

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

Peng Du\*, Piotr Luszczek\*, Jack Dongarra†

\* *EECS, University of Tennessee; 1122 Volunteer Blvd., Knoxville, TN 37996-3450, USA*

*Email: {du, luszczek}@eecs.utk.edu*

† *University of Tennessee; Oak Ridge National Laboratory, Oak Ridge, TN, USA; University of Manchester, Manchester, UK*

*Email: dongarra@eecs.utk.edu*

**Abstract**—In the multi-peta-flop era for supercomputers, the number of computing cores is growing exponentially. However, with integrated circuit technology scaling below 65 nm, the critical charge required to flip a gate or a memory cell is dangerously reduced. Combined with higher vulnerability to cosmic radiation, soft errors are expected to become anything but inevitable for modern supercomputer systems. As a result, for long running applications on high-end machines, including linear solvers for dense matrices, soft errors have become a serious concern. Classical checkpoint and restart (C/R) scheme loses effectiveness against this threat because of the difficulty to detect soft errors in the form of transient bit flips that do not interrupt program execution and therefore leave no trace of error occurrence. Current research of soft errors resilience for dense linear solvers offers limited capability when faced with large scale computing systems that suffer both round-off error from floating point arithmetic and the presence followed by propagation of multiple soft errors. The use of error correcting codes based on Galois fields requires high computing cost for recovery. This work proposes a fault tolerant algorithm for dense linear system solver that is resilient to multiple spatial and temporal soft errors. This algorithm is designed to work with floating point data and is capable of recovering the solution of  $Ax = b$  from multiple soft errors that affect any part of the matrix during computation. Additionally, the computational complexity of the error detection and recovery is optimized through novel methods. Experimental results on cluster systems confirm that the proposed fault tolerance functionality can successfully detect and locate soft errors and recover the solution of the linear system. The performance impact is negligible and the soft errors resilient algorithm's performance scales well on large scale systems.

**Keywords**—soft error; fault tolerance; multiple errors; dense linear system solver;

## I. 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, however, 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 2002 and 2003, the 2048-node ASC Q supercomputer for scientific computing in Los Alamos National Laboratory experienced failure from extensive soft errors [23]. By comparing the error logs with a radiation experiment conducted

in a lab, the cause was soon identified to be the cosmic ray striking its parity-protected cache tag array. The Q computer is more vulnerable to soft errors because it is located at about 7500 feet above the sea level, and the neutrons from cosmic-rays are roughly 6.4 times stronger than the ones occurring at sea level. 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 [20]. These incidents signify that soft errors are 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 [28]. As CMOS technology scales the feature size down with more transistors per chip and lower critical charge [14], [30], the threat of soft errors will keep haunting the computing community.

In order to mitigate the impact of soft errors, modern HPC systems rely heavily on ECC (error correcting code) [3]. Nowadays the most commonly used ECC is SECDED (Single Error Correction, Double Error Detection). For multi-bit errors precipitated by the progress of the integrated circuit technology [29], a more powerful form of ECC has become too expensive due to the higher encoding and decoding overhead and the resulting memory performance loss. In addition, many parts of the chip that are not protected by ECC, like caches, register files and other less commonly known logic circuits, may also fall victim of soft errors. This raises the question on whether soft error resilience can be achieved with less cost from the application side [13]. Among HPC applications that could benefit from such fault tolerance capability, dense linear algebra applications such as the HPL benchmark for the TOP500 competition [22] 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  $A$  as large as 500,000. Soft errors that occur during such long running applications produce incorrect solution with

no apparent reason. This lowers productivity by wasting valuable time and energy in error tracing with little chance of locating the error.

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 the error correcting code does not seriously affect system performance and the encoding can be carried out with exact arithmetic [12], [18]. Unfortunately, none of these assumptions hold true for today’s P flop/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 the spatial and temporal distribution of multiple errors. Spatial soft errors occur at different time, whereas temporal soft errors manifest as simultaneous multiple bit flips in disparate locations. Experiments on the Kraken supercomputer from Cray at the University of Tennessee verified our design for both the error detection and correction capability as well as low performance complexity. 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 II introduces an LU based dense linear solver on distributed memory system. The impact of soft error on the linear solver is then analyzed and the general work flow of the proposed soft error resilience algorithm is shown in Section III. Sections IV and V develop the protection method for both the left factor  $L$  and right factor  $U$ . Section VI proposes a block protection method to reduce the computational complexity of the non-blocking protection algorithm for  $U$ . Finally, the recovery algorithm is discussed in Section VII and the experimental results are shown in Section VIII. Related work is described in Section IX while Section X concludes the paper.

## II. HIGH PERFORMANCE LINEAR SYSTEM SOLVER

For dense matrix  $A$ , the LU factorization produces  $PA = LU$  (or  $P = ALU$ ), where  $P$  is a pivoting matrix,  $L$  and  $U$  are unit lower triangular matrix and upper triangular matrix respectively. LU factorization is popular for solving systems of linear equations. With  $L$  and  $U$ , the linear system  $Ax = b$  is solved by  $Ly = b$  and then  $Ux = y$ . ScaLAPACK 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 distribution is described with an  $N \times N$  matrix (or submatrix)  $A$ .

Split  $A$  into  $2 \times 2$  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

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

This poses as one iteration (step) of the factorization, and pivoting is applied on the left and right of the current panel. The routines names in the ScaLAPACK LU are listed after “ $\rightarrow$ ”. For description, we use  $\bar{U}$  to represent the area of  $U_{12}$  modified by PDTRSM, and  $\tilde{U}$  for  $A_{22}$  in PDGEMM.

Block algorithms offer good granularity to benefit from high performance BLAS routines, while 2D block-cyclic distribution ensures scalability with load balancing.

## III. 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 work flow of error detection and recovery is given. Details of each steps are explained in later sections.

### A. Error Pattern in the Block LU Algorithm

During the process of 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. Soft errors occurred in the factorized  $L$  area do not propagate. This offers an opportunity to use traditional diskless checkpointing method to protect these data. ABFT cannot be applied to the panel factorization since otherwise checksum rows for the panel could be moved into data causing erroneous result. In LU, partial pivoting that swaps rows of both  $L$  and  $U$  is normally 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 should be used economically. For example, even though the factorized  $\bar{U}$  (result of PDTRSM) also stays static once produced, it can be protected by ABFT checksum and therefore causes less overhead.

