A}_{s_2} = A_{s_2} - \delta e_{i_2} e_{j_2}^T$. A_{s_2} is the state of the matrix at step s_2 before soft error occurs, and $\hat{A}_{s_2}$ is outcome of A_{s_2} modified by a soft error of magnitude δ at location (i_2, j_2). e_{i_2} and e_{j_2} are column vectors with all 0s except a 1 at rows i_2 and j_2 respectively.

The error at step s_2 is cast back as a perturbation to the matrix at step s_1,

$$\begin{aligned} & \hat{A}_{s_2} = A_{s_2} - \delta e_{i_2} e_{j_2}^T = L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2}\cdots L_{s_1}P_{s_1}\hat{A}_{s_1} - \delta e_{i_2} e_{j_2}^T \\ \therefore \quad & (L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2}\cdots L_{s_1}P_{s_1})^{-1}\hat{A}_{s_2} = \hat{A}_{s_1} - (L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2}\cdots L_{s_1}P_{s_1})^{-1}\delta e_{i_2} e_{i_2}^T \end{aligned}$$

Let $f = (L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2}\cdots L_{s_1}P_{s_1})^{-1}\delta e_{i_2}$, and $(L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2}\cdots L_{s_1}P_{s_1})^{-1}\hat{A}_{s_2} = \hat{A}_{\bar{s_2}}$, we have:

$$\hat{A}_{\bar{s_2}} = \hat{A}_{s_1} - f e_{j_2}^T \tag{9}$$

Continue casting (9) to the soft error at step s_1, eventually we have

$$\hat{A}_{\bar{s_1}} = A_0 - d e_{j_1}^T - g e_{j_2}^T \tag{10}$$

Here $g = (L_{s_1-1}P_{s_1-1}L_{s_1-2}P_{s_1-2}\cdots L_0P_0)^{-1} \times f = (L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2}\cdots L_0P_0)^{-1}\delta e_{i_2}$

Through this modeling process, the two soft errors are cast back to the input matrix A_0 as perturbation to column j_1 and j_2. For more than 2 errors, the same process can be repeated and the general model for t errors is

$$\hat{A}_0 = A_0 - \sum_{j=1}^{t} d_{j_i} e_{j_i}^T \tag{11}$$

5.2. Errors Detection and Locating

With the model for soft errors, errors' locations can be determined. This model is for the case where soft errors occur only in matrix A. In fact checksum and the right hand sides b of $Ax = b$ are equally susceptible to soft errors. These cases can be protected by duplication and cross check, and the protection method for L in section 4 can be directly applied to protect right hand sides.

In [8], four columns of checksum are used to locate two soft errors. Instead, we show that for N errors, $N + 1$ columns are sufficient for error detection and data recovery.

For the input matrix $A \in \mathbb{R}^{N\times N}$, checksum is generated before the factorization using generator matrix

$$G = \begin{bmatrix} e^T \\ w^T \\ (w^2)^T \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ w_1 & \cdots & w_N \\ w_1^2 & \cdots & w_N^2 \end{bmatrix} \tag{12}$$

and A is encoded as $[A, A \times G^T] = [A, Ae, Aw, Aw^2]$. The squaring operation is elementwise.

LU factorization is applied with the three additional checksum columns on the right as

$$P[A,\ Ae,\ Aw,\ Aw^2] = L[U,\ c,\ v,\ s]$$

$c, v, s \in \mathbb{R}^{N\times 1}$ are checksum after factorization.

Due to soft errors, A becomes erroneous. As shown in the error model, the LU factorization infected with soft errors is equal to an soft-error-free LU factorization of a different initial (erroneous) matrix $\hat{A}_0$. Using A to represent the original correct initial matrix and $\hat{A}$ for the erroneous initial matrix, (5.2) becomes:

$$\hat{P}[\hat{A},\ Ae,\ Aw,\ Aw^2] = \hat{L}[\hat{U},\ \hat{c},\ \hat{v}, \hat{s}]$$

And using relationship between $\hat{c}$ and Ae:

$$\hat{c} \quad = \quad \hat{L}^{-1}\hat{P}Ae = \hat{L}^{-1}\hat{P}(\hat{A} + de_{j_1}^T + ge_{j_2}^T)e = \hat{L}^{-1}(\hat{L}\hat{U} + \hat{P}de_{j_1}^T + \hat{P}ge_{j_2}^T)e = \hat{U}e + \hat{L}^{-1}\hat{P}d + \hat{L}^{-1}\hat{P}g$$

Therefore, $\hat{c} - \hat{U}e = \hat{L}^{-1}\hat{P}d + \hat{L}^{-1}\hat{P}g$.

