ENHANCING BLOCK CIMMINO FOR SPARSE LINEAR SYSTEMS WITH DENSE COLUMNS VIA SCHUR COMPLEMENT *

F. SUKRU TORUN†, MURAT MANGUOGLU‡, AND CEVDET AYKANAT§

Abstract. The block Cimmino is a parallel hybrid row-block projection iterative method successfully used for solving general sparse linear systems. However, the convergence of the method degrades when angles between subspaces spanned by the row-blocks are far from being orthogonal. The density of columns as well as the numerical values of their nonzeros are more likely to contribute to the non-orthogonality between row blocks. We propose a novel scheme to handle such "dense" columns. The proposed scheme forms a reduced system by separating these columns and the respective rows from the original coefficient matrix and handling them via Schur complement. Then, the angles between subspaces spanned by the row-blocks of the reduced system are expected to be closer to orthogonal and the reduced system is solved efficiently by the block Conjugate Gradient accelerated block Cimmino in fewer iterations. We also propose a novel metric for selecting "dense" columns considering the numerical values. The proposed metric establishes an upper bound on the sum of inner–products between row-blocks. Then, we propose an efficient algorithm for computing the proposed metric for the columns. Extensive numerical experiments for a wide range of linear systems confirm the effectiveness of the proposed scheme by achieving fewer iterations and faster parallel solution time compared to the classical CG accelerated block Cimmino algorithm.

Key words. Schur Complement, parallel block Cimmino, hybrid methods, Krylov subspace methods, row projection methods.

AMS subject classifications. 65F10, 65F50, 05C50, 65Y05

1. Introduction.

In computational mathematics, the row projection methods are one of the most fundamental types of iterative methods for solving the system of linear equations of the form

$$\mathcal{A}x = f, \tag{1.1}$$

where $\mathcal{A}$ is an $n \times n$ sparse unsymmetric nonsingular matrix, x and f are the unknown and right-hand side (rhs) vectors, respectively. Kaczmarz [33] and Cimmino [15] are the two main variations of the row projection method, where the solution is computed iteratively through projections. Kaczmarz solves the system using the product of row projections, whereas Cimmino solves the system using sum of the projections. Cimmino algorithm has an advantage on parallel platforms since it obtains row projections independently in each iteration. Summation of these projections is the only part that requires inter-processor communication.

The Cimmino algorithm has been studied extensively [4, 24, 43]. In the Cimmino algorithm, the number of iterations required for convergence can be large. The block version of the Cimmino algorithm, which is a hybrid method in the sense that it combines direct and iterative methods, is proposed [7, 13] to improve the convergence rate. At each iteration of the block Cimmino algorithm, the minimum 2-norm solution of underdetermined linear systems of equations is computed via a direct method, in which the coefficient matrices are the row blocks of $\mathcal{A}$.

*Submitted to the editors October 1, 2021.

Funding: The second author was supported by the BAGEP Award of the Science Academy.

†Department of Computer Engineering, Ankara Yildirim Beyazit University, Ankara, 06020, Turkey (fstorun@aybu.edu.tr).

‡Department of Computer Engineering, Middle East Technical University, 06800, Ankara, Turkey (manguoglu@ceng.metu.edu.tr).

§Department of Computer Engineering, Bilkent University, 06800, Ankara, Turkey (aykanat@cs.bilkent.edu.tr).

The convergence rate of the block Cimmino algorithm depends on the orthogonality between subspaces spanned by the row-blocks. In [19], convergence of the block Cimmino algorithm is studied and some partitioning methods are proposed to improve the structural and numerical orthogonality among block rows. Recently, we proposed a new row-block partitioning method (GRIP) [46] which directly aims to increase numerical orthogonality among subspaces spanned by row-blocks by using a graph theoretical model. In GRIP, the partitioning objective of minimizing the cut-size encodes minimizing the sum of row-inner products between different block rows. We showed that the row-block partitioning obtained with GRIP significantly reduces the required number of iterations for the block Cimmino algorithm.

In this work, we propose a new scheme to reduce the number of iterations for sparse linear systems whose coefficient matrices have some dense columns. In the block Cimmino, if all nonzero entries of a column fit into one row-block, then that column does not disturb the numerical orthogonality between subspaces spanned by the row-blocks. On the other hand, if a column of a matrix contains at least two nonzero entries which are placed in distinct two row blocks, this column can decrease the orthogonality between the subspaces. Therefore, in this work, we study the matrices having dense columns since dense columns in the matrix are more likely to have nonzeros in distinct row-blocks and this can decrease the orthogonality between subspaces spanned by the row-blocks.

