Replacing Pivoting in Distributed Gaussian Elimination with Randomized Techniques

Neil Lindquist*, Piotr Luszczek*, and Jack Dongarra*†
*Innovative Computing Laboratory, University of Tennessee, {nlindqu1,luszczek,dongarra}@icl.utk.edu
†University of Manchester; Oak Ridge National Laboratory

***Abstract*—Gaussian elimination is a key technique for solving dense, non-symmetric systems of linear equations. Pivoting is used to ensure numerical stability but can introduce significant overheads. We propose replacing pivoting with recursive butterfly transforms (RBTs) and iterative refinement. RBTs use an FFT-like structure and randomized elements to provide an efficient, two-sided preconditioner for factoring. This approach was implemented and tested using Software for Linear Algebra Targeting Exascale (SLATE). In numerical experiments, our implementation was more robust than Gaussian elimination with no pivoting (GENP) but failed to solve all the problems solvable with Gaussian elimination with partial pivoting (GEPP). Furthermore, the proposed solver was able to outperform GEPP when distributed on GPU-accelerated nodes.**

***Index Terms*—Linear systems, randomized algorithms**

I. Introduction

Gaussian elimination, or LU factorization, is the primary method for solving dense, non-symmetric systems of linear equations. The most common variant of this factorization, Gaussian elimination with partial pivoting (GEPP), swaps the row with the largest element on or below the diagonal with the diagonal's row before factoring that column. However, partial pivoting adds noticeable overheads to the performance of the factorization. First, to determine the pivots, the largest element below the diagonal element must be found for each column, after the previous column's update has been applied and before the current column's update can be applied. This increases the communication for distributed solvers and limits the amount of overlap with computation. Second, swapping the rows introduces data movement costs, which again, can become expensive in distributed solvers. Unfortunately, pivoting is critical for numerical stability; Gaussian elimination with no pivoting (GENP) is only guaranteed to be safe for specific classes of matrices, such as diagonally dominant matrices and totally nonnegative matrices [1]. Thus, methods with smaller overhead than partial pivoting but better numerical stable than no pivoting are desirable.

There have been alternative pivoting strategies proposed to reduce the overhead of pivoting. One such approach is incremental pivoting, which is based on out-of-core techniques and only applies GEPP to two tiles at a time [2]. This reduces the amount of communication to choose pivots at the cost of increased computation and reduced parallelism in the panel factorization. Another approach is Gaussian elimination with tournament pivoting, or communication avoiding LU, which selects the "best" n_b rows to be the pivots of the n_b-column panel before starting the Gaussian elimination process for those columns [3]. A copy of each tile in the panel is factored with GEPP, and the pivot rows are recorded. Then, the sets of rows are reduced in a binary manner by taking the corresponding rows from the original panel as a $2n_b \times n_b$ matrix, applying GEPP, then taking the n_b pivot rows. When the Gaussian elimination is finally applied, only the chosen n_b rows are considered as pivots, allowing the leading $n_b \times n_b$ matrix to be factored without further communication. Additionally, swapping rows in the trailing matrix update can be partially overlapped with this final factorization. Unfortunately, these approaches still require swapping rows, likely between different processes.

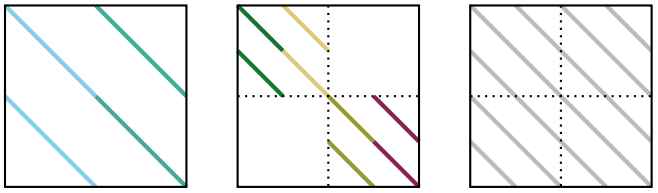
An alternative approach is to completely remove pivoting and instead precondition the matrix to provide numerical stability. There have been a variety of possible preconditioners proposed [4]. However, we are focusing on recursive butterfly transforms (RBTs) [5], which have had promising success in past research [6], [7], [8]. However, previous studies tested problems that were either sparse, single-node, or symmetric, so we explored the dense, distributed, non-symmetric case.

II. Related Work

Parker [5] introduced RBT and proved that, with probability 1, they render a matrix factorizable with GENP. He also compared against LINPACK's dgefa() factorization subroutine on matrices whose elements were drawn from a variety of random distributions. Additional properties of an RBT-variant were further explored in relation to spectral transforms [9].

A distributed-memory variant was proposed as an alternative to LDLT factorization in DPLASMA [7]. It was based on the DaGUE distributed Direct Acyclic Graph (DAG) scheduler that scalably implements a dataflow paradigm. RBT allowed the authors to avoid pivoting that, in the case of LDLT, involves both rows and columns to preserve symmetry and is highly detrimental on large scale machines that may suffer when these latency-sensitive operations interrupt computationally-intensive tasks working on matrix blocks or tiles.

Another application of RBT for Gaussian-Elimination factorization was proposed for multicore systems accelerated with GPU hardware for both symmetric [10] and non-symmetric linear systems [6]. Relatedly, the use of RBT for Intel Xeon Phi accelerators has been tested for the non-symmetric case [11]; this hardware behaved like the GPUs, except for



(a) Sparsity pattern of $B^{\langle n \rangle}$.

(b) Sparsity pattern of $\text{diag}\left(B_1^{\langle n/2 \rangle}, B_2^{\langle n/2 \rangle}\right)$.

(c) Sparsity pattern of a depth two RBT.

Fig. 1: Sparsity patterns of the last two terms of (1) and their product, a depth two butterfly.

the iterative refinement, which had not been optimized. These systems are designed to exploit large amounts of parallelism, which can be limited by the dependencies and data movements in GEPP. Thus, removing pivoting increases their performance, while RBTs can be cheaply applied. Additionally, the works testing GPU-based systems provide numerical accuracy results for the reduced butterfly depth used everywhere, except Parker's original paper.

Applying RBT to sparse matrices was also proposed to reduce the amount of fill-in compared to pivoting [8]. One particularly interesting suggestion from that work is to only apply an RBT to one side of the matrix instead of both sides.

