High Performance Dense Linear System Solver with Soft Error Resilience

Peng Du[*], Piotr Luszczek[*], Jack Dongarra[*†‡]
[*]Electrical Engineering and Computer Science Department, University of Tennessee, Knoxville
[†]Oak Ridge National Laboratory
[‡]University of Manchester UK

***Abstract*—As the scale of modern high end computing systems continues to grow rapidly, system failure has become an issue that requires a better solution than the commonly used scheme of checkpoint and restart (C/R). While hard errors have been studied extensively over the years, soft errors are still understudied especially for modern HPC systems, and in some scientific applications C/R is not applicable for soft error at all due to error propagation and lack of error awareness. In this work, we propose an algorithm based fault tolerance (ABFT) high performance dense linear system solver with soft error resilience. By adopting a mathematical model that treats soft error during LU factorization as rank-one perturbation, the solution of $Ax = b$ can be recovered with Sherman-Morrison formula. Our contribution includes extending error model from Gaussian elimination and pairwise pivoting to LU with partial pivoting, and we provide a practical numerical bound for error detection and a scalable checkpointing algorithm to protect the left factor that is needed for recovering x from soft error. Experimental results on cluster systems with ScaLAPACK show that the fault tolerance functionality adds little overhead to the linear system solving and scale well on modern cluster systems.**

I. Introduction

A soft error or Silent Data Corruption (SDC) [1] occurs when data in a computing system is contaminated without the error being detected. The source of soft error includes temperature and voltage fluctuations, cosmic particles, electrostatic discharge, etc. Soft error becomes more prevalent with new technologies such as higher clock frequencies, transistor density, and lower voltage [2]. Management of such errors has been argued strongly especially in the context of high-end systems [3].

Unlike soft errors, hard errors normally bring down a part of the system, for instance a node, and provide direct notification to the application about the nature of the error as well as logistical details such as the time and location of the error. The real challenge that soft errors bring to computing is that soft error takes place silently during computation, and application ends up producing wrong, or at the very least inaccurate, results. When this happens to a large scale system, computing error caused by the hardware-related soft error normally propagates throughout the application data which leads to practically unrecoverable state with an incorrect result. It also wastes valuable (and expensive) computing hours and takes a long time and requires a huge effort to trace the cause of error because no failure record can be found. Recently soft error is also reported to be happening in GPGPU computing [4] which has been widely deployed in scientific application. For instance, China's Tianhe-1A that ranked number one on the November 2010 TOP500 list [5] uses $7,168$ NVIDIA Tesla M2050 GPGPUs to achieve 2.57 Pflop/s in the context of the High-Performance Linpack Benchmark (HPL) benchmark. HPL is a software package that solves a random generated, dense linear system of linear equations in double precision arithmetic on distributed-memory parallel computers. Even though the newer generation of GPUs from NVIDIA features the Error Correction Codes (ECC) technology, when performing computationally intensive calculations (such as HPL), the feature is usually turned off to avoid the associated performance penalty [6]. This subjects GPUs to soft errors just as the technology starts being used as the critical enabler of high performance scientific computing.

For top machines on the TOP500 list, it is not unusual that running time spans more than 24 hours, more if the time to optimize and prepare for the running [7] is included. This leaves these machines into a large time window for being affected by soft error. Therefore, it is important to equip large scale computing system with soft error resilience.

Among software packages for scientific applications, ScaLAPACK [8], an extension of LAPACK [9] for distributed memory systems based on MPI or PVM, is designed for solving dense linear algebra problem like systems of linear equations, least squares problems, and eigenvalue problems. The algorithm used by HPL is based on the right-looking LU factorization implemented in ScaLAPACK. The complexity of LU factorization implementation makes it hard to be protected from soft error because the recursive manor of the algorithm touches large areas of data and therefore causes error propagation and results in large computing error even from a single bit flip. While traditional checkpointing recovery mechanisms for fault tolerance are not suitable for soft error, algorithm based fault tolerance (ABFT) [10] has been shown to be promising with LU factorization on systolic arrays system. In this paper, based on the previous work, we devise an ABFT based soft error detection and protection mechanism for the LU-based dense linear system solver in ScaLAPACK. This mechanism can detect the occurrence of soft errors and recover the linear system solution with little overhead, and the same method can be easily applied to other factorization routines like Cholesky and QR.

The rest of the paper is organized as follows. Section II talks about related work. Section III introduces the LU factorization

based linear solver and how soft error affects the final result. Section IV gives the soft error model. Section V and VI establish the ABFT algorithm for the right factor while section VII provides solution to the protection of the left factor. Section VIII shows performance evaluation and experiment result, and section IX concludes the work.

II. Related Work

For parallel applications, checkpoint-restart(C/R) is the most commonly used method for fault tolerance [11]. At the application and system level, the running state of the application is dumped to reliable storage at a certain interval, either by the message passing middleware automatically or at the request of user application. C/R requires the least user intervention, but suffers from high checkpointing overhead when writing data to stable storage.

To reduce the overhead, diskless checkpointing [12] has been proposed to use system memory for checksum storage rather than disk storage. Diskless has seen good applications such as FFT [13] and matrix factorizations [14]. Diskless checkpointing is suitable for applications that modify small amounts of memory between checkpoints.

Both C/R and diskless checkpointing need the error information for recovery, and no such information is guaranteed with soft error. Algorithm based fault tolerance (ABFT) offers such merit that no periodical checkpointing is necessary. This eliminates checkpointing overhead, and checksum during computing could reflect the most current status of the data which harbors clues for soft error detection and recovery. Algorithm-based fault tolerance (ABFT) was originally introduced to deal with silent error in systolic arrays [15], [16]. ABFT encodes data once before computation. Matrix algorithms are designed to work on the encoded checksum along with matrix data, and the correctness is checked after the matrix operation completes.