The proposed scheme is a hybrid method which is based on the Schur complement by separating some columns (possibly dense ones) from the solution process of the block Cimmino method to increase the orthogonality among subspaces spanned by the row-blocks. We obtain a column permutation so that those columns that hamper the orthogonality are placed in the (1,2)-block if the matrix is partitioned into 2x2 blocks. We apply the permutation symmetrically. The system involving (1,1)-block as the coefficient matrix is solved with the block Cimmino algorithm which requires fewer iterations since the subspaces spanned by the row-blocks of the (1,1)-block are expected to be closer to orthogonal to each other. As will be explained later, the proposed scheme requires the solution of a system with multiple right-hand side vectors and a small dense system which can be formed explicitly and then solved with a direct solver.

The challenge of handling dense rows and/or columns in sparse linear systems has been extensively studied in the context of linear least squares (LLS) problems [1, 8, 11, 23, 27, 30, 35, 40, 42, 44, 45, 48]. In these problems, dense rows and/or columns cause a dramatic loss of efficiency due to catastrophic fill-in in the factorization. Handling those dense rows/columns via block factorization which results in a Schur complement system is also proposed in [3, 25, 39, 41] for solving such LLS problems. We propose, however, a scheme for identifying "dense" columns in the context of solving non-symmetric linear systems, specifically for improving the convergence rate of the block Cimmino method and tackle these "dense" columns separately by adopting a block LU factorization scheme for solving the system which results in a small Schur complement matrix.

To this end, we also propose two metrics for selecting columns in the proposed scheme. The first metric simply considers the number of nonzeros in the columns for selection. The second metric considers not only the number of nonzeros but also pairwise sum of the values of the nonzeros in the columns for selection. Although computing the first metric can easily be done in linear time in the number of nonzeros in a column, a naive implementation of the second metric runs in square of the number of nonzeros in a column. We also propose an efficient algorithm that enables

computing the second metric in linear time for each column. We show that the second metric outperforms the first one in terms of the required number of iterations for convergence.

There are several advantages of the proposed scheme besides decreasing the number of iterations which leads to faster parallel solution time. Since we use the block iterative method to accelerate the block Cimmino with multiple rhs vectors, the proposed scheme has an additional improvement in the number of iterations due to the faster convergence of eigenvectors associated with the smallest eigenvalues [36]. Another advantage is that the proposed scheme can incur less communication overhead in iterations of the block Cimmino algorithm since denser columns likely to be dropped from the coefficient matrix. This can, in turn, translate into improved parallel time per iteration of the block Cimmino. The other advantage is that less factorization time for the block Cimmino is needed since we have smaller row-blocks to factorize and smaller fill-in. The experimental results performed on a shared-memory and a distributed-memory platforms for a wide range of linear systems validate the effectiveness of the proposed scheme to solve linear systems with "dense" columns through fewer iterations and faster parallel solution time.

The rest of the paper is organized as follows. Section 2 provides the background on block Cimmino. In Section 3, we discuss the proposed scheme along with its solution phases, criteria for selecting columns, effect of dense columns on the spectrum of the iteration matrix corresponding to block Cimmino, and parallelization and implementation details of the proposed scheme. The extensive numerical experiments are expressed in Section 4. Finally, Section 5 concludes the paper.

2. Block Cimmino Algorithm. In the classical block Cimmino algorithm, the original system (1.1) is partitioned into K row-blocks as follows;

$$\begin{pmatrix} \mathcal{A}_1 \\ \mathcal{A}_2 \\ \vdots \\ \mathcal{A}_K \end{pmatrix} x = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_K \end{pmatrix}, \tag{2.1}$$

where $K \leq n$. Here $\mathcal{A}_k$ is row-block of size $m_k \times n$ and f_k is a column vector of size m_k. The solution is obtained iteratively by summing the projections on the subspaces spanned by $\mathcal{A}_k^T$, where $\mathcal{A}_k^T$ denotes the transpose of $\mathcal{A}_k$. The pseudocode of block Cimmino is presented in Algorithm 2.1, where $\mathcal{A}_k^+ = \mathcal{A}_k^T(\mathcal{A}_k\mathcal{A}_k^T)^{-1}$ is the pseudo-inverse of $\mathcal{A}_k$ and ϕ is a relaxation parameter. In the parallel implementation, line 4 can be computed perfectly in parallel without incurring any communication, whereas at line 6, communication is needed to sum up the δ_k vectors. At line 4 of the algorithm, projection onto the range of $\mathcal{A}_k^T$, i.e., $\mathcal{A}_k^+\mathcal{A}_k$, is implicitly computed. If subspaces spanned by $\mathcal{A}_k$ row-blocks are completely orthogonal to each other, the sum of these projections gives the projection onto range of $\mathcal{A}^T$. In this case, the algorithm needs only one iteration if the projections are computed accurately.