By the same token, $\hat{v} - \hat{U}w = w_{j_1}\hat{L}^{-1}\hat{P}d + w_{j_2}\hat{L}^{-1}\hat{P}g$, and $\hat{s} - \hat{U}w^2 = w_{j_1}^2\hat{L}^{-1}\hat{P}d + w_{j_2}^2\hat{L}^{-1}\hat{P}g$.

Let $x = \hat{L}^{-1}\hat{P}d \in \mathbb{R}^{N\times 1}$, and $y = \hat{L}^{-1}\hat{P}g \in \mathbb{R}^{N\times 1}$. We have $\begin{cases} \hat{c} - \hat{U}e = x + y \\ \hat{v} - \hat{U}w = w_{j_1}x + w_{j_2}y \\ \hat{s} - \hat{U}w^2 = w_{j_1}^2 x + w_{j_2}^2 y \end{cases}$

This system of equations is the vector form of (4), and similarly can be reduced to the check equation:

$$(\hat{s} - \hat{U}w^2) - (w_{j_1} + w_{j_2})(\hat{v} - \hat{U}w) + w_{j_1}w_{j_2}(\hat{c} - \hat{U}e) = 0 \tag{13}$$

w_{j_1} and w_{j_2} can be determined by iterating through all possible $N\times(N-1)$ combinations in w for a pair that makes (13) hold. As a result, the error columns j_1 and j_2 are determined. Later, using the error columns, solution of $Ax = b$ can be recovered.

For t soft errors, with the error model in (11), the symptom equation is:

$$\begin{cases} c_0 - \hat{U}w^0 &= w_{j_1}^0 x_1 + \cdots + w_{j_t}^0 x_t \\ c_1 - \hat{U}w^1 &= w_{j_1}^1 x_1 + \cdots + w_{j_t}^1 x_t \\ & \vdots \\ c_{t-1} - \hat{U}w^{t-1} &= w_{j_1}^{t-1} x_1 + \cdots + w_{j_t}^{t-1} x_t \end{cases} \tag{14}$$

All the powers in (14) are elementwise. This general case of check equation in vector form for t soft errors exhibits the same structure as in the scalar form. For $t = 3$ it has been shown that check equation (9) can be used to determine error locations except the scalar residues C_i is replaced with vector residues $c_i - \hat{U}w^i$.

For two errors, the complexity of locating w_{j_1} and w_{j_2} is $O(N^3)$ because for each pair of w_{j_1} and w_{j_2} a vector norm is calculated to test for zero vector in (13) which takes $O(N)$ operations. For $t > 2$, the complexity to determine the error columns exceeds the complexity of LU factorization, rendering this method computationally impractical for real use. The same problem exists for L protection too when $t > 3$. The next section provides solution to this issue.

Since errors in $\bar{U}$ and $\tilde{U}$ propagate, the solution to (14) alone is insufficient for recovering the right factor U as only the columns of the initial errors can be determined. However for a system of linear equations, by using Sherman-Morrison-Woodbury formula, the solution can be recovered.

6. Complexity Reduction

As the number of tolerable errors t increases, the complexity of locating initial error columns grows exponentially. To allow practical use, a complexity reduction method is proposed in this section.

6.1. Reduction for L

In the complexity $O(N^t)$ for locating t errors, N is the factor that determines the range of search. By breaking the search range into smaller segments, the complexity can be decreased to an affordable level. There exist many ways of segmenting N but since each segment requires separate storage for checksum, the segmenting method should minimize the overall storage requirement. Use N_k to represent the segment size, the k_{th} root of the vector length is chosen in this work as the segment size where k is integer and , $k \geq 1$.