Our approach unifies these approaches into a single formulation that targets multicore, GPU, distributed memory, and standards-based runtime DAG scheduling for algorithms expressed with a dataflow approach. Our work applies to general non-symmetric, dense matrices that admit LU factorization and thus are numerically nonsingular in the working precision — either 32-bit or 64-bit floating-point formats.

III. Definition and Properties of Recursive Butterfly Transforms

A butterfly matrix is a matrix of the form

$$B^{\langle n \rangle} = \frac{1}{\sqrt{2}} \begin{bmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{bmatrix},$$

where R_0 and R_1 are $n/2 \times n/2$, nonsingular, diagonal matrices [5]. Then, an RBT of depth d, $U^{\langle n \rangle}$, is a matrix of the form

$$U^{\langle n \rangle} = \begin{bmatrix} B_1^{\langle n/2^{d-1} \rangle} & \dots & 0 \\ \vdots & \ddots & 0 \\ 0 & \dots & B_{2^{d-1}}^{\langle n/2^{d-1} \rangle} \end{bmatrix} \times \dots \times \begin{bmatrix} B_1^{\langle n/2 \rangle} & 0 \\ 0 & B_2^{\langle n/2 \rangle} \end{bmatrix} \times B^{\langle n \rangle}. \quad (1)$$

Note that n must be a multiple of 2^{d-1}; however, a linear system can be augmented with ones on the diagonal and zeros on the off diagonals to add the necessary rows and columns to the matrix. Figure 1 shows the sparsity pattern for butterfly matrices and the resulting depth-two RBT. Additionally, (1) is a generalized form of the original definition of RBTs, which

1: $A' \leftarrow \langle \mathcal{U} \rangle^T \times A \times \langle \mathcal{V} \rangle$
2: $\vec{b}' \leftarrow \langle \mathcal{U} \rangle^T \times \vec{b}$
3: $L, U \leftarrow$ Apply GENP to A'
4: $\vec{x}' \leftarrow U^{-1} \times L^{-1} \times \vec{b}'$
5: $\vec{x} \leftarrow \langle \mathcal{V} \rangle \times \vec{x}'$

Fig. 2: Solving the linear system $A\vec{x} = \vec{b}$ using recursive butterfly transforms $\langle \mathcal{U} \rangle$ and $\langle \mathcal{V} \rangle$.

is the case where $d = log_2(n) + 1$ [5].[1] For example, the structure of $U^{\langle 4 \rangle}$ with a depth of 2 is

$$U^{\langle 4 \rangle} = \frac{1}{2} \begin{bmatrix} r_1^{\langle 2 \rangle} & r_2^{\langle 2 \rangle} & & \\ r_1^{\langle 2 \rangle} & -r_2^{\langle 2 \rangle} & & \\ & & r_3^{\langle 2 \rangle} & r_4^{\langle 2 \rangle} \\ & & r_3^{\langle 2 \rangle} & -r_4^{\langle 2 \rangle} \end{bmatrix} \times \begin{bmatrix} r_1^{\langle 4 \rangle} & & r_3^{\langle 4 \rangle} & \\ & -r_2^{\langle 4 \rangle} & & r_4^{\langle 4 \rangle} \\ r_1^{\langle 4 \rangle} & & -r_3^{\langle 4 \rangle} & \\ & r_2^{\langle 4 \rangle} & & -r_4^{\langle 4 \rangle} \end{bmatrix},$$

where each $r_i^{\langle j \rangle}$ is a scalar. The structure of RBT matrices is equivalent to the bit-shuffle permutations performed by Discrete Fourier Transform (DFT) matrices [5] that bear the butterfly name due to the data transfer patterns they form [12]. This relation to DFT computational and communication processes makes RBT matrices efficient to apply in practice. Additionally, if each nonzero in the component butterfly matrices has magnitude one, the transform is unitary.

RBTs can be used to avoid pivoting by preconditioning the system. Let $\langle \mathcal{U} \rangle$ and $\langle \mathcal{V} \rangle$ be recursive butterfly transforms. Then, the linear system $A\vec{x} = \vec{b}$ can be solved by the algorithm in Fig. 2.[2] If a $2^d \times 2^d$, nonsingular matrix is preconditioned by a random, depth-d RBT, then GENP will succeed with probability 1 [5]. However, previous work has noted that most matrices can be preconditioned successfully with a depth-2 RBT if iterative refinement is also used [6].

It may appear that the RBTs could instead use butterflies of size $2^k \times 2^k$ for $k = 1, 2, \dots, d$, which would reduce communication in distributed contexts. However, the conditions of the leading, principal submatrices with dimension divisible by 2^d do not change. Because GENP is numerically safe if and only if each leading, principal submatrix is nonsingular and well-conditioned [4], this block diagonal form of RBT provides limited benefit unless the depth is close to $\log_2(n)$. On the other hand, when using the form of RBT described in (1), more elements contribute to the leading submatrix condition numbers. Each element of the leading principal submatrices of dimension less than $n/2^d$ are approximately equal to the sum of 2^{2d} elements, with the random factors providing a low

[1] $B^{\langle 1 \rangle}$ is defined to be a nonzero scalar.

[2] While the algorithm uses matrix transposition, it is also valid for RBT matrices in the complex domain, either as written or after replacing the transpositions with conjugate-transpositions.

probability of the sums being too close to 0. Large submatrices are affected by elements from the entire matrix, limiting how poorly conditioned the submatrix can become.

A. Packed Storage for Recursive Butterfly Transforms

Explicitly constructing the RBT matrices would triple the storage required for the linear system. However, the structured sparsity can be used to reduce storage costs. Note that a butterfly matrix is equivalent to

$$\begin{bmatrix} I & I \\ I & -I \end{bmatrix} \times R,$$

where R is diagonal. So, only the m diagonal elements of an $m \times m$ butterfly matrix need to be stored. Hence, a recursive butterfly transform of depth d and size n can be stored in an $n \times d$ dense matrix where each column stores one term of (1). Because $d \ll n$, recursive butterfly transforms introduce little additional storage when using packed storage.