The iteration equations of block Cimmino can be reformulated as follows:

$$\begin{aligned} x^{(j+1)} &= x^{(j)} + \phi \sum_{k=1}^{K} \mathcal{A}_k^+ \left(f_k - \mathcal{A}_k x^{(j)}\right) \\ &= \left(I - \phi \sum_{k=1}^{K} \mathcal{A}_k^+ \mathcal{A}_k\right) x^{(j)} + \phi \sum_{k=1}^{K} \mathcal{A}_k^+ f_k \\ &= (I - \phi H) x^{(j)} + \phi \sum_{k=1}^{K} \mathcal{A}_k^+ f_k \end{aligned} \tag{2.2}$$

Algorithm 2.1 Block Cimmino method

1: Choose $x^{(0)}$
2: **while** $j = 0, 1, 2, \ldots,$ until convergence **do**
3: **for** $k = 1, \ldots, K$ **do**
4: $\delta_k = \mathcal{A}_k^+ (f_k - \mathcal{A}_k x^{(j)})$
5: **end for**
6: $x^{(j+1)} = x^{(j)} + \phi \sum_{k=1}^{K} \delta_k$
7: **end while**

where $(I - \phi H)$ is the iteration matrix for the block Cimmino algorithm and H is a symmetric and positive definite matrix since it is the sum of projections spanned by the subspaces of $\mathcal{A}_k$ row-blocks which are assumed to have full row rank. We note that the system

$$\phi H x = \phi \textstyle\sum_{k=1}^{K} \mathcal{A}_k^+ f_k \tag{2.3}$$

has the same solution vector x in (2.2). Therefore, Conjugate Gradient (CG) accelerated block Cimmino algorithm (CG-BC) [6, 13, 20] solves (2.3) iteratively via the CG method. Since ϕ appears on both sides of the equation it does not affect the convergence of CG. Algorithm 2.2 shows CG-BC algorithm. The convergence rate of Algorithm 2.2 is related to the eigenvalues of the coefficient matrix H whose eigenvalues are correlated with the principal angles between subspaces spanned by $\mathcal{A}_k^T$ [6, 13]. If these principal angles are wider, then more eigenvalues of H cluster around one. This leads to fewer iterations for solving (2.3).

Algorithm 2.2 CG-BC algorithm [13, 37, 50]

1: Choose $x^{(0)}$
2: $r^{(0)} = \sum_{k=1}^{K} \mathcal{A}_k^+ f_k - H x^{(0)}$
3: $p^{(0)} = r^{(0)}$
4: **while** $j = 0, 1, 2, \ldots,$ until convergence **do**
5: $\psi^{(j)} = H p^{(j)}$
6: $\alpha^{(j)} = ({r^{(j)}}^T r^{(j)}) / ({p^{(j)}}^T \psi^{(j)})$
7: $x^{(j+1)} = x^{(j)} + \alpha^{(j)} p^{(j)}$
8: $r^{(j+1)} = r^{(j)} - \alpha^{(j)} \psi^{(j)}$
9: $\beta^{(j)} = ({r^{(j+1)}}^T r^{(j+1)}) / ({r^{(j)}}^T r^{(j)})$
10: $p^{(j+1)} = r^{(j+1)} + \beta^{(j)} p^{(j)}$
11: **end while**

There are several row-block partitioning methods [19, 37, 46] to widen those principal angles for faster convergence. In [46], we proposed a novel graph theoretical row-block partitioning method, GRIP, and showed the effectiveness of GRIP whose objective corresponds to increasing principal angles between the subspaces.

In each iteration of CG-BC, we use a direct method to compute projections (at lines 2 and 5 in Algorithm 2.2). There are several approaches to compute the minimum 2-norm solution of the underdetermined system whose coefficient matrix is $\mathcal{A}_k$, some are; normal equations [29], seminormal equations [28], QR factorization [29] and augmented system approach [5]. Solution with normal and seminormal equations may encounter numerical difficulties when the problem is ill-conditioned [17] and although

the QR factorization is numerically more stable, it is computationally costly. Therefore, we use the augmented system approach as in [6, 19, 20, 46], which requires the solution of a sparse square symmetric linear system that can be performed effectively by using a sparse direct solver. The augmented system approach solves the square symmetric linear system in the form of

$$\begin{pmatrix} I & \mathcal{A}_k^T \\ \mathcal{A}_k & 0 \end{pmatrix} \begin{pmatrix} \delta_k \\ \varsigma_k \end{pmatrix} = \begin{pmatrix} 0 \\ r_k \end{pmatrix}, \tag{2.4}$$

where δ_k is the minimum 2-norm solution of

$$\mathcal{A}_k \delta_k = r_k, \qquad r_k = f_k - \mathcal{A}_k x^{(j)}. \tag{2.5}$$

The system (2.4) is repeatedly solved at each iteration and for each row-block.