ABFT for matrix factorization was explored back in the 1980s [17], [18] for single error, which was later extended to multiple errors [19], [20], [21] by adopting methodology from error correcting code. These methods for systolic arrays offers promising direction, but requires modification in both algorithm and implementation to be used on the dense linear system solver on distributed memory machines. Lately, iterative solvers were evaluated for soft error vulnerability [22], [23] for sparse matrix system, and this shows the recent awareness of soft error for solving large scale problem. For dense matrix, the effect of soft error 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 error patterns of various routines is irrelevant with problem sizes. Also, the possibility of predicting the fault propagation are explored.

III. High Performance Linear System Solver

For dense matrix A, the LU factorization gives $PA = LU$ (or $P = ALU$), where P is 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

$$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

ScaLAPACK implements the right-looking version of LU with partial pivoting based on block algorithm and 2D block cyclic data distribution.

A. Block LU Algorithm and 2D Block Cyclic Distribution

It has been well established that block algorithm and data layout plays an important role in the performance of parallel matrix operations on distributed memory systems [25], [26], [27]. Without loss of generality, matrix A is split into 2×2 blocks and decomposed 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

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

This poses as one iteration of the factorization, and pivoting is applied on the left and right of the current panel. ScaLAPACK LU uses PDGETF2, PDTRSM and PDGEMM for the three steps and PDLASWP for pivoting. “D” in routine names indicates double precision.

Block algorithm offers good granularity to benefit from high performance BLAS routines, while 2D block-cyclic distribution helps with load balancing and therefore ensures scalability.

In 2D block-cyclic distributions, data is divided into equally sized blocks, and all computing units are organized into a virtual two-dimension grid P by Q. Each data block is distributed to computing units in round robin following the two dimensions of the virtual grid. This layout reduces data communication frequency since, in each step of the algorithm, as many computing units can be engaged in computations and most of the time, each computing unit only works on its local data. Fig. 1 is an example of $P = 2, Q = 3$ and a global matrix of 4×4 blocks. The same color represents the same process and numbering in A_{ij} indicates the location in the global matrix. Mapping algorithm between local blocks and their global locations can be found in [28].

When applied to an ABFT algorithm with 2D block-cyclic distribution, checksum that is generated before the factorization, and put conceptually on the right of the matrix, needs to be explicitly located. We assume that, during computation, matrix data and checksum are equally susceptible to soft error. Therefore, checksum is put along with data matrix without using additional “reliable” processes.

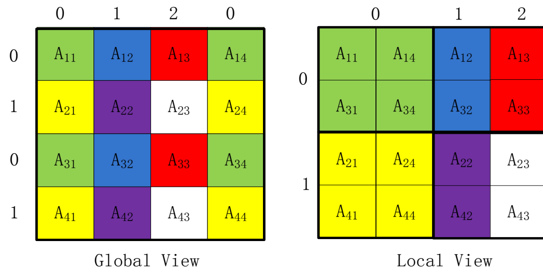

Fig. 1: Example of 2D block-cyclic data distribution

B. Error Propagation

Radiation induced soft error strikes at random moment and area, normally in the form of bit flip. For algorithms like LU factorization, errors caused by bit flip are carried along with computation.

In ScaLAPACK, the left factor L is stored in-place at the lower triangular of the original matrix except the 1's on the diagonal, and the right factor U takes over the upper triangular including diagonals at the end of the factorization. Once an iteration finishes, the lower triangular L and upper triangular U that are finished till this iteration do not participate any future operation and therefore that soft error that occurred after this moment in these area does not propagate except being moved vertically by pivoting. Soft error in the trailing area, namely U_{12}, L_{21} and A_{22} in Equation 1, propagates into clean areas.

Fig. 2 is a demonstration of such a situation. Two LU factorizations of the same data are run. One with errors and one without error. The matrix size is 200×200 with block size 20. The two final results are subtracted and colored by the size of the absolute value of the residue. The brighter the color, the larger the residue. Using MATLAB notation, two soft errors are injected at location $(50,120)$ and $(35,10)$ right before the panel factorization for blocks $(41:200, 41:60)$ starts. Error at $(35,10)$ is in the finished L area and therefore does not propagate. Error at $(50,120)$ is in the PDTRSM area. During PDTRSM, data in column 120 gets affected and this column of errors continues into the PDGEMM area (the trailing matrix for step 40) until PDGETF2 starts on blocks $(100:200, 100:121)$ when errors spread out to the whole trailing matrix $(120:200, 120:200)$. It is worth noting that errors on the diagonals also cause pivoting sequence to divert from the correct sequence, and this affect the areas below row 120 of L.

From the example, it can be seen that large areas of the final L and U can be contaminated by a single soft error, and the affected area is a function of the soft error location and moment of injecting, which is unknown beforehand. Available fault tolerance, like C/R and diskless checkpointing, are not applicable because they require the location and time information of error, and by the end of the factorization the error could have propagated into their checksum and invalidated the redundancy for recovery.

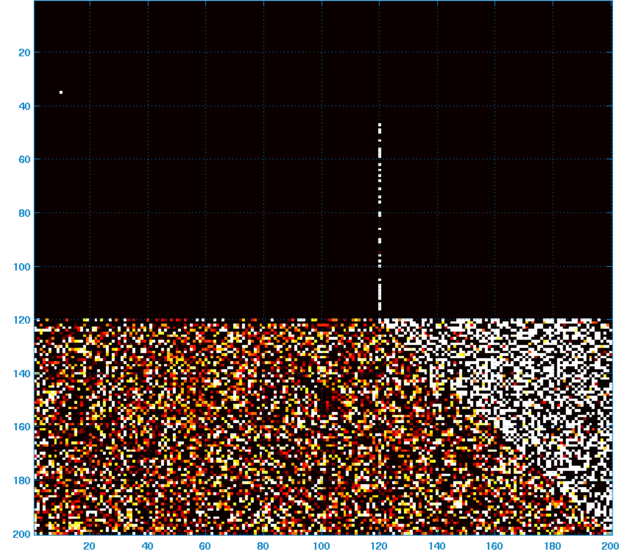

Fig. 2: Error propagation

IV. Soft Error Modeling

Given the feature of soft error propagation in LU factorization, we adopted the error modeling technique proposed in [17], which was designed for soft error in LU factorization with pairwise pivoting on systolic arrays and extended to ScaLAPACK LU.