B. Computation Cost of Recursive Butterfly Transforms

Similar to storing the matrix, utilizing the transform's structure provides significant benefits for RBT application. Let A be an $m \times m$ matrix, and let

$$B_1^{\langle m \rangle} = \begin{bmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{bmatrix} \text{ and } B_2^{\langle m \rangle} = \begin{bmatrix} R_2 & R_3 \\ R_2 & -R_3 \end{bmatrix}$$

be butterfly matrices stored in vectors $\vec{w}_1$ and $\vec{w}_2$ using the packed format from Sec. III-A. Then,

$$\begin{aligned} (B_1^{\langle m \rangle})^T A B_2^{\langle m \rangle} &= \begin{bmatrix} R_0 & R_0 \\ R_1 & -R_1 \end{bmatrix} A \begin{bmatrix} R_2 & R_3 \\ R_2 & -R_3 \end{bmatrix} \\ &= \begin{bmatrix} R_0 & R_0 \\ R_1 & -R_1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} R_2 & R_3 \\ R_2 & -R_3 \end{bmatrix} \\ &= \begin{bmatrix} R_0 & 0 \\ 0 & R_1 \end{bmatrix} C \begin{bmatrix} R_2 & 0 \\ 0 & R_3 \end{bmatrix} \\ &= diag(\vec{w}_1)\, C\, diag(\vec{w}_2) \end{aligned} \tag{2}$$

where

$$C = \begin{bmatrix} A_{11} + A_{12} + A_{21} + A_{22} & A_{11} - A_{12} + A_{21} - A_{22} \\ A_{11} + A_{12} - A_{21} - A_{22} & A_{11} - A_{12} - A_{21} + A_{22} \end{bmatrix}.$$

Thus, transforming an $m \times m$ matrix requires $4m^2$ FLOP. Note that applying RBTs to both sides of an $n \times n$ matrix can be broken down into $(n/m)^2$ butterfly matrix applications of size $m \times m$, and thus, requires $4n^2$ FLOP. Hence, to apply an RBT of depth d requires $4dn^2$ FLOP.

Similarly, an RBT in the packed format can be applying to a vector. Then, each transform only requires $\mathcal{O}(n)$ FLOP for a total of $\mathcal{O}(dn)$ FLOP to apply a recursive butterfly transform of depth d. Because $d \ll n$, the cost of transforming the vectors will be disregarded for the rest of the paper.

In distributed settings, it is also important to consider the amount of inter-process communication needed. Similar to computing the FLOP count, we start by considering the cost of applying a single butterfly matrix to each side of an $m \times m$ matrix, A. First, note that in the description above of applying a butterfly matrix to each side, each element, of $(B_1^{\langle m \rangle})^T A B_2^{\langle m \rangle} = (\alpha_{ij})$ depends on four elements in A, one element from either R_0 or R_1, and one element from either R_2 or R_3. Specifically, the dependencies of $\alpha_{1,1}$, $\alpha_{1+m/2,1}$, $\alpha_{1,1+m/2}$, and $\alpha_{1+m/2,1+m/2}$ on $A = (a_{ij})$ are the elements in the same positions, i.e. $a_{1,1}$, $a_{1+m/2,1}$, $a_{1,1+m/2}$, and $a_{1+m/2,1+m/2}$. Furthermore, the elements in R_0 and R_1 are shared across the rows of the result, and the elements in R_2 and R_3 are shared across the columns. We only consider the case where $(B_1^{\langle m \rangle})^T A B_2^{\langle m \rangle}$ and A are distributed across the processes in the same manner, which occurs when reusing the storage of A to hold the resulting transformed matrix. Additionally, let p be the maximum number of processes the elements of a single row of A can be distributed across and let q be the corresponding value for columns. Then by gathering each set of four interacting elements onto a single process where one of those elements already resides and returning them after doing the appropriate computation, we need to transfer up to $6m^2$ elements of the matrix, $mp/2$ elements of each R_0 and R_1, and $mq/2$ elements of each R_2 and R_3.

Next, we expand the communication cost analysis from a single pair of butterflies to the entire transform. Simply combining the individual butterfly applications as described above would result in transferring $6dn^2$ elements of the matrix, $np/2$ elements of each R_0 and R_1, and $nq/2$ elements of each R_2 and R_3. Note that it is possible to merge the return of elements for one application with the gathering of elements for the next application. With this improvement, it would be possible to reduce the upper bound on matrix element transfers to $4dn^2 + 2n^2$. However, because of the increase in algorithm complexity, we have not implemented this improvement.

IV. Implementing recursive butterfly transforms

We implemented a linear solved based on RBT and GENP using Software for Linear Algebra Targeting Exascale (SLATE). SLATE is a dense linear algebra library for distributed, high-performance systems, both with and without GPU acceleration [13]. SLATE uses a tile layout for matrix storage, which provides significant flexibility and a natural way to organize parallelism. While SLATE is designed for hybrid algorithms between CPU and GPU, we have not implemented a GPU version of RBT application.