3. The proposed scheme. In this section, we first present the formulation of the proposed scheme together with its solution method. Then, we suggest two metrics for selecting columns for the proposed scheme and study the effects of those metrics on the eigenvalue spectrum of H. Finally, we explain the parallelization and implementation details of the proposed scheme.

3.1. Formulation. The proposed scheme adopts the Schur complement approach by separating some columns from the solution process of the block Cimmino method. In the Schur complement approach, the coefficient matrix $\mathcal{A}$ in Equation 1.1 is first permuted symmetrically and then partitioned into 2×2 matrix blocks

$$\mathcal{P}\mathcal{A}\mathcal{P}^{\mathcal{T}} = \begin{bmatrix} A & B \\ C^T & D \end{bmatrix}, \tag{3.1}$$

where $\mathcal{P}$ is a permutation matrix. Here A and D are respectively $n_A \times n_A$ and $s \times s$ square matrices, whereas B and C are $n_A \times s$ rectangular matrices. Since $\mathcal{A}$ is an $n \times n$ matrix, we have

$$n = n_A + s.$$

Vectors x and f are also permuted and partitioned conformably with $\mathcal{A}$. Therefore the linear system $(\mathcal{P}\mathcal{A}\mathcal{P}^{\mathcal{T}})\mathcal{P}x = \mathcal{P}f$ can be written as

$$\begin{bmatrix} A & B \\ C^T & D \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix}, \tag{3.2}$$

where

$$\mathcal{P}x = \begin{bmatrix} y \\ z \end{bmatrix} \text{ and } \mathcal{P}f = \begin{bmatrix} u \\ v \end{bmatrix}. \tag{3.3}$$

Here y and u are column vectors of size n_A, whereas z and v are column vectors of size s. If A and D are not singular matrices, the block LU factorization of (1.1) becomes

$$\begin{bmatrix} I & 0 \\ C^T A^{-1} & I \end{bmatrix} \begin{bmatrix} A & B \\ 0 & S \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix}, \tag{3.4}$$

where $S = D - C^T A^{-1} B$ is called Schur complement. We have the following equations from (3.4):

$$\begin{aligned} Sz &= v - C^T A^{-1} u \\ Ay &= u - Bz. \end{aligned} \tag{3.5}$$

Since the two linear systems $AF = B$ and $Ag = u$ have the same coefficient matrix, these systems can be combined into one system which can be solved at once

$$A[F\ g] = [B\ u]. \tag{3.6}$$

Then, we form the Schur complement

$$S = (D - C^T F) \tag{3.7}$$

via sparse matrix-dense matrix multiplication and sparse matrix-dense matrix subtraction kernels. Then, we solve the following system of equations for z,

$$Sz = v - C^T g. \tag{3.8}$$

Note that S is formed explicitly since s (size of S) is assumed to be small. Therefore, we use a dense direct solver to solve this system. Alternatively, if s is large, one can solve the Schur complement system (3.8) using a preconditioned iterative scheme [10, 26] without forming S explicitly. In such iterative scheme, the matrix-vector multiplications of the form $q = Sw$ are required at each iteration. These multiplications can be performed without forming S by multiplying the vector w with $D - C^T A^{-1} B$. Solving the system without forming S explicitly would require the solution of a larger linear system with the coefficient matrix A at each iteration. Since s is a user controlled parameter and typically a small s is required, in our implementation we form S explicitly.

In the last step, we obtain y as

$$y = g - Fz \tag{3.9}$$

via dense matrix-vector and vector-vector operations which are BLAS level-2 and level-1 operations, respectively. Finally, the solution of (1.1) is obtained via permuting back (3.3).

3.2. Criteria for Selecting Columns. We propose two metrics for selecting columns from $\mathcal{A}$ and forming B in the proposed scheme. To attain the best performance from the scheme, we need to work with the columns which affect orthogonality the most among subspaces spanned by the row blocks. We know that if all nonzero elements of a column can fit into one row-block segment, that column does not deteriorate the numerical orthogonality. Therefore, the denser columns in the matrix are more likely to cause such adverse effects on the conditioning of H than the other columns. For this reason, a straightforward metric, *colnnz*, considers the columns that contain the largest number of nonzeros.