$\tilde{U}$  differs from  $L$  and  $\bar{U}$  in that it undergoes changes constantly from trailing matrix update. If soft errors alter

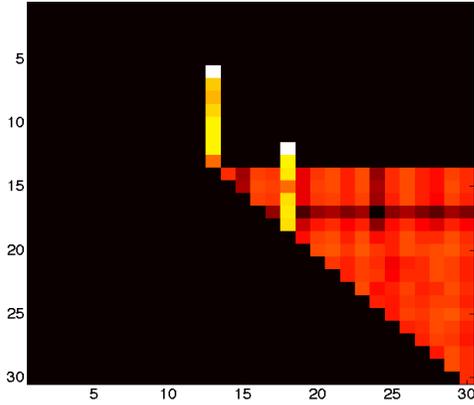


Figure 1. Example of error propagation in the U result of a  $30 \times 30$  matrix

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 a small matrix. Gaussian elimination is applied to a  $30 \times 30$  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 elimination, Two soft errors are injected at step 1 and 3 at location (6,13) and (12, 18) using addition. Since both errors occur below the row 3, these errors fall 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 GEMM(matrix-matrix multiple) picks up the erroneous data for computation. As the iteration continues, the errors grow downward into the trailing matrix (in yellow). When it reaches the diagonal, the erroneous data starts to participate in the vertical scaling of zeroing out values below diagonals, and immediately the errors take over the entire trailing matrix shown in red dots. Both of the two errors follow the same propagation pattern. In the red lower right section, propagated errors from both initial errors merge. Since the propagation occurs silently, it is challenging to detect and recover from such situation without any sign of error.

### B. General Work Flow

We proposed an ABFT based method to protect LU based linear solver. This method can tolerant multiple occurrences of soft error in the whole area of factorization result and restore the correct solution  $x$  to the linear system of equations  $Ax = b$ . The general work flow of error detection and recovery is in Algorithm 1.

---

### Algorithm 1 Fault Tolerant System Work Flow

---

**Require:**  $Ax = b$ ; Generator matrix  $G$ ; Check matrix  $H$

Step 1: Checkpointing  $A$  by  $A_c = [A \quad A \times G]$

Step 2: Perform LU factorization  $L_c U_c = P \times A_c$  in block algorithm of block size  $nb$  with partial pivoting; Panel factorization result in each step is checkpointed immediately once produced

Step 3: Detect error occurrence by  $\delta = \|U_c \times H\|$

**if** Found error(s) from  $\delta > 0$  **then**

    Step 3.1: Locate initial error(s) using  $\delta$

    Step 3.2: Detect and eliminate error(s) in  $L$

    Step 3.3: Calculate  $\hat{x}$  by  $\hat{x} = \hat{U} \setminus (\hat{L} \setminus (P \times b))$ , and

    Step 3.4: Adjust  $\hat{x}$  to the correct solution  $x = \hat{x} + \Delta$

**else**

    Step 4: Reach the correct solution  $x = U \setminus (L \setminus (P \times b))$

**end if**

---

### IV. ENCODING FOR MULTIPLE ERRORS IN $L$

The first step of the work flow 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 extend this idea to more than one errors cases for  $L$ , and later sections further develop it for protecting  $U$ . Only the encoding issue is discussed. For a scalable 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  $[l_1, l_2, \dots, l_k]^T$  in  $L$ , the objective of checkpointing is to allow recovery from errors that occur to a certain number of items silently altered in the column. First the errors are located, and then the correct values are restored.

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

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

Since all computation are carried out in floating point number with a fixed number of digits for exponent and fraction, the selection of  $w_i$  and  $u_i$  should avoid causing large contrast between operands during computing that encourages

the accumulation of round-off errors. As an opposite example, in [12], the use of Vandermonde matrix where  $w_i = j$  and  $u_i = j^2$  incur fast increase of checkpointing weight magnitude and causes notable precision loss from round-off errors. When this method is used with large matrices, the resulted error locations are ambiguously in between integers.

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 detection step (step 3.2) in Algorithm 1, re-generating the checksum gives:

$$\begin{cases} l_1 + \dots + \hat{l}_i + \dots + \hat{l}_j + \dots + l_k = \hat{c}_1 \\ w_1 l_1 + \dots + w_i \hat{l}_i + \dots + w_j \hat{l}_j + \dots + w_k l_k = \hat{c}_2 \\ u_1 l_1 + \dots + u_i \hat{l}_i + \dots + u_j \hat{l}_j + \dots + u_k l_k = \hat{c}_3 \end{cases} \quad (3)$$

Subtract (3) from (2), we have

$$\begin{cases} \hat{c}_1 - c_1 = \hat{l}_i - l_i + \hat{l}_j - l_j \\ \hat{c}_2 - c_2 = w_i(\hat{l}_i - l_i) + w_j(\hat{l}_j - l_j) \\ \hat{c}_3 - c_3 = u_i(\hat{l}_i - l_i) + u_j(\hat{l}_j - l_j) \end{cases} \quad (4)$$

This system of equations is defined 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 knowns, let  $u_i = w_i^2$ ,  $i = 1, \dots, k$ . Combine the first and second equation in (4):

$$\hat{l}_j - l_j = \frac{1}{w_j - w_i} ((\hat{c}_2 - c_2) - w_i(\hat{c}_1 - c_1)) \quad (5)$$

And similarly combine the first and third equation:

$$\hat{l}_j - l_j = \frac{(\hat{c}_3 - c_3) - w_i^2(\hat{c}_1 - c_1)}{w_j^2 - w_i^2} \quad (6)$$

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

$$(\hat{c}_3 - c_3) - (w_i + w_j)(\hat{c}_2 - c_2) + w_i w_j(\hat{c}_1 - c_1) = 0 \quad (7)$$

This equation is, in this work, defined 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).

This checkpointing method also applies to one-error recovery. Suppose an error occurs to  $l_i$  only, and (2) becomes

$$\begin{cases} \tilde{c}_1 - c_1 = \tilde{l}_i - l_i \\ \tilde{c}_2 - c_2 = w_i(\tilde{l}_i - l_i) \\ \tilde{c}_3 - c_3 = u_i(\tilde{l}_i - l_i) \end{cases} \quad (8)$$

The same method in [9] can be used to determine  $l_i$  from the first two equations of (8).

Using (4), the error detection and recovery algorithm is summarized in Algorithm 2. Note that this error protection for  $L$  applies for each column of  $L$ .