Given the recursive structure of RBT, we implemented the application by applying one term of the product in (1) at a time, albeit in a two-sided manner, to transform the matrix. Furthermore, the transformation is simplified by applying one pair of butterflies at a time, as described in Section III-B. This is implemented as an elementwise operation on the four submatrices corresponding to each pair of left and right butterflies. Because the matrices are distributed, each set of four elements is sent to the node owning the upper left element to be transformed. Figure 3 shows pseudocode for our two-sided RBT. The first procedure, RBT, breaks the transformation into individual butterfly applications. The second procedure, $\text{RBT2}_{\text{TILE}}$, applies a single pair of butterflies where the matrices and butterfly values are separated similar to the organization in (2). In a non-tiled code, $\text{RBT2}_{\text{TILE}}$ is equivalent

```
1:  procedure RBT(A, U, V)
2:      d ← depth(U)
3:      for k from d − 1 to 0 do
4:          b_n ← 2^k                          ▷ Number of butterflies
5:          h ← n/2^{k+1}                      ▷ Half a butterfly's size
6:          for b_j from 0 to b_n do           ▷ Right butterflies
7:              j_1 ← 2b_j h
8:              j_2 ← j_1 + h                  ▷ Column indices for b_j
9:              j_3 ← j_2 + h
10:             for b_i from 0 to b_n do       ▷ Left butterflies
11:                 i_1 ← 2b_i h
12:                 i_2 ← i_1 + h              ▷ Row indices for b_i
13:                 i_3 ← i_2 + h
14:                 A_11 ← A[i_1:i_2, j_1:j_2]
15:                 A_12 ← A[i_1:i_2, j_1:j_2]
16:                 A_21 ← A[i_2:i_3, j_2:j_3]
17:                 A_22 ← A[i_2:i_3, j_2:j_3]
18:                 U_1 ← U[i_1:i_2, k]
19:                 U_2 ← U[i_2:i_3, k]
20:                 V_1 ← V[j_1:j_2, k]
21:                 V_2 ← V[j_2:j_3, k]
22:                 RBT2(A_11, A_12, A_21, A_22,
                         U_1, U_2, V_1, V_2)

23: procedure RBT2_TILE(A_11, A_12, A_21, A_22, U_1, U_2, V_1, V_2)
24:     m_b × n_b ← dim(A_11)    ▷ Tiles are all the same size
25:     for j from 0 to n_b do
26:         v_1 ← V_1[j]
27:         v_2 ← V_2[j]
28:         for i from 0 to m_b do
29:             u_1 ← U_1[j]
30:             u_2 ← U_2[j]
31:             a_11 ← A_11[i, j]
32:             a_12 ← A_12[i, j]
33:             a_21 ← A_21[i, j]
34:             a_22 ← A_22[i, j]
35:             A_11[i, j] ← u_1 v_1 (a_11 + a_12 + a_21 + a_22)
36:             A_12[i, j] ← u_1 v_2 (a_11 − a_12 + a_21 − a_22)
37:             A_21[i, j] ← u_2 v_1 (a_11 + a_12 − a_21 − a_22)
38:             A_22[i, j] ← u_2 v_2 (a_11 − a_12 − a_21 + a_22)
```

Fig. 3: Algorithms from the two-sided RBT application $U^T AV$. The procedure RBT breaks the problem into individual butterfly applications. The procedure $\text{RBT2}_{\text{TILE}}$ applies a single, two-sided butterfly transform to a set of four tiles.

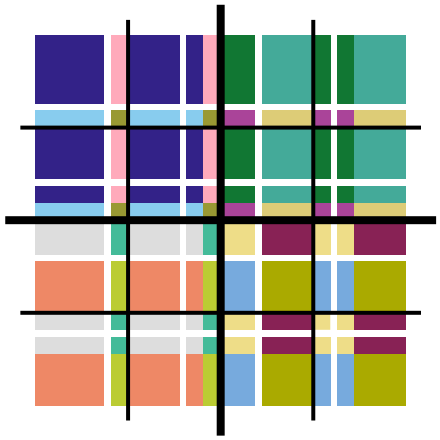


Fig. 4: Subtiles gathered for computing a layer of an RBT for butterflies of size $n/2$. All submatrices of the same color are gathered onto the process owning the upper left entry of the color for the computation.

to the omitted RBT2. However, as our implementation is tile-based and distributed, we have an additional step to group appropriate sets of elements into tiles on a single process before calling $\text{RBT2}_{\text{TILE}}$.

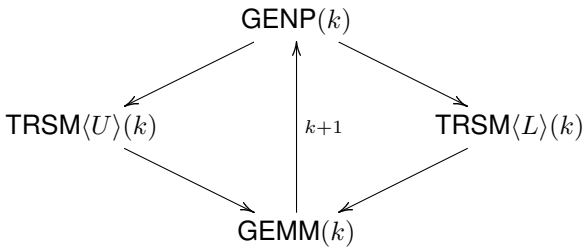
Unfortunately, the way the elements interact when applying an RBT rarely correspond to the tiles used to store the matrix data. So, we implemented support for transmitting partial tiles in a manner similar to previous work on distributed, RBT-based solvers [7]; however, our approach differs in that we explicitly gather the elements into tiles defined by the size and owning-process of the upper left submatrix. The gathered tiles can then be treated as the application of a single butterfly described in Sec. III-B. Figure 4 shows an example of how elements are gathered for the two-sided transform of A,

$$\begin{bmatrix} B_1^{\langle n/2 \rangle} & 0 \\ 0 & B_2^{\langle n/2 \rangle} \end{bmatrix}^T A \begin{bmatrix} B_3^{\langle n/2 \rangle} & 0 \\ 0 & B_4^{\langle n/2 \rangle} \end{bmatrix},$$

when A is tiled into a 5×5 grid of uniform size. Currently, our implementation does involve duplicate transfers of the butterfly nonzeros to reduce the complexity of computing the indices and managing storage lifetimes.

Note that if the size of the matrix can be controlled, it may be beneficial to adjust the matrix size so that the transformation can be efficiently applied. For example, a 2d block-cyclic distribution of an $n \times n$ matrix on a $p \times q$ process grid with tiles of size $b \times b$ does not need to subdivide tiles to apply RBTs when $2pb$ and $2qb$ divide n. Furthermore, for a butterfly depth of d, inter-process communication is unneeded when $2^{d+1}pb$ and $2^{d+1}qb$ divide n.