Soft error is treated as rank-one perturbation to the original matrix prior to factorization. LU factorization is viewed as matrix multiplied from the left by a set of triangularization matrices to get the final triangular form. Let

$$A_0 = A$$

and

$$A_{t+1} = L_t P_t A_t$$

P_t is the partial pivoting matrix at step t. At step t, error occurs at random location (i, j) in matrix A as

$$\begin{aligned} \tilde{A}_t &= L_t P_t A_t - \lambda e_i e_j^T \\ &= (L_{t-1} P_{t-1} \ldots L_0 P_0) A_0 - \lambda e_i e_j^T \end{aligned} \tag{2}$$

e_i is a column vector with all 0 elements except 1 as the ith element. Continuing factorization from step t equals to starting factorization from the initial error matrix

$$\tilde{A} = A - d e_j^T$$

where $d = \lambda (L_{t-1} P_{t-1} \ldots L_0 P_0)^{-1} e_i$

And at the end of the factorization,

$$PA_0 = LU$$

where U is upper triangular matrix.

In essence, this model treats soft error as a perturbation to the initial matrix similar to rounding errors so that backward error analysis [29] can be used for designing the recovery algorithm.

V. Checksum for the Right Factor

Since the upper triangular matrix U undergoes changes during each iteration of the LU factorization, static checkpointing is not suitable for recovery. For HPL, algorithm based fault tolerance has been shown for fail-stop failure [30], but without the knowledge of error location, the same method cannot be directly applied to soft error recovery. Instead, We chose the method proposed in [18] which was also designed for soft error in systolic arrays.

To capture one error, for input matrix $A \in \mathbb{R}^{m\times n}$, two generator matrices are used, $e = (1,1,\ldots,1)$ and a random matrix w. $e,w \in \mathbb{R}^{m\times 1}$.

LU factorization is appended with checksum by e and w on the right as

$$P(A\ Ae\ Aw) = L(U\ c\ v) \tag{3}$$

$c,v \in \mathbb{R}^{m\times 1}$ are checksum after factorization. To prevent pivoting from rotating checksum into matrix, no column-wise checksum is taken.

At the time or error, A becomes the erroneous matrix $\tilde{A}$, and checkpointed matrix becomes

$$(\tilde{A}\ Ae\ Aw)$$

And the LU factorization becomes

$$\tilde{P}(\tilde{A}\ Ae\ Aw) = \tilde{L}(\tilde{U}\ \tilde{c}\ \tilde{v}) \tag{4}$$

From eq. 4

$$\begin{aligned} \tilde{c} &= \tilde{L}^{-1}\tilde{P}Ae = \tilde{L}^{-1}\tilde{P}(\tilde{A}+de_j^T)e \\ &= \tilde{L}^{-1}(\tilde{L}\tilde{U}+de_j^T)e \\ &= \tilde{U}e+\tilde{L}^{-1}de_j^Te = \tilde{U}e+\tilde{L}^{-1}d \end{aligned}$$

By the same token,

$$\tilde{v} = \tilde{U}w + w_j\tilde{L}^{-1}d$$

Assume residual vectors $r,s \in \mathbb{R}^{m\times 1}$

$$\tilde{r} = \tilde{c} - \tilde{U}e = \tilde{L}^{-1}d \tag{5}$$

and

$$\tilde{s} = \tilde{v} - \tilde{U}w = w_j\tilde{L}^{-1}d \tag{6}$$

Combining Equation 5 and 6,

$$\tilde{s} = w_j\tilde{r}. \tag{7}$$

$\tilde{r}$ can be used to check for error, and in case error occurs, the column in which the error resides can be determined by Equation 7.

A. Error Detection Threshold

In ScaLAPACK, since the actual computation is carried out in finite precision arithmetic, $\tilde{r}$ is rarely zero due to the present of rounding error even when no error occurs. This calls for an upper bound to differentiate in $\tilde{r}$ the error caused by rounding error and soft error. In [17], the upper bound for LU with pairwise pivoting is discussed without a definite conclusion. We derive in an alternative way for such bound.

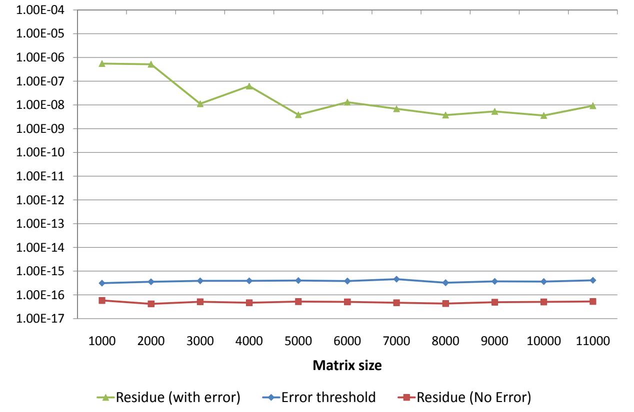

Fig. 3: Error detection

For LU with partial pivoting, we have $PA = LU$, and $A + \delta A = \hat{L}\hat{U}$, where δA is the backward error.

$$\begin{aligned} \hat{U}e - \hat{c} &= \hat{U}e - \hat{L}^{-1}PAe \\ &= (\hat{U} - \hat{L}^{-1}PA)e \\ &= (\hat{U} - \hat{L}^{-1}(\hat{L}\hat{U} - \delta A))e \\ &= \hat{L}^{-1}\delta Ae \\ \therefore \frac{\|\hat{U}e - \hat{c}\|}{\|\hat{c}\|} &\le \frac{1}{\|\hat{c}\|}\|\hat{L}^{-1}\|\,\|\delta A\|\,\|e\| \end{aligned} \tag{8}$$

And because LU with partial pivoting is backward stable, we have $\frac{\|\delta A\|}{\|A\|} \le O(\rho\varepsilon)$, where ε is the machine epsilon and ρ is the growth factor defined as $\rho = \frac{max_{i,j}|u_{i,j}|}{max_{i,j}|a_{i,j}|}$. The inequity in Equation 8 becomes

$$\frac{\|\hat{U}e - \hat{c}\|}{\|\hat{c}\|} \le \frac{1}{\|\hat{c}\|}\|\hat{L}^{-1}\|\,O(1)\rho\varepsilon\,\|A\| = \tau \tag{9}$$

τ is the threshold for error detection. Soft error recovery is triggered when $\frac{\|\hat{U}e-\hat{c}\|}{\|\hat{c}\|} > \tau$.