---

### Algorithm 2 Error detection and recovery in L

---

**Require:**  $\tilde{A}$ , error column  $l$  and generator row  $w$  of length  $N$ ,  $w_i, w_j \in w$  and  $w_i \neq w_j$ ,  $i, j \in \{1 \dots N\}$   
 Calculate  $\tilde{c}_i = \hat{c}_i - c_i$ ,  $i = 1, 2, 3$   
**if**  $\tilde{c}_i == 0$ ,  $i = 1, 2, 3$  **then**  
   No error  
**else if**  $\tilde{c}_2/\tilde{c}_1 == \tilde{c}_3/\tilde{c}_2 == w_i$  **then**  
   One error in row  $i$ , column  $l$  of the output matrix  
   Recover by solving  $\hat{c}_1 - c_1 = \hat{l}_i - l_i$   
**else**  
   At least two errors in column  $l$  of the output matrix  
   Iterate all possible pairs  $w_i, w_j \in w$   
   **if**  $(\tilde{c}_3 - c_3) - (w_i + w_j)(\tilde{c}_2 - c_2) + w_i w_j(\tilde{c}_1 - c_1) = 0$  **then**  
     Two errors are in rows  $i$  and  $j$ , column  $l$  of the output matrix  
     Recover by solving the overdetermined least square equations in (4) with  $w_i$  and  $w_j$  as known constants and  $x = \hat{l}_i - l_i$  and  $y = \hat{l}_j - l_j$  as unknowns  
   **else**  
     More than two errors occurs  
**end if**  
**end if**

---

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

$$\begin{cases} \hat{c}_1 - c_1 = \hat{l}_i - l_i + \hat{l}_j - l_j + \hat{l}_k - l_k \\ \hat{c}_2 - c_2 = w_i(\hat{l}_i - l_i) + w_j(\hat{l}_j - l_j) + w_k(\hat{l}_k - l_k) \\ \hat{c}_3 - c_3 = u_i(\hat{l}_i - l_i) + u_j(\hat{l}_j - l_j) + u_k(\hat{l}_k - l_k) \\ \hat{c}_4 - c_4 = h_i(\hat{l}_i - l_i) + h_j(\hat{l}_j - l_j) + h_k(\hat{l}_k - l_k) \end{cases} \quad (9)$$

Here  $i$ ,  $j$  and  $k$  correspond to the three errors' locations. Similar to the double-error case, we use  $u_i = w_i^2$  and  $h_i = w_i^3$ ,  $i = 1 \dots k$ . The symptom equations in (9) is simplified to:

$$\begin{cases} C_1 = x + y + z \\ C_2 = w_i x + w_j y + w_k z \\ C_3 = w_i^2 x + w_j^2 y + w_k^2 z \\ C_4 = w_i^3 x + w_j^3 y + w_k^3 z \end{cases} \quad (10)$$

where  $C_i = \hat{c}_i - c_i$ ,  $i = 1 \dots 4$ , and  $x = \hat{l}_i - l_i$ ,  $y = \hat{l}_j - l_j$ , and  $z = \hat{l}_k - l_k$ . The task is to determine  $w_i$ ,  $w_j$  and  $w_k$ .

Represent  $x$  and  $y$  as functions of  $z$  using the first two equations from (10):

$$\begin{cases} x = \frac{w_j C_1 C_2 - (w_j - w_k) z}{w_j - w_i} \\ y = \frac{w_i C_1 C_2 - (w_i - w_k) z}{w_i - w_j} \end{cases} \quad (11)$$

Replace  $x$  and  $y$  in the 3rd and 4th equations of (10)

with (11) and reduce  $z$ , the check equation is formed as:

$$\begin{aligned} & \frac{C_4(w_i - w_j) + w_i^3(w_j C_1 - C_2) - w_j^3(w_i C_1 - C_2)}{(w_i - w_j)w_k^3 - (w_i - w_k)w_j^3 + (w_j - w_k)w_i^3} \\ &= \frac{C_3(w_i - w_j) + w_i^2(w_j C_1 - C_2) - w_j^2(w_i C_1 - C_2)}{(w_i - w_j)w_k^2 - (w_i - w_k)w_j^2 + (w_j - w_k)w_i^2} \quad (12) \end{aligned}$$

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.

## V. 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 therefore propagate to large areas. The case with two errors are discussed in detail and is then shown how to extend to  $t > 2$  errors.

### A. Soft Errors Modeling

For temporal multiple soft errors, Luk et al. has proposed to cast soft error to an initial erroneous matrix to avoid considering the timing of soft error [18]. Soft error is treated as rank-one perturbation to the original matrix. Fitzpatrick et al. applied this method to double error modeling for Gaussian elimination [12]. We extended it to LU with partial pivoting using the round-off error resilient encoding method in section IV. This model is used in later sections for error detection and solution recovery.

LU factorization is viewed as multiplying a set of triangularization matrices from the left on 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  $s_2$ . 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  $\delta$  at location  $(i_2, j_2)$ .  $e_{i_2}$  and  $e_{j_2}$  are zero column vectors with 1s 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  $s_1$ ,

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

Let

$$\begin{aligned} f &= (L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2} \cdots L_{s_1}P_{s_1})^{-1}\delta e_{i_2}, \\ (L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2} \cdots L_{s_1}P_{s_1})^{-1}\hat{A}_{s_2} &= \hat{A}_{\bar{s}_2} \end{aligned}$$

Therefore,

$$\hat{A}_{\bar{s}_2} = \hat{A}_{s_1} - f e_{j_2}^T \quad (13)$$

Continue casting (13) to the soft error at step  $s_1$ :

$$\begin{aligned} & \hat{A}_{\bar{s}_2} = \hat{A}_{s_1} - f e_{j_2}^T \\ &= A_{s_1} - \lambda e_{i_1} e_{j_1}^T - f e_{j_2}^T \\ &= L_{s_1-1}P_{s_1-1}L_{s_1-2}P_{s_1-2} \cdots L_0P_0A_0 - \lambda e_{i_1} e_{j_1}^T - f e_{j_2}^T \end{aligned}$$

Let

$$\begin{aligned} d &= (L_{s_1-1}P_{s_1-1}L_{s_1-2}P_{s_1-2} \cdots L_0P_0)^{-1}\lambda e_{i_1}, \\ (L_{s_1-1}P_{s_1-1}L_{s_1-2}P_{s_1-2} \cdots L_0P_0)^{-1}\hat{A}_{\bar{s}_2} &= \hat{A}_{\bar{s}_1} \end{aligned}$$

And notice that

$$\begin{aligned} & (L_{s_1-1}P_{s_1-1}L_{s_1-2}P_{s_1-2} \cdots L_0P_0)^{-1} \times \\ & (L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2} \cdots L_{s_2}P_{s_2})^{-1} \\ &= (L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2} \cdots L_0P_0)^{-1} \end{aligned}$$

Let

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

And we have

$$\hat{A}_{\bar{s}_1} = A_0 - d e_{j_1}^T - g e_{j_2}^T \quad (14)$$

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

$$\hat{A}_0 = A_0 - \sum_{j=1}^t d_{j_i} e_{j_i}^T \quad (15)$$

### B. Errors Detection

With the model for soft errors, errors' locations can be determined for recovery. 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, however they can be protected by duplication and cross check, and the protection method for  $L$  in section IV can be directly applied to protect right hand sides.