Similar to previous work [6], we used a depth 2 RBT for our solver, chose RBT elements of the form $\exp(r/10)$ with r chosen from the uniform distribution $[-\frac{1}{2}, \frac{1}{2}]$, and provide iterative refinement. While a depth of 2 does not provide the probabilistic guarantee of a depth of $\log_2(n)$, it does not require the matrix size to be a power of 2 and has lower



GENP($k \in \{1 \dots n\}$)
TRSM$\langle U\rangle(k \in \{1 \dots n-1\})$
TRSM$\langle L\rangle(k \in \{1 \dots n-1\})$
GEMM($k \in \{1 \dots n-1\}$)

Fig. 5: Enumerated DAG of GENP with the graph nodes representing tasks that are parameterized by input parameter ranges.

computational overheads. The copy of the matrix for iterative refinement is always kept entirely on the CPUs to increase the maximum problem size that can be stored in GPU memory. Because a copy of the matrix is already needed to achieve the same accuracy as GEPP, there is limited overhead to check the backward error of the solution provided by the RBT-solver. So, problems that cannot be solved accurately with our solver can be re-solved with a regular GEPP implementation.

V. Implementation of GENP

Our implementation of GENP is a right-looking variant. Its organization resembles SLATE's Cholesky factorization that is limited to Hermitian, positive-definite matrices – GENP operates on both upper and lower parts of the matrix due to the lack of symmetry and does away with the calls to the SYRK BLAS subroutine. In keeping with SLATE's design, we use large OpenMP tasks to specify data dependencies. Fig. 5 shows the DAG that is instantiated by OpenMP's task scheduling runtime. Each of these large tasks is then split into independent operations on individual tiles, which can run in parallel. First, the current diagonal tile, $A_{k,k}$, is factored into LU components with the GENP task called GENP(k) in the figure. Then, the rest of the column, $A_{k+1:n,k}$, is replaced with $A_{k+1:n,k}U^{-1}$, and the rest of the row, $A_{k,k+1:n}$, is replaced with $L^{-1}A_{k,k+1:n}$ that are called TRSM$\langle U\rangle(k)$ and TRSM$\langle L\rangle(k)$ in the figure, respectively. Finally, as the last phase of applying the Schur complement $A_{k,k+1:n} \times A_{k,k}^{-1} \times A_{k,k+1:n}$, the trailing matrix $A_{k+1:n,k+1:n}$ is updated with newly computed $\hat{A}_{k+1:n,k} \times \hat{A}_{k,k+1:n}$ denoted with GEMM(k) in the figure. These steps are repeated until the entire matrix is factored. Fig. 6 shows the parameterized DAG of our GENP that specifies the tile dependencies for any matrix of $n \times n$ tiles.

- GENP(k) :
  - $\leftarrow$ GEMM($k-1$)
  - $\rightarrow$ TRSM$\langle U\rangle(k)$
  - $\rightarrow$ TRSM$\langle L\rangle(k)$
- GENP(1) :
  - $\rightarrow$ TRSM$\langle U\rangle(k)$
  - $\rightarrow$ TRSM$\langle L\rangle(k)$
- GENP(n) :
  - $\leftarrow$ GEMM($n-1$)
- TRSM$\langle U\rangle(k)$:
  - $\leftarrow$ GENP(k)
  - $\rightarrow$ GEMM(k)
- TRSM$\langle L\rangle(k)$:
  - $\leftarrow$ GENP(k)
  - $\rightarrow$ GEMM(k)
- GEMM(k) :
  - $\leftarrow$ TRSM$\langle U\rangle(k)$
  - $\leftarrow$ TRSM$\langle L\rangle(k)$
  - $\rightarrow$ GENP($k+1$)

Fig. 6: Parameterized DAG of GENP that fully specifies each task and its dependencies in a parameterized form. Note that GENP(1) and GENP(n) are the special cases of factoring the first and last tiles, respectively.

VI. Experimental Results

Using the implementation of the RBT-based solver described in Sec. IV, we tested its accuracy and performance relative to SLATE's GEPP implementation for double precision. Our results were measured on the Summit supercomputer at the Oak Ridge National Laboratory. We used eight nodes, each containing two sockets, with one 22-core IBM POWER9 processors. Each socket was connected to three NVIDIA Volta V100 GPUs each with 16 GiB High Bandwidth Memory (HBM) 2, and 256 GiB DDR4. The network consists of dual-rail EDR InfiniBand connected in a Non-blocking Fat Tree topology. The operating system is Red Hat Enterprise Linux version 7.6 with an IBM Spectrum Scale™ filesystem.

Our software stack included GCC 8.1.1, CUDA 10.1.243, Spectrum MPI 10.3.1.2, ESSL 6.1.0-2, and Netlib LAPACK 3.8.0. The code was compiled with `-fopenmp -O3`, and `nvcc` was configured for the Volta architecture. One process was allocated and bound to each socket for a total of 16 processes. Through the job launcher, each process was assigned all 21 available cores and 3 available GPUs, and tasks were bound and distributed with `packed:21` and `packed`, respectively. Summit was set to `smt2` mode. Our results were measured using a modified version of SLATE's tester that supports our new solver plus additional matrix generators. The solvers' parameters were tuned for performance, except for tile size, which was only tuned for GEPP. Tests were run with the flags `--origin h --target d --ref n --nb 832 --seed 96 --seedB 42 --ib 64 --lookahead 0 --dim $DIMS`. GEPP also used the flags `--panel-threads 20 --p 2 --q 8`, and the RBT solver used `--depth` and `--refine` as appropri-

TABLE I: Normwise backward error of various solvers for matrices of size 100 000. NA indicates the solution was invalid due to division by zero or overflow.