The second metric, *ppsum*, takes the numerical values of nonzeros into account. Given a vector $c \in \mathbb{R}^n$, the outer product matrix is defined as

$$T = cc^T. \tag{3.10}$$

Based on T, we introduce a function, *ppsum()* as

$$ppsum(c) = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} |T_{ij}|, \tag{3.11}$$

or equivalently

$$ppsum(c) = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} |c_i||c_j|. \tag{3.12}$$

That is, $ppsum(c)$ is equal to the sum of the pairwise products of the absolute values of the nonzeros in column c. The motivation of $ppsum$ is based on the GRIP [46] partitioning method. In GRIP, firstly the row–inner–product graph $\mathcal{G}_{\text{RIP}}(\mathcal{A}) = (\mathcal{V}, \mathcal{E})$ of matrix $\mathcal{A}$ is constructed, where each row r_i in $\mathcal{A}$ is represented by a vertex $v_i \in \mathcal{V}$. For each nonzero inner–product between row pairs $\langle r_i, r_j \rangle > 0$, an edge $(v_i, v_j) \in \mathcal{E}$ is added with cost $|\langle r_i, r_j \rangle|$. Assuming the rows of the coefficient matrix $\mathcal{A}$ are normalized to have unit length, cost of edge (v_i, v_j) corresponds to $cos(\theta) = |\langle r_i, r_j \rangle|$.

Then, $\mathcal{G}_{\text{RIP}}(\mathcal{A})$ is partitioned into K disjoint vertex parts $\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ by maintaining balance over parts and minimizing the cutsize. Here, the cutsize refers to the sum of costs of cut-edges which are the edges that connect different parts. In matrix theoretical view, K vertex parts induce K row-blocks with almost equal number of rows, where minimizing the cutsize corresponds to minimizing the sum of inner products between $\mathcal{A}_i$ row blocks. In other words, GRIP aims to increase the orthogonality between subspaces spanned by the row blocks, this in turn leads to fewer number of iterations in the Block Cimmino. In the best case of zero cutsize, we have fully orthogonal row-blocks that enable block Cimmino to converge only in one iteration if the projections are computed in exact arithmetic.

Let $\mathcal{C}$ denote the set of columns in $\mathcal{A}$. Considering the sum of all ppsum values, and by changing the order of summation, we have

$$\sum_{c \in \mathcal{C}} ppsum(c) = \sum_{c \in \mathcal{C}} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ic} a_{jc}| \tag{3.13}$$

$$= \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \sum_{c \in \mathcal{C}} |a_{ic} a_{jc}|. \tag{3.14}$$

Then by using the triangle inequality and the definition of inner product, we obtain

$$\sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} |\langle r_i, r_j \rangle| = \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} \Big(\Big| \sum_{c \in \mathcal{C}} a_{ic} a_{jc} \Big| \Big) \leq \sum_{c \in \mathcal{C}} ppsum(c). \tag{3.15}$$

Hence, the sum of ppsum values of all columns is an upper bound on the inner products of all rows. Moreover, the cutsize of GRIP, by definition, is at most the summation of all row inner-products. Thus, summation of ppsum values is an upper bound on the cutsize of GRIP. Therefore, separating the columns with the largest $ppsum$ values before the construction of $\mathcal{G}_{\text{RIP}}$ is expected to decrease the cutsize after the partitioning.

A naive implementation of $ppsum(c)$ takes $O(n^2)$ time. However, this running time can easily be reduced to $O(nnz^2(c))$ time by exploiting the sparsity of column c. Here $nnz(c)$ denotes the number of nonzeros in column c. In this work, we propose an efficient algorithm for computing $ppsum(c)$ in linear time in the number of nonzeros of column c, i.e., in $\theta(nnz(c))$ time. The proposed algorithm is based on factoring

the c_i term out of the second summation in the double summation expression given in (3.12). That is,

$$ppsum(c) = \sum_{i=1}^{n} |c_i| \left(\sum_{j=1}^{n} |c_j| - |c_i| \right). \tag{3.16}$$