In [12], four columns of checksum are used to locate two soft errors. Instead, we show that for  $N$  errors,  $N + 1$  columns are enough 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

$$G = \begin{bmatrix} e^T \\ w^T \\ (w^2)^T \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ w_1 & \cdots & w_N \\ w_1^2 & \cdots & w_N^2 \end{bmatrix} \quad (16)$$

and  $A$  is encoded as

$$[A, A \times G^T] = [A, Ae, Aw, Aw^2]$$

Note that the square 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 errors is equal to an error-free LU factorization to 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, (V-B) 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  $Ae$ :

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

Therefore

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

By the same token,

$$\begin{aligned} \hat{v} - \hat{U} w &= w_{j_1} \hat{L}^{-1} \hat{P} d + w_{j_2} \hat{L}^{-1} \hat{P} g \\ \hat{s} - \hat{U} w^2 &= w_{j_1}^2 \hat{L}^{-1} \hat{P} d + w_{j_2}^2 \hat{L}^{-1} \hat{P} g \end{aligned}$$

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:

$$\begin{aligned} (\hat{s} - \hat{U} w^2) - (w_{j_1} + w_{j_2})(\hat{v} - \hat{U} w) + \\ w_{j_1} w_{j_2} (\hat{c} - \hat{U} e) = 0 \end{aligned} \quad (17)$$

$w_{j_1}$  and  $w_{j_2}$  can be determined by iterating through all possible  $N \times (N - 1)$  combinations in  $w$  for a pair that makes (17) 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 (15), the check equation is:

$$\begin{cases} c_0 - \hat{U} w^0 &= w_{j_1}^0 x_1 + \dots + w_{j_t}^0 x_t \\ c_1 - \hat{U} w^1 &= w_{j_1}^1 x_1 + \dots + w_{j_t}^1 x_t \\ \vdots & \\ c_{t-1} - \hat{U} w^{t-1} &= w_{j_1}^{t-1} x_1 + \dots + w_{j_t}^{t-1} x_t \end{cases} \quad (18)$$

All powers in (18) are elementwise. This general case of check equation in vector form for  $t$  errors exhibits the same structure as in the scalar form. For  $t = 3$  it has been shown that check equation (12) 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 (17) which takes  $O(N)$  operations. For  $t > 2$ , the complexity to determine the error columns matches the complexity of LU factorization itself, making 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 (18) alone is insufficient for recovering the right factor  $U$  as only the columns of the initial errors can be determined. However for system of linear equations, by using Sherman-Morrison-Woodbury formula, the solution can be recovered.

## VI. COMPLEXITY REDUCTION

As the number of tolerable errors  $t$  increases, the complexity of locating initial error columns grows exponentially. To mitigate this issue and provide multiple error resilience capability with manageable overhead, this section offers complexity reduction methods for  $L$  and  $U$ .

### A. Reduction for $L$

As shown in (12), to tolerate three errors in a column of  $L$  of length  $N$ ,  $O(N^3)$  operations are required. Even though the search can be embarrassingly parallelized since each search path is independent of others, the overall complexity is still high when large  $t$  is desired.

In the complexity  $O(N^{t+1})$ ,  $N$  is the factor that determines the range of search. By breaking the search range into smaller segments and therefore reducing  $N$ , the complexity can be decreased to an affordable level.

There exist many ways of segmenting  $N$  but since each segment requires 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 equal 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

$$N^{1-\frac{1}{k}} \times (t+1) \quad (19)$$

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

$$\lim_{N \rightarrow \infty} (N^{1-\frac{1}{k}} \times (t+1) \times \frac{1}{N}) = \lim_{N \rightarrow \infty} \frac{t+1}{\sqrt[k]{N}} = 0 \quad (20)$$

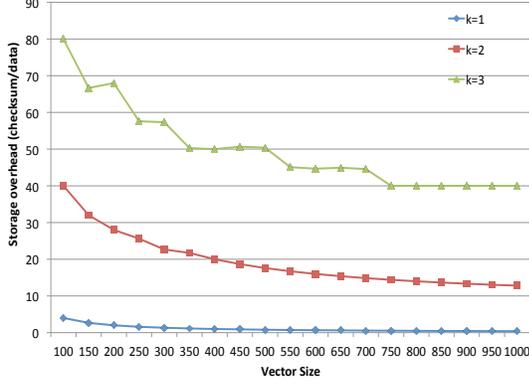


Figure 2. Storage overhead ( $t = 3$ )

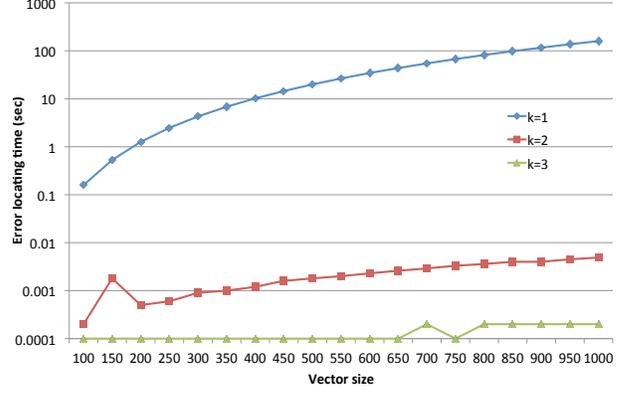


Figure 3. Error locating time ( $t = 3$ )

Based on the  $k_{th}$  root segmenting method, the error detection and recovery are performed following Algorithm 3 (Using  $t = 2$  as an example). Since the expensive error

---

**Algorithm 3** Error detection and recovery for one column  $l$  of  $L$

---

**Require:** Vector  $l$  of length  $N$ ; Segment length  $nb = \sqrt[k]{N}$ .  
**for**  $i = 1 \rightarrow N^{1-\frac{1}{k}}$  **do**  
    Using notation in (2),  
    In the  $i_{th}$  segment with elements  $l_1, \dots, l_{nb}$   
    **if**  $\|(l_1 + l_2 + \dots + l_{nb}) - c_1\| > 0$  **then**  
        Locate errors using (7)  
        Fix errors by solving the symptom equations in (4)  
    **end if**  
**end for**