The constant O(1) depends on the matrix used. Fig. 3 is an evaluation of the error detection threshold using uniformly distributed random matrix.

VI. Recovery of Linear Solver Solution

LU factorization is normally used to solve a system of linear equations $Ax = b$. Therefore when it comes to recovery, there are two options for recovery: recovering only x and/or recovering L, U and P. In this work we focus on the solution of linear system equations based on LU factorization, and recovering L and U will be addressed in future work. The following recovery method is based on [18] and applied explicitly to partial pivoting.

A. Recovery Algorithm

With Equation 6, error column can be determined but no further knowledge is available to precisely pinpoint the error.

Therefore,

$$\begin{aligned}\tilde{P}A - \tilde{P}\tilde{A} &= (\tilde{P}a_{\cdot j} - \tilde{L}\tilde{U}_{\cdot j})e_j^T && (10)\\ \tilde{P}A &= \tilde{L}\tilde{U} + (\tilde{P}a_{\cdot j} - \tilde{L}\tilde{U}_{\cdot j})e_j^T \\ \tilde{P}A &= \tilde{L}\tilde{U} + \tilde{L}(\tilde{L}^{-1}\tilde{P}a_{\cdot j} - \tilde{U}_{\cdot j})e_j^T \\ \tilde{P}A &= \tilde{L}(\tilde{U} + pe_j^T),\ p = \tilde{L}^{-1}\tilde{P}a_{\cdot j} - \tilde{U}_{\cdot j} && (11)\end{aligned}$$

Let $v = \tilde{U}^{-1}p$

$$\begin{aligned}\tilde{P}A &= \tilde{L}\tilde{U}(I + ve_j^T)\\ \therefore\ (\tilde{P}A)^{-1} &= (I + ve_j^T)^{-1}(\tilde{L}\tilde{U})^{-1}\end{aligned}$$

By Sherman-Morrison formula [31], let $\tilde{x}$ be the erroneous solution due to soft error:

$$\begin{aligned}(\tilde{P}A)^{-1} &= \left(I - \frac{1}{1+v_j}ve_j^T\right)(\tilde{L}\tilde{U})^{-1} && (12)\\ \because\ \tilde{P}\tilde{A} &= \tilde{L}\tilde{U},\ \tilde{A}\tilde{x} = b\\ \therefore\ \tilde{L}\tilde{U}\tilde{x} &= \tilde{P}b\\ \therefore\ \tilde{x} &= (\tilde{L}\tilde{U})^{-1}\tilde{P}b\end{aligned}$$

Multiply $\tilde{P}b$ to both sides of Equation 12, we get

$$x = \left(I - \frac{1}{1+v_j}ve_j^T\right)\tilde{x} \qquad (13)$$

v_j is the jth element of v. Therefore, the correct solution x can be obtained from $\tilde{x}$ through an update procedure.

B. Implementation

The core part of Equation 13 is the computing of v. Since $v = \tilde{U}^{-1}p$ and $p = \tilde{L}^{-1}\tilde{P}a_{\cdot j} - \tilde{U}_{\cdot j}$, $\tilde{L}^{-1}\tilde{P}a_{\cdot j}$ is in turn at the center of recovery.

Let $t = \tilde{L}^{-1}\tilde{P}a_{\cdot j}$, and we have $\tilde{L}q = \tilde{P}a_{\cdot j}$. While this can be solved by a combination of PDLAPIV and PDTRSM in ScaLAPACK, two assumptions taken by previous work [17], [18] are revealed:

1) A column of the original matrix is required for recovery
2) The left factor L and pivoting matrix P need to be clean without soft error

The first requirement is trivial to fulfill since for many scientific applications [32], [33], more time is spent in solving the linear system of equations than generating the original matrix. In the case of HPL, the original matrix uses the linear congruential random number generator (RNG) and, consequently, any column can be generated with negligible overhead. In particular, for the general formula for Linear Congruential Generators (LCG) $X_{n+1} = (aX_n + c) \pmod{m}$ with $n \geq 0$,. The values for the constants are choosen to be: $a = 6364136223846793005$, $c = 1$, and $m = 2^{64}$ which creates a sequence with a period of 2^{64} [34]. The *leap frogging* [35] property of LCGs allows us to jump forward in the sequence by utilizing the following formula: $X_{n+k} = (a^k X_n + \frac{a^k-1}{a-1}c) \pmod{m}$, with $k \geq 0, n \geq 0$. In order to obtain X_{n+k} from X_n, we observe that $k = \sum_i b_i 2^i$ where b_i are the binary digits (bits). For each non-zero bit b_i we use a^{2^i} to jump ahead in the sequence using the leap frogging formula. And each a^{2^i} may be generated recursively from $a^{2^{i-1}}$ as the subsequent leaps are performed. Consequently, the number of steps needed to obtain X_{n+k} from X_n is equal to $\log_2 k$: $2\log_2 N$ in the worst case for an N by N matrix generated for HPL.