Split N into equally sized segments of length $N_k = N^{\frac{1}{k}}$. Apply the encoding method in (2) to each of the $N^{1-\frac{1}{k}}$ segments. For each vector to tolerate t errors, $t + 1$ checksum items are required. Therefore for a vector of length N, the total amount of space required to store checksums is

$$N^{1-\frac{1}{k}} \times (t + 1) \tag{15}$$

And the storage overhead over that for the data vector has the trend

$$\lim_{N\to\infty} (N^{1-\frac{1}{k}} \times (t + 1) \times \frac{1}{N}) = \lim_{N\to\infty} \frac{t + 1}{\sqrt[k]{N}} = 0 \tag{16}$$

Since the expensive error locating procedure is now carried out within a smaller range, the complexity of error detection is largely reduced. The total overhead of locating t soft errors includes $N^{1-\frac{1}{k}}$ vector norms of length $\sqrt[k]{N}$, and iterating in $\sqrt[k]{N}$ for the correct pair of w_i and w_j $O(N^{\frac{1}{k}} \times N^{1-\frac{1}{k}}) + O((N^{\frac{1}{k}})^t) = O(N) + O(N^{\frac{t}{k}}) = \begin{cases} O(N) & \text{if } t \leq k \\ O(N^{\frac{t}{k}}) & \text{if } t > k \end{cases}$

Note that the number of tolerable soft error t is for each segment. Therefore to tolerate the same amount of soft errors, each segment could opt for a smaller t than in the $k = 1$ case. For a fixed t, increasing k has the same effect by reducing the range of search, but comes at the cost of more extra storage according to (15).

6.2. Reduction for U

For $\bar{U}$ and $\tilde{U}$, without any complexity management, locating t soft errors requires $O(N^{t+1})$ operations, one order higher than the original complexity for L protection. To reduce the complexity to an affordable level, the segmenting method in section 6.1 is extended to the block LU algorithm for $\bar{U}$ and $\tilde{U}$ protection.

6.2.1. Block Encoding of Matrix

In block LU algorithm, the panel factorization itself is an LU factorization of a tall and skinny block, therefore the encoding technique in (5) can be used to protect a panel or several panels too if the encoding is performed accordingly. For the whole matrix, the segmented encoding can be applied based on the following theorem (proof is omitted due to space constrain):

Theorem 6.1. *Block Encoding protects the trailing matrix at the end of each iteration of LU factorization*

For illustration, consider the following example: take a matrix A of 2×2 blocks encoded using the generator in (12)

$$A_c = \begin{bmatrix} A_{11} & A_{12} & A_{11}G & A_{12}G \\ A_{21} & A_{22} & A_{21}G & A_{22}G \end{bmatrix}$$

Carry out LU factorization to A_c and we have: $A_c = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L22 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & C_{11} & C_{12} \\ 0 & U_{22} & C_{21} & C_{22} \end{bmatrix}$

And C_1 to C_4 are calculated as: $\begin{cases} C_{11} & = U_{11}G, \; C_{12} & = U_{12}G \\ C_{21} & = \varnothing, \; C_{22} & = U_{22}G \end{cases}$. This shows that after LU factorization, the added four checksum blocks offer protection to the three data blocks U_{11}, U_{12} and U_{22} independently. Since G in (12) offers t errors protection capability, the three data blocks in U ***each*** can tolerate up to t soft errors.

In ScaLAPACK, matrix A is split into blocks of size $NB \times NB$, therefore when $k = 2$, the encoding block size N_k is $N \times \sqrt{N}$ rounded to multiple of NB. Error detection is performed on each $\sqrt{N} \times \sqrt{N}$ blocks. For U_{11}, first $\|U_{11} \times G(:, 1) - C_{11}(:, 1)\|$ is checked and if the norm is sufficiently large, the error detection procedure in 5.2 is then performed for this $\sqrt{N} \times \sqrt{N}$ block.

The complexity of performing blocked error detection and locating includes the error check that is either a full or upper triangular matrix-vector multiplication and the error locating operation within the block. Suppose $N_k = N^{\frac{1}{k}}$ rounded to a multiple of NB, and the generator matrix G has size $N_k \times t$ for t error resilience capability. Since error checking is only carried out in the upper triangular blocks of A, there are in total $1 + 2 + \cdots + N^{1-\frac{1}{k}}$ number of blocks. Therefore the error checking complexity is $(1 + 2 + \cdots + N^{1-\frac{1}{k}}) \times O((N^{\frac{1}{k}})^2) = \frac{N^{1-\frac{1}{k}}(N^{1-\frac{1}{k}}+1)}{2} \times O(N^{\frac{2}{k}})$.