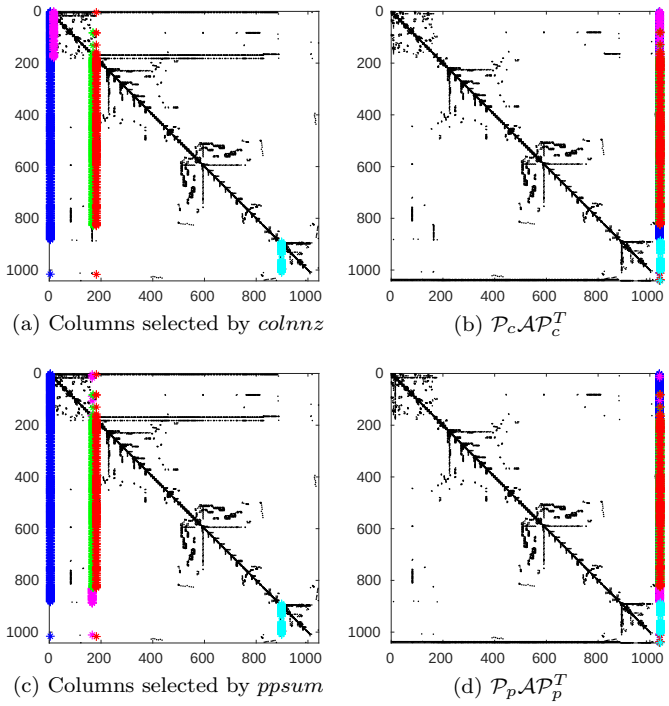
In this way, the summation $\sum_{j=1}^{n} |c_j|$ inside the parenthesis can be computed only once and used for each different c_i. Algorithm 3.1 shows the steps of the algorithm. The first inner for loop (lines 3–5) computes the sum of the absolute value of nonzeros in the current column in *colSum* which corresponds to the summation term inside the parenthesis of (3.16). The second inner for loop (lines 6–8) computes *ppsum* value for the current column by using *colSum*. The running time of outer for loop is $\theta(nnz)$ (lines 1–9). At line 10, the selection operation can be efficiently done in $O(n + s \log n)$ time by using binary heap implementation of priority queue. Here, $O(n)$ time comes from the Build-Heap operation and $O(s \log n)$ comes from s successive Extract-Max operation performed on the heap.

Algorithm 3.1 Selecting Columns with *ppsum*

```
1:  for each column c ∈ A do
2:      colSum = 0
3:      for each nonzero a_ic in column c do
4:          colSum = colSum + |a_ic|
5:      end for
6:      for each nonzero a_ic in column c do
7:          ppsum[c] = ppsum[c] + |a_ic| × (colSum − |a_ic|)
8:      end for
9:  end for
10: Select s columns with the largest ppsum values
```

3.3. Effect of dense columns on the eigenvalue spectrum of H. We illustrate the effect of dense columns on the orthogonality between subspaces spanned by row blocks via studying the eigenvalue spectrum of H (in Equation (2.3)) for a toy problem, `rajat04`. `rajat04` is a sparse unsymmetric nonsingular matrix (arising in circuit simulation) of size $1{,}041 \times 1{,}041$ with 8,725 nonzeros from the SuiteSparse Sparse Matrix Collection [16]. In this matrix, the average number of nonzeros in the columns is 8.3 and the densest column has 642 (62% of n) nonzeros. The number of nonzeros in the next four densest columns are 438, 258, 85 and 81. Figure 3.1 shows nonzero patterns of the matrix and the matrices after being symmetrically permuted with $\mathcal{P}_c$ for *colnnz* and with $\mathcal{P}_p$ for *ppsum*. Figure 3.1 shows the columns selected according to the *colnnz* and *ppsum* metrics as highlighted in colors. For this matrix, *colnnz* and *ppsum* select the column indices of $\{4, 17, 169, 182, 898\}$ and $\{4, 166, 169, 182, 898\}$, respectively. Even though Figures 3.1a and 3.1c visually do not look much different, one change in the selected columns (column 17 instead of column 166) incurs an improvement on the eigenvalue distribution for *ppsum*. As seen in Figures 3.1b and 3.1d, the selected columns are swapped with the rightmost columns of the matrix and the respective rows are symmetrically swapped with the rows at the very bottom of the matrix.

Figure 3.2 shows the spectra of the H matrices for the original matrix $\mathcal{A}$, and for the A matrices after five rightmost columns/bottom rows separated (as in Equation



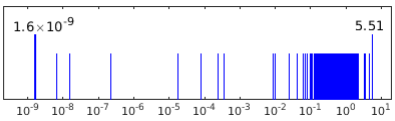
(a) Columns selected by *colnnz* (b) $\mathcal{P}_c\mathcal{A}\mathcal{P}_c^T$

(c) Columns selected by *ppsum* (d) $\mathcal{P}_p\mathcal{A}\mathcal{P}_p^T$

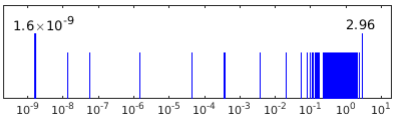
FIG. 3.1. *Sparsity patterns of original and permuted* `rajat04` *matrices. The columns selected according to colnnz and ppsum metrics are highlighted in colors.*

(3.1)) with *colnnz* and *ppsum*. All matrices are partitioned into eight row blocks uniformly. Comparison of Figures 3.2a and 3.2b shows that symmetrical dropping of five columns/rows according to *colnnz* shifts the largest and some small eigenvalues of H towards one and results in better clustering around one on the spectrum of H. On the other hand, with the *colnnz* method the smallest eigenvalue (1.6×10^{-9}) still appears at the end of the spectrum. Comparison of Figures 3.2b and 3.2c shows that *ppsum* achieves much better eigenvalue clustering around one than *colnnz*. With *ppsum*, the smallest eigenvalue (9.9×10^{-9}) is now much closer to one and the largest eigenvalue (2.45) is improved further.