---

locating procedure is now carried out within a smaller range, the complexity of error detection is largely reduced. The operation counts for Algorithm 3 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  $w_j$ . The total overhead of locating  $t$  errors in one segment is

$$\begin{aligned}
 & O(N^{\frac{1}{k}} \times N^{1-\frac{1}{k}}) + O((N^{\frac{1}{k}})^t) \\
 & = O(N) + O(N^{\frac{t}{k}}) = \begin{cases} O(N) & \text{if } t \leq k \\ O(N^{\frac{t}{k}}) & \text{if } t > k \end{cases}
 \end{aligned}$$

Note that the number of tolerable soft error  $t$  is for each segment. Therefore for large total number of tolerable errors per vector, each segment can select a smaller  $t$ , hence reducing error locating overhead. 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 (19). To evaluate the effect that different  $k$  plays on storage overhead and error locating time, a simulation is performed for  $t = 3$ . Figure 2 and 3 show the result. In vector case,  $t = 3$  is the smallest “forbidden case” since the complexity to locate errors is  $O(N^3)$ , already the same as that of LU

factorization. In this simulation, three errors are injected to the farthest end of input vectors, making it the worst case for error locating since all combinations of  $w_i$ ,  $w_j$  and  $w_k$  have to be tried against (12) before a match can be found.

Compare the storage overhead and error locating time, when  $k = 3$ , checksum uses the most (nearly 40%) extra storage while finds error in less than 0.001 seconds, while  $k = 2$  only requires slightly over 10% extra storage but still achieves over  $10^4$  speed up to locate errors at large sizes. When  $k > 3$  the storage overhead becomes unaffordable with little improvement in error locating speed. Therefore  $k = 2$  is a fair choice compromising both storage overhead and error locating speed, and the complexity when  $t = 3$  and  $k = 2$  is  $O(N^{\frac{3}{2}}) < O(N^3)$ .

### B. 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 VI-A is extended to apply on blocked LU algorithm for  $\bar{U}$  and  $\tilde{U}$  protection.

1) *Block Encoding of Matrix:* In blocked LU algorithm, panel factorization itself is an LU factorization of a tall and skinny panel, therefore the encoding technique in V can be used to protect a panel or several panels too if the encoding is performed accordingly.

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

*Proof:* Given a matrix  $A$  of size  $N \times N$  and generator matrix  $G$ . Split  $A$  into equally sized block  $N_k \times N_k$  and let  $G$  have size  $N_k \times (t + 1)$ , where  $t$  is the number of errors tolerable by  $G$ . Matrix  $A$  is encoded as:

$$\left[ \begin{array}{c|ccc|ccc}
 A_{11} & A_{12} & \cdots & A_{1n} & A_{11}G & \cdots & A_{1n}G \\
 A_{21} & A_{22} & \cdots & A_{2n} & A_{21}G & \cdots & A_{2n}G \\
 \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
 A_{n1} & A_{n2} & \cdots & A_{nn} & A_{n1}G & \cdots & A_{nn}G
 \end{array} \right]$$

Start by performing one iteration of LU factorization for the first panel of block of size  $N \times N_k$ , generating  $U_{11}$  and  $L_{i,1}$  where  $2 < i < n$ . Then perform triangular solving and trailing matrix update making the encoded matrix into state:

$$\left[ \begin{array}{c|ccc|ccc} L_{11} \backslash U_{11} & U_{12} & \cdots & U_{1n} & C_{11} & \cdots & C_{1n} \\ \hline L_{21} & \tilde{A}_{22} & \cdots & \tilde{A}_{2n} & C_{21} & \cdots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ L_{n1} & \tilde{A}_{n2} & \cdots & \tilde{A}_{nn} & C_{n1} & \cdots & C_{nn} \end{array} \right] \quad (21)$$

Note that similar to ScaLAPACK storage format, the lower triangular blocks  $L_{i,1}, 2 < i < n$  are stored in the zeroed out area in the first panel.

According to (1), we have

$$\left\{ \begin{array}{l} U_{1j} = L_{11}^{-1} A_{1j} \\ C_{1j} = L_{11}^{-1} A_{1j} G \quad i = \{2, \dots, n\} \\ \tilde{A}_{ij} = A_{ij} - L_{i1} \times U_{1j} \quad j = \{1, \dots, n\} \\ C_{ij} = C_{ij} - L_{i1} \times C_{1j} \end{array} \right.$$

$$\therefore \tilde{C}_{ij} = A_{ij} G - L_{i1} L_{11}^{-1} A_{1j} G$$

$$= (A_{ij} - L_{i1} U_{1j}) G = \tilde{A}_{ij} G$$

Similar method can be used in the rest iterations.  $\blacksquare$

As an example, take a matrix  $A$  of  $2 \times 2$  blocks encoded using the generator in (16):

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

Carry out LU factorization to  $A_c$  and we have:

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

And  $C_1$  to  $C_4$  can be calculated as:

$$\begin{cases} C_{11} = U_{11}G, & C_{12} = U_{12}G \\ C_{21} = \emptyset, & C_{22} = U_{22}G \end{cases} \quad (22)$$

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 (16) 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 V-B is then activated for this  $\sqrt{N} \times \sqrt{N}$  block.

The complexity of performing blocked error detection and locating includes the error check that is either 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  $1 + 2 + \dots + N^{1-\frac{1}{k}}$  number of blocks. Therefore the error checking complexity is

$$(1 + 2 + \dots + N^{1-\frac{1}{k}}) \times O((N^{\frac{1}{k}})^2)$$

$$= \frac{N^{1-\frac{1}{k}}(N^{1-\frac{1}{k}} + 1)}{2} \times O(N^{\frac{2}{k}})$$

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 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

$$\lim_{N \rightarrow \infty} (N \times N^{1-\frac{1}{k}} \times (t + 1) \times \frac{1}{N^2}) = \lim_{N \rightarrow \infty} \frac{t + 1}{\sqrt{k}N} = 0$$

Similar to the scalar case in VI-A, compromise has to be made between  $t$  and  $k$  for number of error tolerated and storage overhead for checksum. Following the evaluation in Figure 3 and 2,  $t = k = 2$  is chosen for the experiments in this work.

2) *Reduction of ABFT Extra Flops*: The additional ABFT checksum columns to protect  $U$  participate in the trailing matrix update ( $L_{22}U_{22} = A_{22} - L_{21}U_{12}$  in (1)) of LU factorization, and since trailing matrix update takes up over 90% of the floating point operations (FLOPS) of LU, extra FLOPS from the additional checksum columns contributes significant overhead even if no errors occur at all. Block encoding in section VI-B1 helps remove the error locating overhead, but in fact it also offers an insight to lower the error-free overhead.