And the error locating complexity is $O(N^{\frac{t}{k}} \times N^{\frac{1}{k}})$. For instance, when $k = 2$ and $t = 2$, the total overhead of error detection and locating is $\frac{\sqrt{N}(\sqrt{N}+1)}{2} \times O(N) + O(N^{\frac{3}{2}}) = O(N^2) < O(N^3)$. Therefore the overhead is affordable for the solver based on LU factorization.

The total amount of extra storage for storing checksum columns is $N \times N^{1-\frac{1}{k}} \times (t + 1)$. And the storage overhead over that for the data vector has the trend $\lim_{N\to\infty} (N \times N^{1-\frac{1}{k}} \times (t + 1) \times \frac{1}{N^2}) = \lim_{N\to\infty} \frac{t+1}{\sqrt[k]{N}} = 0$.

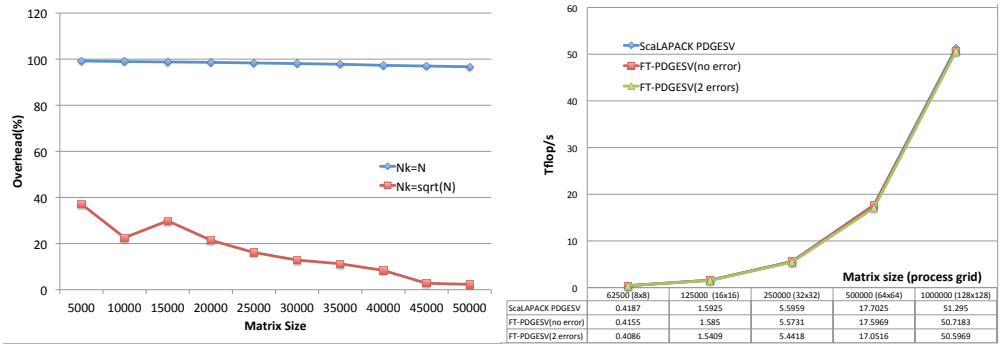
7. Recovery Algorithm

After soft errors are detected and located by their columns, the correct solution to the system of equations $Ax = b$ can be recovered using the Sherman-Morrison-Woodbury formula as suggested by [8]. The computing complexity of recovery the solution x is $O(N^2)$.

8. Performance Evaluation

Soft errors in the left factor are static, and the detection and recovery for this area has been validated in [11, 9] showing little performance impact to the host algorithm. The algorithms for multiple soft errors in the right factor, namely $\tilde{U}$ and $\bar{U}$, on the other hand, have higher complexity and are most effectively affected by the proposed encoding and complexity reduction method. Therefore only the validation for this part is shown in this section.

The experiments are performed on a large scale distributed memory system: the Kraken supercomputer by Cray Inc. at the Oak Ridge National Lab. Kraken has 9,408 compute nodes. Each node has two Istanbul 2.6 GHz six-core AMD Opteron processors and 16 GB of memory. All the nodes are Connected by the Cray SeaStar2+ interconnect. In the experiments, two soft errors are injected into location $(336, 361)$ and $(347, 359)$ at the beginning of the $2nd$ and $3rd$ panel factorization, respectively. Data values are incremented with random magnitudes to simulate the results of bit flips in the memory slots that hold these data. The block size for encoding is $\sqrt{N}$.