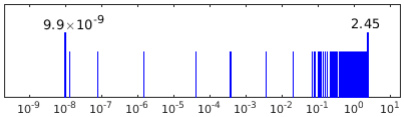
Moreover, we study the condition numbers of H matrices for each spectrum in Figure 3.2. For the original matrix the condition number of H is 3.3×10^9. *colnnz* and *ppsum* methods reduce the condition number of H to 1.8×10^9 and 2.4×10^8, respectively. In other words, *colnnz* and *ppsum* reduce the condition number 1.83 and 13.75 times, respectively. As seen in Figure 3.2c, *ppsum* attains an H matrix with a smaller condition number and better eigenvalue clustering around one which is expected to lead to a better convergence. For solving the resulting linear systems via block Cimmino, *ppsum* and *colnnz* require 62 and 75 iterations, respectively, while the original problem requires 171 iterations for the convergence. For *ppsum* and *colnnz*, we first extract matrices by separating some predefined columns and the respective rows. Then, on the resulting system, we apply the block Cimmino (CG-BC) algorithm, not the proposed scheme since block CG of the proposed scheme can further decrease the number of iterations for convergence due to a better detection of clusters of eigenvalues when the number of columns increases.



(a) Original matrix, $cond(H) = 3.3\times10^9$



(b) After 5 columns/rows separated with *colnnz*, $cond(H) = 1.8\times10^9$



(c) After 5 columns/rows separated with *ppsum*, $cond(H) = 2.4\times10^8$

FIG. 3.2. *Eigenvalue spectra of H matrix (with the smallest and largest eigenvalues)*

3.4. Parallelization and Implementation Details. Algorithm 3.2 shows the steps of the proposed parallel algorithm for solving a linear system with single *rhs* vector. In the algorithm, lines 1–9 constitute the preprocessing stage which is performed by processor p_1. In this stage, p_1 first reads the input matrix and then applies a column permutation $\mathcal{Q}$ to the linear system (1.1)

$$\mathcal{A}\mathcal{Q}^T\mathcal{Q}x = f \tag{3.17}$$

using the *hsl_mc64* [21] subroutine in HSL Mathematical Software Library [31] in order to maximize the product of the diagonal entries of $\mathcal{A}$. This ensures that the diagonal blocks of $\mathcal{A}\mathcal{Q}^T$ have zero-free main diagonals and hence more likely to be nonsingular. At line 3, p_1 performs diagonal scaling $\mathcal{D}^{-1}$ to the linear system (3.17) by rows, that is

$$\mathcal{D}^{-1}(\mathcal{A}\mathcal{Q}^T)(\mathcal{Q}x) = \mathcal{D}^{-1}f, \tag{3.18}$$

so that 2-norm of each row of the scaled system is equal to one.

At line 4, p_1 selects s columns in the coefficient matrix of (3.18) by either using *ppsum* or *colnnz* metric and then permute the system symmetrically

$$\mathcal{P}(\mathcal{D}^{-1}\mathcal{A}\mathcal{Q}^T)\mathcal{P}^T\mathcal{P}\mathcal{Q}x = \mathcal{P}\mathcal{D}^{-1}f \tag{3.19}$$

to move selected columns to the rightmost and the respective rows to the very bottom of the matrix. At line 5, p_1 extracts sub-matrices A, B, C^T and D of $\hat{\mathcal{A}} = \mathcal{P}(\mathcal{D}^{-1}\mathcal{A}\mathcal{Q}^T)\mathcal{P}^T$, where the selected columns form B, the respective rows form C and the intersection of B and C forms D. We note that the row scaling performed at line 3 is done in order to enable the GRIP method to obtain a better row partitioning in A. At line 6, p_1 partitions A into K row blocks using the GRIP method [46]. At line 7, p_1 applies the same partitioning on B and u of $\hat{f} = \mathcal{P}\mathcal{D}^{-1}f$ conformally with the row-block partition of A. Therefore, the i^{th} row of A and B sub-matrices as well as the i^{th} entry of the u sub-vector are assigned to the same processor. At line 8,