Matrix	GEPP	RBT Solver refined	RBT Solver	GENP refined	GENP
Random $[0, 1]$	1.23×10^{-15}	2.97×10^{-17}	6.43×10^{-12}	2.67×10^{-17}	4.10×10^{-12}
Random $[-1, 1]$	2.39×10^{-15}	2.93×10^{-17}	1.53×10^{-11}	1.19×10^{-15}	8.76×10^{-11}
Random Normal	1.77×10^{-15}	3.29×10^{-17}	6.68×10^{-12}	3.23×10^{-17}	1.71×10^{-11}
Random $\{0, 1\}$	1.82×10^{-15}	2.25×10^{-17}	6.15×10^{-12}	NA	NA
`fielder`	3.03×10^{-18}	2.92×10^{-19}	1.73×10^{-17}	NA	NA
`orthog`	2.29×10^{-16}	9.19×10^{-3}	1.00×10^{-2}	1.21×10^{-1}	1.30×10^{-1}
`gfpp`	NA	2.79×10^{-19}	5.06×10^{-18}	NA	NA

ate. GEPP's process grid of 2×8 was allocated such that the processes sharing a column were located on the same node. The accuracy tests also used the flags `--matrix` and `--matrixB` to control the problem generator. The performance tests were preceded by solving a problem of size 6656 or 7072 (for tile-aligned and not tile-aligned results, respectively) to ensure that the initialization of BLAS and MPI was not included.

A. Accuracy Results

First, we tested the accuracy of the RBT solver in comparison to GEPP and GENP. We tested matrices of size 100 000 and compared the normwise backward error, $\|r\|_1/(\|A\|_1\|x\|_1)$, in Table I. Both the RBT solver and GENP were run with and without one step of iterative refinement for the sake of comparison. The tested matrices are a subset of the matrices previously used to test RBT-based solvers for problems of size 1024 [6], with minor modification to the last matrix. The elements of the first two matrices are uniformly selected from the given ranges. The third matrix has elements selected from a normal distribution with mean 0 and standard deviation 1. The fourth matrix has elements selected from 0 and 1 with equal probability. The fifth and sixth matrices come from MATLAB's gallery function. Note that $|i - j|$ used in previous works is identical to `fielder`. The last matrix is based on the matrix in Higham's Matrix Computation Toolbox [14], except the elements below the diagonal are halved to ensure GEPP pivots the matrix as intended. All problems had a right hand side selected from a normal distribution with mean 0 and standard deviation 1.

As Table I shows, the RBT-based solver was able to solve all but one problem, including all of the problems solvable by GENP and the example of worst-case behavior of GEPP. The one matrix that the RBT-based solver had poor accuracy was the `orthog` matrix. This matrix is constructed by setting the $(i, j) \in [1, n]^2$ element to $\sqrt{2/(n+1)}\sin(ij\pi/(n+1))$, which makes it orthogonal and symmetric. Previous work showed success on this matrix for a size of 1024 [6] but not for a size of 30 000 [15] when using an RBT depth of 2 and 1 step of iterative refinement. To understand the behavior of this matrix as the problem size grows, we plotted the backward error, $\|r\|_1/(\|A\|_1\|x\|_1)$, for varying problem sizes and RBT depths, but otherwise as the first test, in Fig. 7 For the smaller

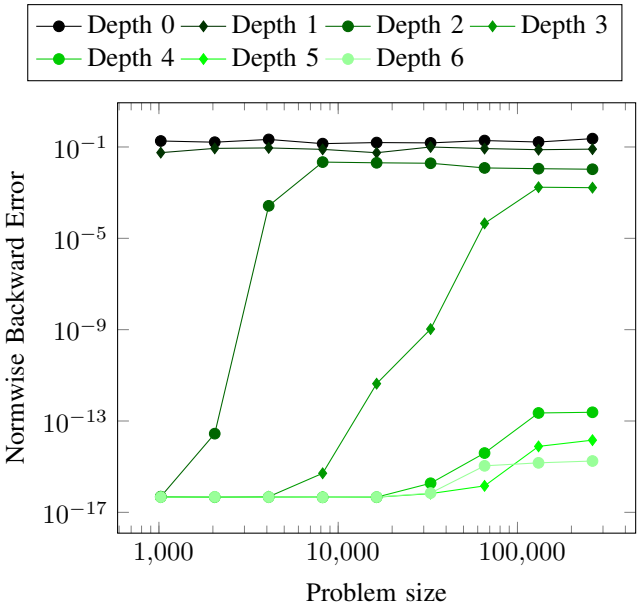


Fig. 7: Accuracy of the RBT-based solver for various sizes of the `orthog` and various RBT depths.

depths tested, there appears to be a problem size at which the depth loses effectiveness. However, depths of 4, 5, and 6 all started losing accuracy at a similar point, which complicates the situation We believe the sine-based structure of this matrix causes the difficulties but have not been able to quantify the source of the issue.

B. Performance Results

Next, we tested the performance of the RBT solver in comparison to GEPP and GENP. Because of the increase in the complexity of inter-process communication, the RBT solver was tested on two sets of sizes. First are problem sizes that allow the RBT communication to be done as whole tiles, specifically, problem sizes that are multiples of 16 640 that did not run out of memory for the GEPP solver. This first set of problem sizes was also used for the GEPP and GENP performance. Second are problem sizes that are smaller by 416 elements (half a tile width) than the first set of problem sizes. Figures 8 presents the performance in both TFLOP/s and seconds. This performance is computed from the mean