In (22), encoding is performed within blocks  $[A_{11}, A_{21}]^T$  and  $[A_{12}, A_{22}]$  separately. For block  $A_{11}$ , the checksum  $C_{11}$ 's relationship with  $U_{11}$  by  $C_{11} = U_{11} \times G$  is established when panel  $[A_{11}, A_{21}]^T$  is factorized. After this point,  $C_{11}$  are not subject to any further change and  $C_{21}$  remains zero even though further operations (triangular solve with  $C_{21}$  as right hand sides) are applied. The invariance of  $C_{11}$  and  $C_{21}$  after the first panel factorization indicates that  $[C_{11}, C_{21}]^T$  can be excluded from any later operations.

*Corollary 6.2*: After each step of LU factorization, one panel of checksum columns corresponding to the panel being factorized in this step can be excluded from further operation.

*Proof*: In (21),  $U_{12}$  does not participate in any further operation because the next iteration starts from  $\tilde{A}_{22}$  to the

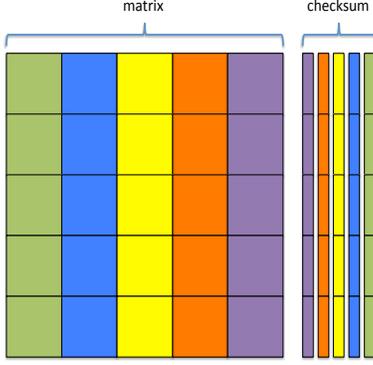


Figure 4. Checksum layout example of a  $5 \times 5$  blocks matrix

bottom-right corner of the encoded matrix.

$$\begin{aligned} \therefore L_{i1} &= A_{i1} \times U_{11}^{-1}, \quad i = [2, \dots, n] \\ \therefore C_{i1} &= A_{i1}G - L_{i1}C_{11} = A_{i1}G - L_{i1}U_{11}G \\ &= L_{i1}U_{11}G - L_{i1}U_{11}G = \emptyset \end{aligned}$$

Since  $C_{21}$  is used as the right hand sides of the triangular solve in the next iteration, which produces  $\emptyset$  as result too, after the trailing matrix update of the next iteration,  $C_{21}, \dots, C_{n1}$  are all still  $\emptyset$ . Therefore the panel  $[C_{11}, \dots, C_{n1}]$  are not subject to further change nor does it contribute to any factorization result, hence this panel can be excluded. In the next iteration, the actively participating data is

$$\left[ \begin{array}{c|ccc|ccc} L_{11} \setminus U_{11} & U_{12} & \cdots & U_{1n} & C_{12} & \cdots & C_{1n} \\ \hline L_{21} & \tilde{A}_{22} & \cdots & \tilde{A}_{2n} & C_{22} & \cdots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ L_{n1} & \tilde{A}_{n2} & \cdots & \tilde{A}_{nn} & C_{n2} & \cdots & C_{nn} \end{array} \right]$$

By the same process, in each iteration the panel of checksum columns that corresponds to the just factorized matrix can be left out of further operation. ■

In order to benefit from the complexity reduction of Corollary 6.2, the layout of checksum columns is reversed horizontally. Figure 4 shows an example of such design. The block size is the  $N_k$ . Each  $N_k \times N_k$  block has a  $N_k \times t$  block of checksum. The checksum blocks are labelled with the same color as the data they serve, for example the green blocks on the right end protects the green data blocks on the left end. This layout makes it easy to implement the complexity reduction in ScaLAPACK PDGESV by simply reducing the scope of PDTRSM and PDGEMM. When panel factorization finishes the green data blocks, the green checksum blocks are no longer touched in coming iterations and therefore the extra FLOPS of updating the green blocks are eliminated. The same process continues with each checksum panels till the end of the factorization.

## VII. 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. [12] suggested using Sherman-Morrison-Woodbury formula. Here we first review the recovery procedure and then analyze the computational complexity.

### A. Correction for $x$

As shown in Algorithm 1, factorization result  $\hat{L}$  and  $\hat{U}$  are used to compute the solution  $\hat{x}$  even if the factorization has been subject to soft errors. The solution  $x$  is corrected later from  $\hat{x}$  when errors are detected.

From  $Ax = b$ , we have

$$\begin{aligned} x &= A^{-1}b = A^{-1}(\hat{P}^{-1}\hat{P})b \\ &= (\hat{P}A)^{-1}\hat{P}b \end{aligned} \quad (23)$$

Both  $\hat{P}$  and  $b$  are known, so  $(\hat{P}A)^{-1}$  is needed for  $x$ .

From (14), the erroneous initial matrix  $\hat{A}_{s_1}$  differs from the real initial matrix  $A_0$  by column  $j_1$  and  $j_2$ , therefore

$$\begin{aligned} \hat{P}A - \hat{P}\hat{A} &= (\hat{P}a_{\cdot j_1} - \hat{L}\hat{U}_{\cdot j_1})e_{j_1}^T + (\hat{P}a_{\cdot j_2} - \hat{L}\hat{U}_{\cdot j_2})e_{j_2}^T \\ \therefore \hat{P}A &= \hat{L}\hat{U} + (\hat{P}a_{\cdot j_1} - \hat{L}\hat{U}_{\cdot j_1})e_{j_1}^T + (\hat{P}a_{\cdot j_2} - \hat{L}\hat{U}_{\cdot j_2})e_{j_2}^T \\ &= \hat{L}\hat{U} + \hat{L}(\hat{L}^{-1}\hat{P}a_{\cdot j_1} - \hat{U}_{\cdot j_1})e_{j_1}^T + \hat{L}(\hat{L}^{-1}\hat{P}a_{\cdot j_2} - \hat{U}_{\cdot j_2})e_{j_2}^T \end{aligned}$$

Let  $t_{j_1} = \hat{L}^{-1}\hat{P}a_{\cdot j_1} - \hat{U}_{\cdot j_1}$ , and  $t_{j_2} = \hat{L}^{-1}\hat{P}a_{\cdot j_2} - \hat{U}_{\cdot j_2}$ ,

$$\therefore \hat{P}A = \hat{L}\hat{U}(I + \hat{U}^{-1}t_{j_1}e_{j_1}^T + \hat{U}^{-1}t_{j_2}e_{j_2}^T)$$

Let  $v_{j_1} = \hat{U}^{-1}t_{j_1}$  and  $v_{j_2} = \hat{U}^{-1}t_{j_2}$ ,

$$\begin{aligned} \therefore \hat{P}A &= \hat{L}\hat{U}(I + v_{j_1}e_{j_1}^T + v_{j_2}e_{j_2}^T) \\ &= \hat{L}\hat{U}(I + [v_{j_1} \quad v_{j_2}] [e_{j_1} \quad e_{j_2}]^T) \end{aligned}$$

Let  $U_x = [v_{j_1} \quad v_{j_2}]$ ,  $V_x = [e_{j_1} \quad e_{j_2}]$ ,

$$\therefore (\hat{P}A)^{-1} = (I + U_x V_x^T)^{-1}(\hat{L}\hat{U})^{-1} \quad (24)$$

Apply the Sherman-Morrison-Woodbury formula [31], [32] to (24):

$$\begin{aligned} x &= (\hat{P}A)^{-1}\hat{P}b \\ &= (I - U_x(I + V_x U_x^{-1} V_x^T)(\hat{L}\hat{U})^{-1}\hat{P}b \\ &= (I - U_x(I + V_x^T U_x)^{-1} V_x^T)\hat{x} \end{aligned} \quad (25)$$

Hence the correct solution  $x$  can be corrected from  $\hat{x}$ .