	62500 (8x8)	125000 (16x16)	250000 (32x32)	500000 (64x64)	1000000 (128x128)
ScaLAPACK PDGESV	0.4187	1.5925	5.5959	17.7025	51.295
FT-PDGESV(no error)	0.4155	1.585	5.5731	17.5969	50.7183
FT-PDGESV(2 errors)	0.4086	1.5409	5.4418	17.0516	50.5969

(a) Result on Kraken with 16,384 (128×128) cores (b) Weak scalability result on Kraken

Figure 2: Experiment result on Kraken

Figure 2a shows the effectiveness of the complexity reduction method for U with a 16×16 process grid on Kraken, $t = 2$. The overhead is in Gflop/s and calculated as $\frac{FLOPS_{non-FT} - FLOPS_{FT}}{FLOPS_{non-FT}}\%$. When $N_k = N$, block encoding for soft errors in U is not in effect. The whole matrix is encoded with a generator matrix of size $N \times 3$. In this case the overhead is close to 100% (the blue line), which means the error detection and recovery combined take as much time as solving the linear system of equations. This is consistent with the theoretical complexity of $O(N^{t+1}) = O(N^3)$. The red line, on the other hand, is the result when $N_k = \sqrt{N}$. The overhead drops quickly from a little less than 40% to 2%, which verifies that block encoding largely reduces the error detection overhead. The cost of this improvement is the extra space for storing checksum which is roughly 1% of the input matrix of size 50,000.

Figure 2b is the weak scalability experiment result where matrix size and grid dimension are doubled in proportion. Throughout all the testing sizes from 64 to 16,384 cores, FT-PDGESV declares around 1% overhead with and without soft error recovery. The solution to the linear system was successfully recovered.

The experiment result confirms that the complexity of recovering the solution to $Ax = b$ from two soft errors in the right factor has been effectively managed by the complexity reduction method, and soft errors can be precisely detected and located with the presence of round-off error. The fault tolerance functionalities can recover the solution of the dense linear system with trivial performance impact.

9. Related Work

In the field of fault tolerance for HPC systems, checkpoint-restart (C/R) is the most commonly used method [12]. The running state of the application is written to reliable storage at certain intervals automatically by the message passing middleware or at the request of the user application. C/R requires the least user intervention but suffers from high overhead from checkpointing through disk I/O. To reduce the overhead, diskless checkpointing [13] turns to system memory for checksum storage rather than disks. Applications have seen better fault tolerance performance than C/R [14] for dense linear algebra problem. Both C/R and diskless checkpointing need the error information for recovery, which is not available with soft error. Algorithm based fault tolerance (ABFT) eliminates the need for periodical checkpointing. This significantly reduced checkpointing overhead during computing, and the checksum by ABFT reflects the most current status of the data and therefore offers clues for soft error detection and recovery. ABFT was originally introduced to deal with silent error in systolic arrays [15, 16]. Data is encoded before the computation begins. Matrix algorithms are designed to work on the encoded checksum along with matrix data, and the correctness is checked after the matrix operation completes.

Using ABFT to mitigate single soft errors in dense matrix factorization has been explored in [7, 17]. Later, this was extended to multiple errors [18, 8, 19] by adopting methodology from finite-field based error correcting code (e.g. Reed-Solomon [20]) where only the right factor of factorization result is protected and computation is assumed to take place with exact arithmetics. These make the ECC based error location determination method loose effectiveness.

Recently, iterative solvers were evaluated for soft error vulnerability [21, 22, 23], signifying the continued awareness of soft error for solving large scale problems. For dense matrices, the effect of soft errors on linear algebra

packages like BLAS and LAPACK has also been studied [24], which showed that their reliability can be improved by checking the output of the routine, and the error patterns do not depend on the problem size. Also, the possibility of predicting the fault propagation is explored. For dense matrix factorization based solver, method to mitigate single soft error has been discussed in [9] and the recovery of matrix factorization was shown in [11] using QR for demonstration.

10. Conclusion

Soft error resilient algorithm for LU factorization based dense linear system solver is proposed in this work. Both spatial and temporal multiple soft errors in the whole matrix can be addressed with the existence of round-off errors from floating point operation. Once errors are detected, the solution of $Ax = b$ can be recovered with low overhead using the complexity reduction technique. Experimental results on the Kraken supercomputer confirm both the soft error mitigation capability and the negligible performance overhead. In future work, the proposed method will be extended to the protection of dense matrix factorizations like LU and QR.

References

[1] S. Michalak, K. Harris, N. Hengartner, B. Takala, S. Wender, Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer, Device and Materials Reliability, IEEE Transactions on 5 (3) (2005) 329–335.

[2] D. Lyons, Sun screen, available at `http://www.forbes.com/forbes/2000/1113/6613068a.html` (November 13 2000).