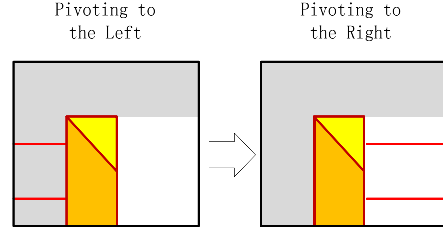

Fig. 5: Two pivoting sweeps in LU factorization

The second requirement shows that the mathematical recovery model is not practical without an extra fault tolerance scheme for the left factor because all part of the matrix is equally possible to be affect by soft error. While the pivoting matrix P (a vector *ipiv* in ScaLAPACK) is easy to protect simply by duplication, the left factor covers a much larger area and therefore is an easy target for soft error. Without confidence in the sanity of the left factor L, the result of the recovery cannot be trusted, even in the case of HPL where the left factor is by default not referenced.

VII. Checkpointing for the Left Factor

It has been shown that the left factor is kept relatively static during LU factorization except row exchanges due to pivoting. This indicates that a simpler checkpointing scheme could be sufficient to give protection to this part. Diskless checkpointing collects checksum by performing checkpointing within a group of processes, and write the result to the memory of a "checkpointing process" at a certain interval. The only requirement is that pivoting does not break the relationship between the data and their diskless checksum, once generated.

A. Diskless Checkpointing

To protect L during factorization, another set of column-wise checksum (*vertical checksum*) is appended at the bottom of the data matrix as shown in fig 4. The checksum is calculated by the same method as the checksum for the right factor. For one soft error, two generator matrices e_2 and w_2 are used in addition to e_1 and e_2 used for the right factor. For original matrix A, two checksums are appended as:

$$\begin{pmatrix} A & Ae_1\ Aw_1 \\ e_2A & \dots \\ w_2A & \dots \end{pmatrix} \qquad (14)$$

At each iteration, once a panel factorization finishes, the newly generated partial L is checkpointed into the same columns of the vertical checksum. The left pivoting in Fig. 5 has a different effect on e_2A and w_2A. Suppose pivoting requests exchanging row j and k. For e_2A:

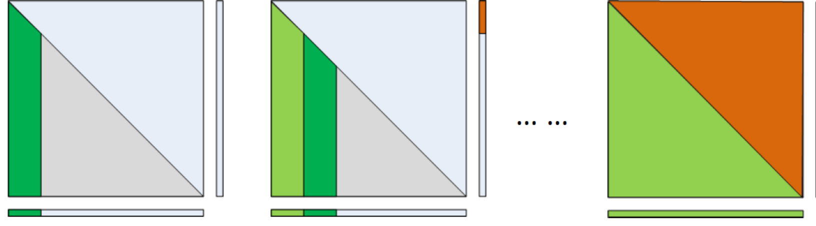

Fig. 4: Checksum protecting partial result of L

$$e_2A = (1,1,\dots,1) \times \begin{pmatrix} a_{1\cdot} \\ \vdots \\ a_{j\cdot} \\ \vdots \\ a_{k\cdot} \\ \vdots \\ a_{m\cdot} \end{pmatrix} = (1,1,\dots,1) \times \begin{pmatrix} a_{1\cdot} \\ \vdots \\ a_{k\cdot} \\ \vdots \\ a_{j\cdot} \\ \vdots \\ a_{m\cdot} \end{pmatrix}$$

This means e_2A is immune to the left pivoting, but for w_2A this conclusion does not hold since data in w_2 are random numbers and let $w_{2,i}$ be the ith element of w_2:

$$w_{2,1}a_1 + w_{2,2}a_2 + \cdot + w_{2,j}a_j + \cdots + W_{2,k}a_k + \cdots + w_{2,m}a_m$$
$$\neq w_{2,1}a_1 + w_{2,2}a_2 + \cdot + w_{2,j}a_k + \cdots + W_{2,k}a_j + \cdots + w_{2,m}a_m$$

and therefore each left pivoting invalidates all previous vertical checksum, and re-checkpointing is not an option because of the high cost.

To cope with this situation, all left pivoting is delayed until the end of all iterations, and for HPL, even though by default the left factored is not referenced, if error is detected in U and therefore requires L for recovery, the pivoting is then performed on demand.

For any column of the computed left factor $[a_1, a_2, \cdots, a_k]^T$, the vertical checkpointing produces the following two sets of checksum:

$$\begin{cases} a_1 + a_2 + \cdots + A_k = c_1 \\ w_{2,1}a_1 + w_{2,2}a_2 + \cdots + w_{2,k}A_k = c_2 \end{cases} \tag{15}$$

Suppose a_j is hit by soft error to $\tilde{a_j}$, the new checksum suite becomes

$$\begin{cases} a_1 + \cdots + \tilde{a_j} + \cdots + A_k = \tilde{c_1} \\ w_{2,1}a_1 + \cdots + w_{2,j}\tilde{a_j} + \cdots + w_{2,k}A_k = \tilde{c_2} \end{cases} \tag{16}$$

Subtract Equations 16 from Equations 15, we get

$$\begin{cases} \tilde{c_1} - c_1 = \tilde{a_j} - a_j \\ \tilde{c_2} - c_2 = w_{2,j}(\tilde{a_j} - a_j) \end{cases}$$

and therefore $w_{2,j} = \frac{\tilde{c_2} - c_2}{\tilde{c_1} - c_1}$. j can be determined by looking up $w_{2,j}$ in w_2, and the erroneous data can be recovered from the first equation of Equations 15.

The check matrix used for L is

$$H = \begin{bmatrix} 1,1,\cdots,1 & -1 \\ w_{2,1}, w_{2,2}, \cdots, w_{2,m} & -1 \end{bmatrix}$$

And it is trivial to show that any two columns of H is independent given the random numbers in the second row do not repeat. By coding theory [36] the minimal distance of this error correcting code is 3, and therefore it can correct up to 1 error per column. In practice, the first row of H could cause large rounding error in the recovery process due to floating pointing arithmetic. Another row of different random numbers can solve the issue as long as no two column of H are linear dependent. Also, using a generator and check matrix with higher minimal distance, more error can be tolerated in one column. This will be addressed in future work.

B. *Local Checkpointing*

From experiment result in VIII-B, we see that vertical checkpointing is limited by scalability because the operation of checkpointing is implemented as a PDGEMM operation on the critical path of LU execution. Only a small amount of processes can participate and the rest are stalled.