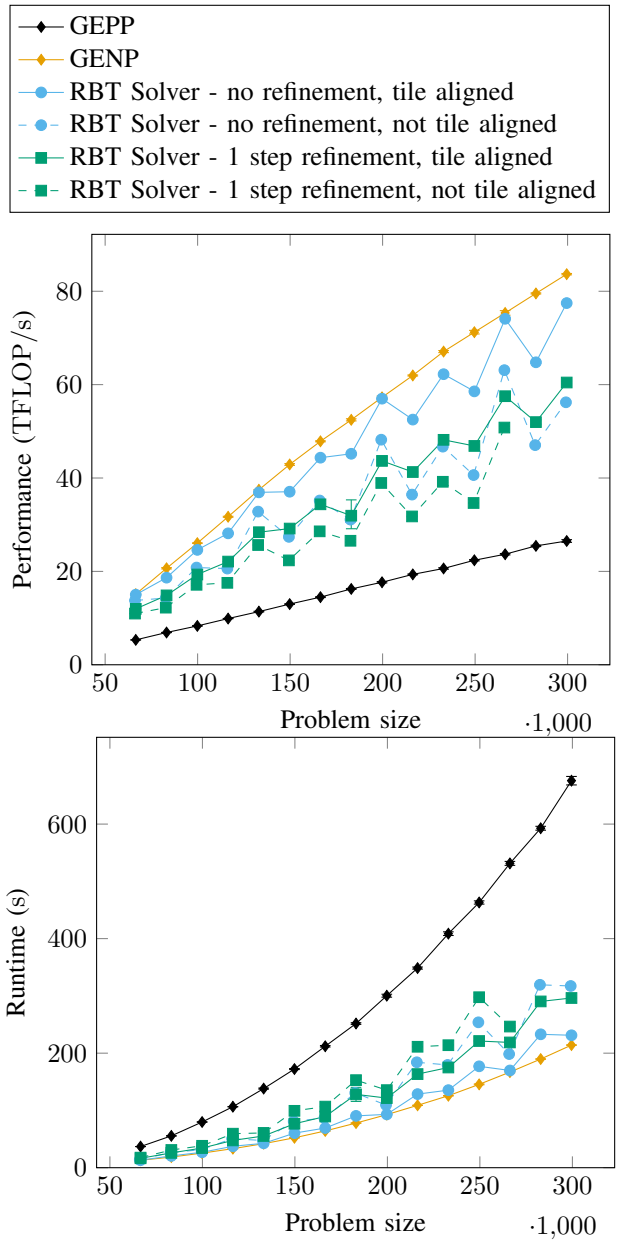


Fig. 8: Performance of various solvers on 8 nodes of Summit.

runtime of 3 executions and a flop count of $\frac{2}{3}n^3$. Additionally, 99.9 % confidence intervals are also included in the figure but are less than 1 TFLOP/s in each direction for all but one case. Finally, the RBT-based solver was also tested without iterative refinement to better understand the sources of overhead.

The RBT solver was successfully able to outperform GEPP. For problems of size greater than 200 000, the tile-aligned RBT solver had speedups ranging between 1.82 and 2.68 times faster, and the non-tile-aligned solver had speedups ranging between 1.56 and 2.26 times faster. For reference, GENP had speedups ranging between 3.12 and 3.25 times faster for those problem sizes.

Next, consider the jagged performance of the RBT-based results. For the tile aligned problems, the first, fifth, ninth, and thirteen sizes were all aligned such that no communication was needed for RBT application. The RBT-based solver without refinement was able to perform very closely to GENP at these sizes with other tile sizes having worse performance. This indicates that inter-process communication causes most of the cost to apply the transforms. So, optimization efforts of the transform application should focus on inter-process communication. The iterative refinement also introduced a noticeable overhead. Optimization of the iterative refinement step likely involves increased utilization of the GPUs and improving the performance of the individual kernels.

At the time of writing, SLATE was not effective at keeping the GPUs busy in its LU factorizations, which results in performance significantly below the theoretical peak of 336 TFLOP/s for our 8-node configuration. The issue is known to SLATE's developers; however, improvements to this performance were still in development and unable to be tested.

Our performance results are reasonably similar to previous works. First, the prior work on a distributed system solved symmetric problems and was tested on a 16-node cluster, each with eight cores of Intel Nehalem and no GPUs [7]. The relative performances of the non-pivoted factorization and the RBT-transformed solver shown in Baboulin et al.'s Fig. 10 are similar to those in Fig. 8 with the RBT solver's performance depending on whether the tiles are aligned with the butterfly's structure. However, when using iterative refinement, the performance of Baboulin et al.'s solver peaks at almost that of the non-pivoted solver, while our solver performs noticeably worse. Furthermore, Baboulin et al.'s solver runs entirely on the CPU, while our solver uses the GPU more often in the factorization than in the refinement, unbalancing the relative performances. Second, the prior work involving GPU-accelerated systems and non-symmetric problems was tested on a single-node with twelve cores of Intel Westmere and either a NVIDIA K20 GPU or an Intel Xeon Phi co-processor 7120 [11]. There, the RBT solver outperforming GEPP, although not to the degree that occurred in our work; additionally, the RBT overhead was consistently similar to the non-communicating, tile-aligned case from this work. Finally, iterative refinement added little overhead on the NVIDIA GPU, compared to our work where the CPU computed a significant part of the refinement. (The Intel Xeon Phi system did not have an optimized iterative refinement, making it unsuitable for comparison.)

VII. Conclusion

Our proposed approach of transforming the problem with RBTs then solving with GENP provided a speedup over GEPP between 1.56 and 2.68 times faster. However, the approach described here is unable to solve all problems. Those cases can revert to a regular GEPP implementation and only introduce an overhead between 37 % and 64 %, based on the results for larger problem sizes. Thus, this approach should provide a performance benefit, even if it cannot solve one in three problems.

There are a variety of future directions for this work. First, it may be possible to reduce the overhead of applying the RBT, particularly when the transforms and tiles are not in alignment. One proposed approach is to only apply an RBT to one-side of the problem [8], which would significantly reduce the complexity of communications. It has received limited attention but has had success for factoring sparse problems. Relatedly, there have been other transformations proposed for enabling GENP [4], and it may be beneficial to explore whether other transforms could perform better or be used to augment the RBT. For example, a randomized permutation would likely destroy the structure of the problematic matrix discussed in Sec. VI-A. It would also be interesting to investigate the strong scaling of the RBT solver compared to GEPP. The lack of a monolithic, highly synchronous, panel factorization in GENP may make it highly effective on extremely parallel systems.