### B. Computation Complexity

At the center of computing the correct solution is

$$U_x(I + V_x^T U_x)^{-1} V_x^T$$

For  $t$  errors,  $V_x^T U_x$  produces a  $t \times t$  matrix.  $t$  is normally selected as small integers such as 2 or 3 for the protection from flips in  $2 \times$  or  $3 \times 64$  bits, hence the inverse of  $t \times t$  can be solved directly. For example, when  $t = 2$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

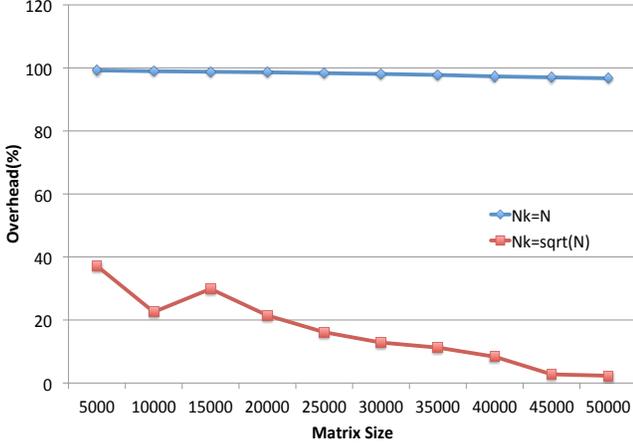


Figure 5. Overhead comparison result on Kraken (16 × 16 grid)

resulting in eight FLOPS. And since  $V_x$  is filled with 0s except the two 1s at row  $j_1$  and  $j_2$  of column one and two respectively, four FLOPS are needed to generate  $V_x^T U_x$  and  $I$  plus the result of  $V_x^T U_x$ , each. Let  $Y = (I + V_x^T U_x)^{-1}$ , similarly due to the sparsity of  $V_x$ ,  $Y \times V_x^T$  also requires four FLOPS. Let  $Z = Y \times V_x^T$ , compute  $Z \times \hat{x}$  yields a  $2 \times 1$  matrix costing six FLOPS, and at last  $4 \times N$  FLOPS are paid to update the solution on  $\hat{x}$ . In summary,  $O(N)$  overhead is required to calculate (25).

Another part of operation overhead comes from computing  $U_x$ , namely  $v_{j_1} \cdots v_{j_t}$ . Each of these vectors takes  $O(N^2)$  to compute by PDTRSM with  $\hat{U}$ , and also  $O(N^2)$  to generate the  $t$  right hand side vectors from  $t_{j_k} = \hat{L}^{-1} \hat{P} a_{\cdot j_k} - \hat{U}_{\cdot j_k}$ ,  $k \in [1 \cdots t]$  for PDTRSM. Therefore, to tolerate up to  $t$  soft errors, with  $t$  being a constant,  $O(N^2)$  is the computation complexity for the recovery of  $x$  from  $\hat{x}$ .

## VIII. PERFORMANCE EVALUATION

Soft errors in the left factor are static and the detection and recovery in this area has been validated in [9], [10] showing the scalability and small performance impact to the host algorithm. The algorithms for multiple soft errors in the right factor, 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.

The experiments are carried 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  $\sqrt{N}$ .

Figure 5 shows the effectiveness of the complexity reduction method for  $U$  with a  $16 \times 16$  process grid on Kraken, and  $t = 2$ . 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 equation. 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 for size 50,000.

Figure 6 shows the performance of different matrix sizes on Kraken using 16,384 cores in a  $128 \times 128$  grid. As the matrix becomes larger, both the original ScaLAPACK PDGESV and fault tolerant PDGESV with and without errors exhibit close performance. At the largest size 1000,000, the non-error case adds roughly 1.1% overhead, and with error correction the overhead increases to 1.3%.

Figure 7 is the weak scalability experiment result where both matrix size and grid dimension are doubled. Throughout all the testing sizes from 64 to 16,384 cores, FT-PDGESV declares around 1% overhead for both with and without errors cases.

From the result in experiments, it can be confirmed that the complexity of recovering the solution to  $Ax = b$  from double 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.

## IX. RELATED WORK

In the field of fault tolerance for HPC systems, checkpoint-restart (C/R) is the most commonly used method [1]. 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 often has high overhead from checkpointing through disk I/O.

To reduce the overhead, diskless checkpointing [26] turns to system memory for checksum storage rather than disks. It has seen good applications such as FFT [11] and matrix factorizations [25].

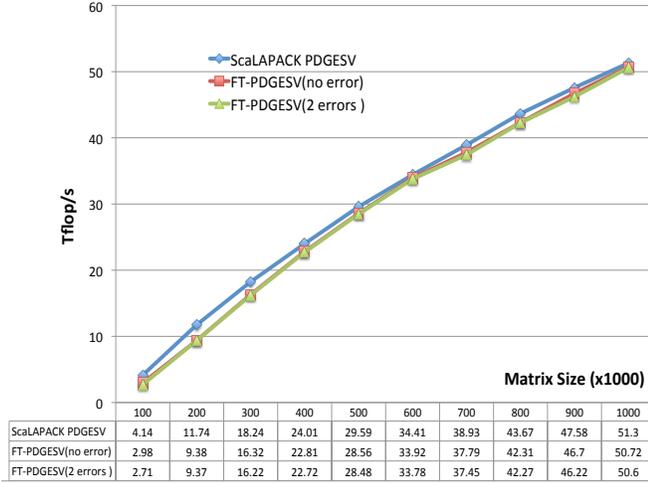


Figure 6. Result on Kraken with 16,384 ( $128 \times 128$ ) cores

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 [2], [17]. 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 [18], [19] Later, this was extended to multiple errors [4], [12], [24] by adopting methodology from finite-field based error correcting code (Reed-Solomon [27], BCH [6], [16], etc.) where only the right factor of factorization result is protected and computation is assumed to take place with exact arithmetics. In reality, soft error could strike any area of matrix and modern HPC systems use floating point operations which produce round-off error. With the presence of round-off error, the BCH code based error location determination method no longer produces exact number as location, which renders the result ambiguous and therefore useless.