Since the left pivoting is delayed, the left factor once computed is not touched any more. The communication incurred by the PDGEMM-based checkpointing can be removed by a local checkpointing scheme. Similar error correcting code is used.

Fig. 6 is an example of the local checkpointing. Block size is $nb \times nb$ and matrix has 5×5 blocks. The process grid is 2×3. Suppose $np_{i,j}, np_{i,j}$ are the size of data owned by process (i, j) (yellow and green for process (0,1) and (1,1)). Each process has a local vertical checksum space in memory of size $2 \times nq_{i,j}$.

Suppose LU factorization proceeds till the second column resulting in the left factor in the area covered in red trapezoid. Right after the panel factorization, all processes that have blocks in the current matrix column started to check if they own any blocks belonging to the current left factor. In this example, process (0,0) has 2 blocks in the red rectangle, and (0,1) has one and half blocks in the red trapezoid. Both of these two processes start to apply their local generator matrix of size $2 \times nq_{i,j}$ for the 2 blocks using DGEMM, and for process (0,1) the first DGEMM is carried out in DTRMM because only the strict lower triangular part is needed. The result is written in the corresponding local checksum location depicted in red lines in Fig. 6.

To recover from error, the same checkpointing scheme as in section VII-A is used locally by each process. Every column of the involved processes is checked for erroneous

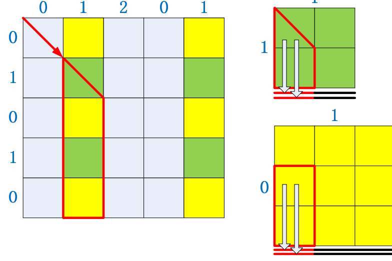

Fig. 6: Local checkpointing algorithm

data and therefore the local checkpointing makes the left-factor protection capable of recovering from one soft error per column of each process.

The advantage of this checkpointing is that it removes unnecessary global communication during checkpointing and breaks the checkpointing operation into $\lceil P \rceil$ embarrassingly parallelism. Further more, on a cluster where more than one core is available on each computing node, this checkpointing can be further hidden by executing it in a separate thread so that the main thread can move on quickly to later steps.

VIII. Experimental Result

This section evaluates the performance for our algorithm in scalability, checkpointing overhead and performance. The scalability and overhead tests are carried out on a small cluster at the University of Tennessee, Knoxville (UTK) named "Dancer", which is an 8-node based on two quad Intel 2.27GHz Xeon cores per node, with an Infiniband 20G interconnect. For the performance experiment, we use another cluster at UTK called 'Newton', which has 72 Dell C6100 computing nodes connected by QDR Infiniband for MPI application. Each node has two 6-core Intel Xeon CPUs. We use OpenMPI on both clusters, and our algorithm implementation is based on ScaLAPACK 1.8.0 from the Netlib using double precision, and on each node GotoBLAS2-1.13 is used. In all the experiments, block size *NB* for ScaLAPACK is set to 100. The column of original matrix that is required for recovery is re-generated by PDMATGEN of ScaLAPACK.

A. Performance Model for the Right Factor

For the right factor, two columns of checksum are appended at the beginning of the factorization, therefore the overhead consists of this one-time checkpointing and extra FLOPS of carrying out LU factorization with the checksum.

According to [37], the execution time of LU driver (PDGESV) in ScaLAPACK is

$$T(N,P) = C_f \frac{N^3}{P} t_f + C_v \frac{N^2}{\sqrt{P}} t_v + C_m \frac{N}{NB} t_m \quad (17)$$

Here N and NB are matrix size and block size (supposed square matrix with square blocks), and P is the total number of processes. $C_f = \frac{2}{3}$, $C_v = 3 + \frac{1}{4}\log_2 P$ and $C_m = NB(6 + log_2 P)$. Because in our implementation, checksum resides in-site with computing processes, all three constants remain unchanged in Equation 17.

The two extra columns of checksum cause

$$T_{extra} = C_f \frac{6N^2 + 12N + 8}{P} t_f + C_v \frac{4N+4}{\sqrt{P}} t_v + C_m \frac{1}{NB} t_m$$

extra run time, which is $O(N^2)$ and is negligible to $T(N,P)$ when problem size and machine size scale up.

The initial checkpointing for the right factor is dominated by a matrix-matrix operation PDGEMM with matrices sized $N \times N$ and $N \times 2$. Using a similar model in Equation17, this overhead is also small compared to PDGESV.

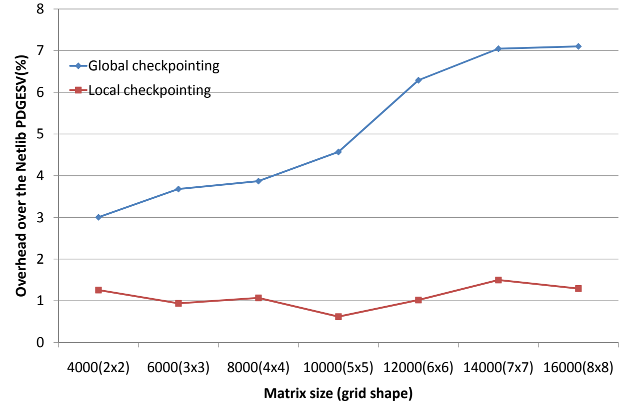

Fig. 7: Weak scalability of global and local checkpointing for the left factor on the Dancer cluster

B. Scalability

Since checkpointing is performed in each iteration for the left factor, the scalability of such algorithm is the main concern. The operation counts of checkpointing a panel of height N_i using PDGEMM is $2 \times NB \times N_i$.

Fig. 7 is the overhead experiment under weak scaling on the Dancer cluster. The overhead is calculated by

$$\frac{T_{ft_pdgesv} - T_{netlib_pdgesv}}{T_{netlib_pdgesv}} \times 100\%$$

And T_{ft_pdgesv} is the run time of the soft resilient version of PDGESV, whereas T_{netlib_pdgesv} is the run time of the Netlib PDGESV, which is what the fault tolerance version is built upon, and serves as a performed baseline.