ACKNOWLEDGMENTS

This material is based upon work supported by the University of Tennessee grant MSE E01-1315-038 as Interdisciplinary Seed funding and in part by UT Battelle subaward 4000123266. This material is also based upon work supported by the National Science Foundation under OAC Grant No. 2004541.

This research used the computational resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 provided by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] A. George and K. D. Ikramov, "Gaussian elimination is stable for the inverse of a diagonally dominant matrix," *Mathematics of Computing*, vol. 73, no. 246, pp. 653–657, 2004.

[2] T. Joffrain, E. S. Quintana-Ortí, and R. A. van de Geijn, "Rapid development of high-performance out-of-core solvers," in *Applied Parallel Computing. State of the Art in Scientific Computing*, ser. Lecture Notes in Computer Science, J. Dongarra, K. Madsen, and J. Waśniewski, Eds. Berlin, Heidelberg: Springer, 2006, pp. 413–422, doi: 10.1007/11558958_49.

[3] L. Grigori, J. W. Demmel, and H. Xiang, "Communication avoiding Gaussian elimination," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08, Nov. 2008, pp. 1–12, doi: 10.1109/SC.2008.5214287.

[4] V. Y. Pan and L. Zhao, "Numerically safe Gaussian elimination with no pivoting," *Linear Algebra and its Applications*, vol. 527, pp. 349–383, Aug. 2017, doi: 10.1016/j.laa.2017.04.007.

[5] D. S. Parker, "Random butterfly transformations with applications in computational linear algebra," Computer Science Department, UCLA, Tech. Rep. CSD-950023, 1995.

[6] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov, "Accelerating linear system solutions using randomization techniques," *ACM Transactions on Mathematical Software*, vol. 39, no. 2, pp. 1–13, Feb. 2013, doi: 10.1145/2427023.2427025.

[7] M. Baboulin, D. Becker, G. Bosilca, A. Danalis, and J. Dongarra, "An efficient distributed randomized algorithm for solving large dense symmetric indefinite linear systems," *Parallel Computing*, vol. 40, no. 7, pp. 213–223, Jul. 2014, doi: 10.1016/j.parco.2013.12.003.

[8] M. Baboulin, X. S. Li, and F.-H. Rouet, "Using random butterfly transformations to avoid pivoting in sparse direct methods," in *High Performance Computing for Computational Science – VECPAR 2014*, ser. Lecture Notes in Computer Science, M. Daydé, O. Marques, and K. Nakajima, Eds. Cham: Springer International Publishing, 2015, pp. 135–144, doi: 10.1007/978-3-319-17353-5_12.

[9] T. Trogdon, "On spectral and numerical properties of random butterfly matrices," *Applied Mathematics Letters*, vol. 95, pp. 48–58, 2019, doi: 10.1016/j.aml.2019.03.024.

[10] M. Baboulin, R. A. Dongarra Jack, T. Stanimire, and Y. Ichitaro, "Dense symmetric indefinite factorization on GPU accelerated architectures," in *Parallel Processing and Applied Mathematics. PPAM 2015*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, J. Kitowski, and K. Wiatr, Eds. Springer, 2016, doi: 10.1007/978-3-319-32149-3_9.

[11] M. Baboulin, A. Khabou, and A. Rémy, "A randomized LU-based solver using GPU and Intel Xeon Phi accelerators," in *Euro-Par 2015: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander, Eds. Cham: Springer International Publishing, 2015, pp. 175–184, doi: 10.1007/978-3-319-27308-2_15.

[12] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*. Philadelphia: SIAM, 1992.

[13] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "SLATE: Design of a modern distributed and accelerated linear algebra library," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. Denver, Colorado: Association for Computing Machinery, Nov. 2019, pp. 1–18, doi: 10.1145/3295500.3356223.

[14] N. J. Higham, "The matrix computation toolbox," 2002, http://www.ma.man.ac.uk/~higham/mctoolbox.

[15] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki, "A survey of recent developments in parallel implementations of Gaussian elimination," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 5, pp. 1292–1309, 2015, doi: 10.1002/cpe.3306.

APPENDIX
ARTIFACT DESCRIPTION

Summary of the Experiments Reported

We ran SLATE's `gesv`, `gesv_nopiv`, and `gesv_rbt` tests on Oak Ridge National Laboratory's Summit supercomputer using a modified version of SLATE, as described in Sec. VI.

Artifact Availability

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs where artifacts are available: https://doi.org/10.5281/zenodo.4006641

Baseline Experimental Setup, and Modifications Made for the Paper

Relevant Hardware Details: The Summit supercomputer at the Oak Ridge National Laboratory. We used eight nodes, each containing two sockets, with one 22-core IBM POWER9 processors. Each socket was connected to three NVIDIA Volta V100 GPUs each with $16\,\mathrm{GiB}$ HBM 2, and $256\,\mathrm{GiB}$ DDR4. The network consists of dual-rail EDR InfiniBand connected in a Non-blocking Fat Tree topology.

Operating systems and versions: Red Hat Enterprise Linux version 7.6

Compilers and versions: GCC 8.1.1, CUDA 10.1.243

Libraries and versions: SLATE commit c9266da, Spectrum MPI 10.3.1.2, ESSL 6.1.0-2, Netlib LAPACK 3.8.0

Key algorithms: LU factorization with and without pivoting, recursive butterfly transforms

Modifications made for paper: SLATE was modified to provide the RBT and iterative refinement. Additionally, the performance of SLATE's `getrf_nopiv` was improved. Finally, matrix generators were added to SLATE's tester for random, binary matrices, as well as the structured matrices `fielder`, `orthog`, and `gfpp`. While not strictly a software modification, the backward error provided by the SLATE tester is

$$\epsilon_{\text{SLATE}} = \frac{\|r\|_1}{\|A\|_1 \|x\|_1 n}$$

which we scaled to

$$\epsilon = \frac{\|r\|_1}{\|A\|_1 \|x\|_1}$$

in order to provide a better comparison with machine precision.