Recently, iterative solvers were evaluated for soft error vulnerability [7], [15], [21], signifying the recent 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 [8], 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

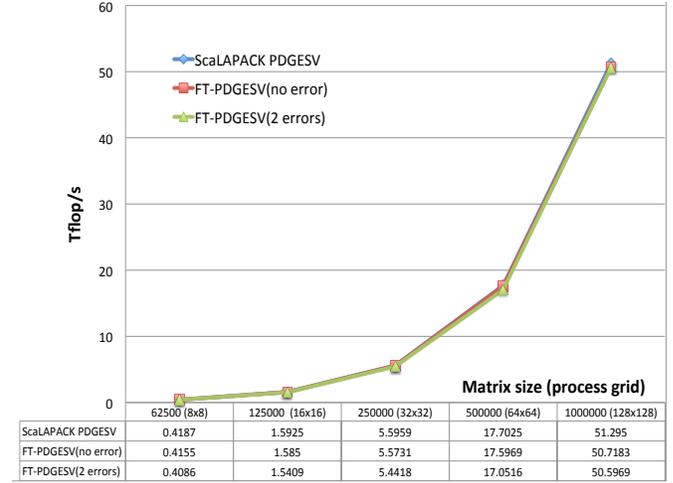


Figure 7. Weak scalability result on Kraken

shown in [9] with implementation on large scale distributed memory system, and the recovery of matrix factorization was proposed in [10] using QR for demonstration.

## X. 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. The proposed method can be extended to the protection of dense matrix factorizations like LU and QR. Further research also includes hardening the implementation for the case where soft errors strike during the detection and recovery process.

## REFERENCES

- [1] Fault tolerance for extreme-scale computing workshop report, 2009.
- [2] J. Abraham. Fault tolerance techniques for highly parallel signal processing architectures. *Highly parallel signal processing architectures*, pages 49–65, 1986.
- [3] D. Abts, J. Thompson, and G. Schwoerer. Architectural support for mitigating dram soft errors in large-scale supercomputers.
- [4] C. Anfinson and F. Luk. A linear algebraic model of algorithm-based fault tolerance. *Computers, IEEE Transactions on*, 37(12):1599–1604, 1988.
- [5] R. Barrett, T. Chan, E. D’Azevedo, E. Jaeger, K. Wong, and R. Wong. Complex version of high performance computing linpack benchmark (hpl). *Concurrency and Computation: Practice and Experience*, 22(5):573–587, 2010.

- [6] R. Bose and D. Ray-Chaudhuri. On a class of error correcting binary group codes\*. *Information and control*, 3(1):68–79, 1960.
- [7] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 155–164. ACM, 2008.
- [8] G. Bronevetsky, B. de Supinski, and M. Schulz. A Foundation for the Accurate Prediction of the Soft Error Vulnerability of Scientific Applications. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2009.
- [9] P. Du, P. Luszczek, and 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] P. Du, P. Luszczek, S. Tomov, and J. Dongarra. Soft error resilient QR factorization for hybrid system. Technical Report 252, LAPACK Working Note, July 2011.
- [11] E. Elnozahy, D. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on*, pages 39–47. IEEE, 1991.
- [12] P. Fitzpatrick and 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*, pages 469–480. IEEE, 1992.
- [13] A. Gonzalez, S. Mahlke, S. Mukherjee, R. Sendag, D. Chiou, and J. Yi. Reliability: Fallacy or reality? *Micro, IEEE*, 27(6):36–45, 2007.
- [14] P. Hazucha and C. Svensson. Impact of cmos technology scaling on the atmospheric neutron soft error rate. *Nuclear Science, IEEE Transactions on*, 47(6):2586–2594, 2000.
- [15] V. Heuveline, D. Lukarski, F. Oboril, M. Tahoori, and J. Weiss. Numerical defect correction as an algorithm-based fault tolerance technique for iterative solvers.
- [16] A. Hocquenghem. Codes correcteurs derreurs. *Chiffres*, 2(2):147–56, 1959.
- [17] K. Huang and J. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6):518–528, 1984.
- [18] F. Luk and H. Park. An analysis of algorithm-based fault tolerance techniques\* 1. *Journal of Parallel and Distributed Computing*, 5(2):172–184, 1988.
- [19] F. Luk and H. Park. Fault-tolerant matrix triangularizations on systolic arrays. *Computers, IEEE Transactions on*, 37(11):1434–1438, 1988.
- [20] D. Lyons. Sun screen, November 13 2000. Available at <http://www.forbes.com/forbes/2000/1113/6613068a.html>.
- [21] K. Malkowski, P. Raghavan, and M. Kandemir. Analyzing the soft error resilience of linear solvers on multicore multiprocessors. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE.
- [22] H. W. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon. *TOP500 Supercomputer Sites*, 36<sup>th</sup> edition, November 2010. (The report can be downloaded from <http://www.netlib.org/benchmark/top500.html>).
- [23] S. Michalak, K. Harris, N. Hengartner, B. Takala, and 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):329–335, 2005.
- [24] H. Park. On multiple error detection in matrix triangularizations using checksum methods. *Journal of Parallel and Distributed Computing*, 14(1):90–97, 1992.
- [25] J. Plank, Y. Kim, and J. Dongarra. Algorithm-based diskless checkpointing for fault-tolerant matrix operations. In *ftcs*, page 0351. Published by the IEEE Computer Society, 1995.
- [26] J. Plank, K. Li, and M. Puening. Diskless checkpointing. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):972–986, 1998.
- [27] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.
- [28] B. Schroeder, E. Pinheiro, and 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*, pages 193–204. ACM, 2009.
- [29] A. Tipton, J. Pellish, R. Reed, R. Schrimpf, R. Weller, M. Mendenhall, B. Sierawski, A. Sutton, R. Diestelhorst, G. Espinel, et al. Multiple-bit upset in 130 nm cmos technology. *Nuclear Science, IEEE Transactions on*, 53(6):3259–3264, 2006.
- [30] M. White, J. Qin, and J. Bernstein. A study of scaling effects on dram reliability. In *Reliability and Maintainability Symposium (RAMS), 2011 Proceedings-Annual*, pages 1–6. IEEE.
- [31] M. Woodbury. The stability of out-input matrices. *Chicago, IL*, 1949.
- [32] M. Woodbury. Inverting modified matrices. *Memorandum report*, 42:106, 1950.