[3] B. Schroeder, E. Pinheiro, W. Weber, DRAM errors in the wild: a large-scale field study, in: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, ACM, 2009, pp. 193–204.

[4] H. W. Meuer, E. Strohmaier, J. J. Dongarra, H. D. Simon, TOP500 Supercomputer Sites, 36th Edition, (The report can be downloaded from `http://www.netlib.org/benchmark/top500.html`) (November 2010).

[5] R. Barrett, T. Chan, E. D'Azevedo, E. Jaeger, K. Wong, R. Wong, Complex version of high performance computing linpack benchmark (hpl), Concurrency and Computation: Practice and Experience 22 (5) (2010) 573–587.

[6] D. Abts, J. Thompson, G. Schwoerer, Architectural support for mitigating dram soft errors in large-scale supercomputers.

[7] F. Luk, H. Park, An analysis of algorithm-based fault tolerance techniques* 1, Journal of Parallel and Distributed Computing 5 (2) (1988) 172–184.

[8] P. Fitzpatrick, C. Murphy, Fault tolerant matrix triangularization and solution of linear systems of equations, in: Application Specific Array Processors, 1992. Proceedings of the International Conference on, IEEE, 1992, pp. 469–480.

[9] P. Du, P. Luszczek, J. Dongarra, High performance dense linear system solver with soft error resilience, in: Proceedings of the IEEE Cluster 2011, IEEE Computer Society Press, 2011.

[10] J. Dongarra, L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, et al., ScaLAPACK user's guide, Society for Industrial and Applied Mathematics, Philadelphia, PA.

[11] P. Du, P. Luszczek, S. Tomov, J. Dongarra, Soft error resilient QR factorization for hybrid system., Tech. Rep. 252, LAPACK Working Note (Jul. 2011).
URL `http://www.netlib.org/lapack/lawnspdf/lawn252.pdf`

[12] Fault tolerance for extreme-scale computing workshop report (2009).
URL `http://www.teragridforum.org/mediawiki/images/8/8c/FT_workshop_report.pdf`

[13] J. Plank, K. Li, M. Puening, Diskless checkpointing, Parallel and Distributed Systems, IEEE Transactions on 9 (10) (1998) 972–986.

[14] J. Plank, Y. Kim, J. Dongarra, Algorithm-based diskless checkpointing for fault-tolerant matrix operations, in: ftcs, Published by the IEEE Computer Society, 1995, p. 0351.

[15] K. Huang, J. Abraham, Algorithm-based fault tolerance for matrix operations, Computers, IEEE Transactions on 100 (6) (1984) 518–528.

[16] J. Abraham, Fault tolerance techniques for highly parallel signal processing architectures, Highly parallel signal processing architectures (1986) 49–65.

[17] F. Luk, H. Park, Fault-tolerant matrix triangularizations on systolic arrays, Computers, IEEE Transactions on 37 (11) (1988) 1434–1438.

[18] H. Park, On multiple error detection in matrix triangularizations using checksum methods, Journal of Parallel and Distributed Computing 14 (1) (1992) 90–97.

[19] C. Anfinson, F. Luk, A linear algebraic model of algorithm-based fault tolerance, Computers, IEEE Transactions on 37 (12) (1988) 1599–1604.

[20] I. Reed, G. Solomon, Polynomial codes over certain finite fields, Journal of the Society for Industrial and Applied Mathematics 8 (2) (1960) 300–304.

[21] G. Bronevetsky, B. de Supinski, Soft error vulnerability of iterative linear algebra methods, in: Proceedings of the 22nd annual international conference on Supercomputing, ACM, 2008, pp. 155–164.

[22] K. Malkowski, P. Raghavan, M. Kandemir, Analyzing the soft error resilience of linear solvers on multicore multiprocessors, in: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, IEEE, pp. 1–12.

[23] V. Heuveline, D. Lukarski, F. Oboril, M. Tahoori, J. Weiss, Numerical defect correction as an algorithm-based fault tolerance technique for iterative solvers.

[24] G. Bronevetsky, B. de Supinski, M. Schulz, A Foundation for the Accurate Prediction of the Soft Error Vulnerability of Scientific Applications, Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA (2009).