The result shows that the overhead of vertical checkpointing increases as computing scale and problem size scales up. Since vertical checkpointing is implemented by PDGEMM with $M = 2$, $K = N_i$ and $N = NB$, the checkpointing performance is limited by the performance of PDGEMM. Fig. 8 is PDGEMM performance under such shape comparing to the $M = N = K$ case. The colors of lines are coordinated with the color of vertical axis titles. Clearly PDGEMM does not scale in this matrix shape. In fact, PDGEMM in PBLAS (part of ScaLAPACK) is implemented based on the DIMMA [38] algorithm, which is an enhanced version of SUMMA [39]. SUMMA is designed to work with outer product shape for high parallelism

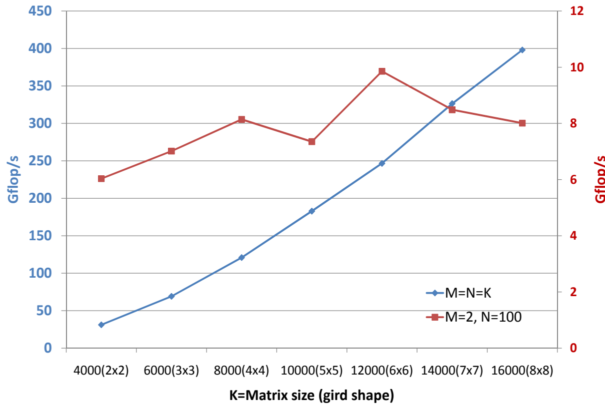

Fig. 8: Weak scalability test of PDGEMM on the Dancer cluster

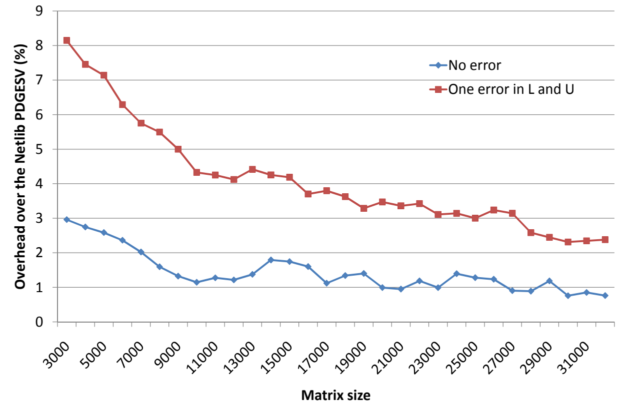

Fig. 9: The checkpointing and recovery overhead on the Dancer cluster

along with sophisticated broadcasting scheme, therefore the inner product shape used by the vertical checkpointing cannot benefit from such a design. In contrast, the local checkpointing scales well because checkpointing is performed in parallel by all involved processes and global collective operation is avoided. This scalability ensures that the overhead caused by the left factor checkpointing will not grow into a performance drag when moving to a larger scale.

With the local checkpointing, the overall overhead of the fault tolerant PDGESV is shown in Fig. 9, where 64 processes are arranged in a 8×8 grid. For the case marked with "one error in L and U", two data items are modified as error injection at location (400,150) and (300,500) right before the panel factorization for blocks (501:end,501:600) starts. The "one error" case includes the checkpointing overhead and the time to recover from the two errors. Same setup applied to performance experiments with alike marks.

This experiment shows that the overhead decreases with larger problems. At 32000, the overhead of the initial checkpointing for the right factor, local checkpointing for the left factor and the extra FLOPs from doing PDGESV with two extra columns is below 1%.

C. Recovery Performance

To test the recovery performance, experiments are carried out on both Dancer and Newton clusters.

Fig. 10 is the performance in Gflop/s of the same experiment in Fig. 9. PDGEMM performance is included as the achievable machine peak to show that ScaLAPACK PDGESV runs at a reasonable speed. Fig. 11 is the result on Newton with 256 processes in a 16×16 grid. Both Gflop/s performance results show that the soft error resilience functionality demands little overhead, and moving to a larger grid does not cause overhead increase.

For LU, algorithm stability is an important issue and it is critical that the recovered solution is numerically close to the original solution. Since in all our experiments the recovered residue $r = \frac{\|Ax-b\|}{\|A\|\|b\|M}$ is in the same magnitude as that of the original solution, this comparison is skipped.

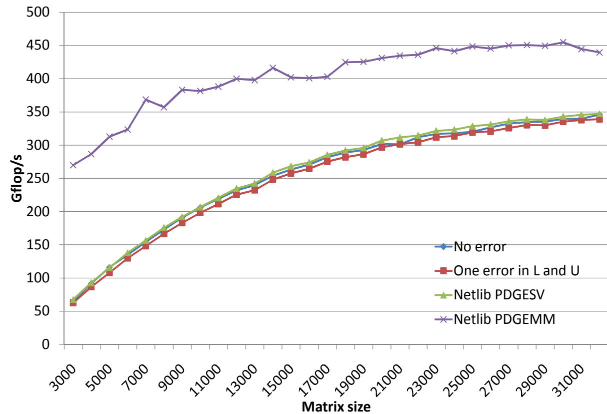

Fig. 10: PDGESV performance with and without soft error resilience on the Dancer cluster

IX. Conclusion

In this paper, a high performance dense linear system solver with soft error resilience is proposed. This work is based on a mathematical model of treating soft error during LU factorization as rank-one perturbation and recovering the solution of $Ax = b$ with the Sherman-Morrison formula. We extended this model to LU with partial pivoting with a practical numerical bound for error detection, and a scalable checkpointing algorithm to protect the left factor from soft error which is needed for recovery. Experimental results based on a ScaLAPACK implementation show that the fault tolerance functionality adds negligible overhead to the linear system solving and scale well on modern cluster systems. As future work, multiple-error problem will be addressed, and the protection to other matrix factorizations will also be explored.

Acknowledgment

The author would like to thank Gerald Ragghianti at the Newton HPC Program of UTK for the generous help with our experiment on the Newton cluster.

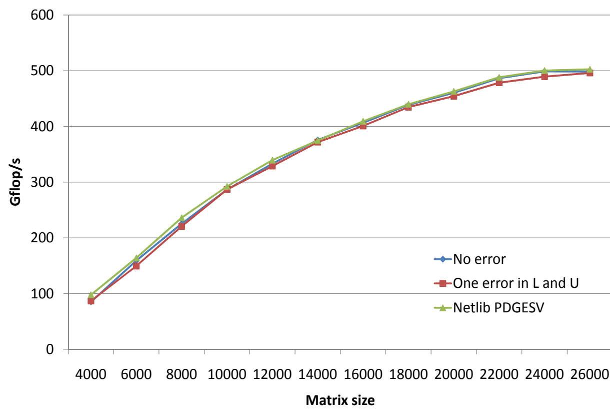

Fig. 11: PDGESV performance with and without soft error resilience on the Newton cluster

REFERENCES

[1] S. Mukherjee, J. Emer, and S. Reinhardt, "The soft error problem: An architectural perspective," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 2005, pp. 243–247.

[2] R. Baumann, "Soft errors in advanced computer systems," *Design & Test of Computers, IEEE*, vol. 22, no. 3, pp. 258–266, 2005.

[3] D. NNSA and D. DARPA, "High-End Computing Resilience: Analysis of Issues Facing the HEC Community and Path-Forward for Research and Development."

[4] I. Haque and V. Pande, "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE, 2010, pp. 691–696.

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

[6] (2010) Cuda toolkit 3.2 math library performance. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2/docs/CUDA_3.2_Math_Libraries_Performance.pdf

[7] J. Dongarra and P. Luszczek, "Reducing the time to tune parallel dense linear algebra routines with partial execution and performance modelling," *niversity of Tennessee Computer Science Technical Report, Tech. Rep*, 2010.

[8] J. Dongarra, *LINPACK: users' guide*. Society for Industrial Mathematics, 1979.

[9] E. Anderson, Z. Bai, and C. Bischof, *LAPACK Users' guide*. Society for Industrial Mathematics, 1999.

[10] K. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 518–528, 2006.

[11] (2009) Fault tolerance for extreme-scale computing workshop report. [Online]. Available: http://www.teragridforum.org/mediawiki/images/8/8c/FT_workshop_report.pdf

[12] J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 9, no. 10, pp. 972–986, 1998.

[13] E. Elnozahy, D. Johnson, and W. Zwaenepoel, "The performance of consistent checkpointing," in *Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on*. IEEE, 1991, pp. 39–47.

[14] J. Plank, Y. Kim, and 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 and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 518–528, 1984.

[16] J. ABRAHAM, "Fault tolerance techniques for highly parallel signal processing architectures," *Highly parallel signal processing architectures*, pp. 49–65, 1986.

[17] F. Luk and H. Park, "An analysis of algorithm-based fault tolerance techniques* 1," *Journal of Parallel and Distributed Computing*, vol. 5, no. 2, pp. 172–184, 1988.

[18] ——, "Fault-tolerant matrix triangularizations on systolic arrays," *Computers, IEEE Transactions on*, vol. 37, no. 11, pp. 1434–1438, 1988.

[19] H. Park, "On multiple error detection in matrix triangularizations using checksum methods," *Journal of Parallel and Distributed Computing*, vol. 14, no. 1, pp. 90–97, 1992.

[20] 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*. IEEE, 1992, pp. 469–480.

[21] C. Anfinson and F. Luk, "A linear algebraic model of algorithm-based fault tolerance," *Computers, IEEE Transactions on*, vol. 37, no. 12, pp. 1599–1604, 1988.

[22] G. Bronevetsky and 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.

[23] 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*. IEEE, pp. 1–12.

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

[25] E. Anderson and J. Dongarra, "Evaluating block algorithm variants in LAPACK," Technical Report LAPACK working note 19, Computer Science Department, University of Tennessee, Knoxville, TN, Tech. Rep., 1990.

[26] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. Whaley, "ScaLAPACK: a portable linear algebra library for distributed memory computers–design issues and performance," *Computer Physics Communications*, vol. 97, no. 1-2, pp. 1–15, 1996.

[27] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings, 1994, vol. 400.

[28] 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*, 1997.

[29] J. Wilkinson, *The algebraic eigenvalue problem*. Oxford University Press, USA, 1965.

[30] T. Davies, C. Karlsson, H. Liu, C. Ding, , and Z. Chen, "High Performance Linpack Benchmark: A Fault Tolerant Implementation without Checkpointing," in *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011)*. ACM.

[31] J. Ortega, *Numerical analysis: A second course*. Society for Industrial Mathematics, 1990.

[32] K. Klimkowski and H. Ling, "Performance evaluation of moment-method codes on an intel iPSC/860 hypercube computer," *Microwave and Optical Technology Letters*, vol. 6, no. 12, pp. 692–694, 1993.

[33] E. Lezar and D. Davidson, "GPU-Accelerated Method of Moments by Example: Monostatic Scattering," *Antennas and Propagation Magazine, IEEE*, vol. 52, no. 6, pp. 120–135, 2010.

[34] D. Knuth, "The art of computer programming, volume 2: seminumerical algorithms," 1997.

[35] J. E. Gentle, *Random Number Generation and Monte Carlo Methods*, 2nd ed. Springer-Verlag New York, Inc., 2003, iSBN 0-387-00178-6.

[36] A. Neubauer, J. Freudenberger, and V. Kühn, *Coding theory: algorithms, architectures, and applications*. Wiley-Interscience, 2007.

[37] L. Blackford, A. Cleary, J. Choi, E. d'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, *et al.*, *ScaLAPACK users' guide*. Society for Industrial Mathematics, 1997.

[38] J. Choi, "A new parallel matrix multiplication algorithm on distributed-memory concurrent computers," in *hpc-asia*. Published by the IEEE Computer Society, 1997, p. 224.

[39] R. Van De Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.