Algorithm 3.2 The proposed scheme for processor p_k

Input: $\mathcal{A}, f$
Output: y, z
1: **if** $k = 1$ **then**
2: Apply hsl_mc64 for column permutation $\mathcal{Q}^T$ (Eq. 3.17)
3: Perform 2-norm row scaling $\mathcal{D}^{-1}$ (Eq. 3.18)
4: Select s columns and apply symmetric permutation $\mathcal{P}$ (Eq. 3.19)
5: Extract submatrices A, B, C, D and subvectors u, v (Eq. 3.1)
6: Obtain K-way partition $\Pi(A)=\{A_1, \ldots, A_K\}$ on rows of A via GRIP ([46])
7: Partition B and u conformably with $\Pi(A)$ as $\{B_1, \ldots, B_K\}$ and $\{u_1, \ldots, u_K\}$
8: Send A_k, B_k, and u_k to p_k for $k = 2, \ldots, K$
9: **end if**
10: Construct and factorize the augmented system (Eq. 2.5)
11: Solve $A\,[F\ g] = [B\ u]$ using parallel BCG-BC (Algorithm 3.3)
12: **if** $k = 1$ **then**
13: Solve $Sz = v - C^T g$ using a dense direct solver (Eqn. 3.8)
14: $y = g - Fz$
15: **end if**

p_1 sends A_k, B_k and u_k to p_k for $k = 2, \ldots, K$. After this step, each processor p_k including p_1 owns A_k, B_k and u_k for $k = 1, \ldots, K$.

Figure 3.3 displays a sample 4-way uniform row-block partitioning where blocks A_k, B_k, and u_k are assigned to processor p_k. In the figure, for the sake of better visualization, A- and B-matrix blocks of rows and u-vector blocks of entries that are assigned to the same processor are shown ordered consecutively. In the proposed scheme, we choose relatively small s values (justification is discussed later) which gives rise to small C^T and D sub-matrices. Therefore, the C^T and D matrices are not partitioned and only p_1 performs the associated computations which take relatively small amount of time with respect to the other parts of the scheme.

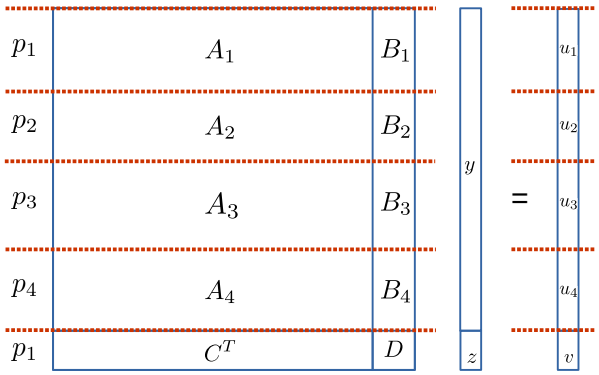


FIG. 3.3. *Row-block partitioning of the global system for 4 processors*

At lines 10–11, we obtain the unknown matrix $[F\ g]$ of the system $A[F\ g] = [B\ u]$ by solving $H[F\ g] = \sum_{k=1}^{K} A_k^+ [B\ u]$, where $H = \sum_{k=1}^{K} A_k^+ A_k$. At line 10, each processor p_k forms the coefficient matrix of the symmetric and indefinite augmented system and factorizes it into sparse Bunch-Kaufman-Parlett factors using MUMPS [2] multi-

Algorithm 3.3 BCG-BC algorithm [6, 37, 50]

Input: A_k, B_k, u_k
Output: X // $X = [F\ g]$

1: Choose $X^{(0)}$
2: $R^{(0)} = \sum_{k=1}^{K} A_k^+ [B_k\ u_k] - HX^{(0)}$
3: $[\gamma^{(0)}, \bar{R}^{(0)}] = stab\_1({R^{(0)}}^T R^{(0)})$ // Stabilization
4: $P^{(0)} = R^{(0)}$
5: **for** $j = 0, 1, 2, \ldots,$ until convergence **do**
6: $\quad [\beta^{(j)}, \bar{P}^{(j)}, \Psi^{(j)}] = stab\_2(P^{(j)}, HP^{(j)})$ // Stabilization
7: $\quad \lambda^{(j)} = {\beta^{(j)}}^{-T}$
8: $\quad X^{(j+1)} = X^{(j)} + \bar{P}^{(j)} \lambda^{(j)} (\Pi_{i=j}^{0} \gamma_i)$
9: $\quad R^{(j+1)} = \bar{R}^{(j)} - \Psi^{(j)} \lambda^j$ // $\Psi^{(j)} = H\bar{P}^{(j)}$
10: $\quad [\gamma^{(j+1)}, \bar{R}^{(j+1)}] = stab\_1({R^{(j+1)}}^T R^{(j+1)})$ // Stabilization
11: $\quad \alpha^{(j)} = \beta^{(j)} {\gamma^{(j+1)}}^T$
12: $\quad P^{(j+1)} = \bar{R}^{(j+1)} + \bar{P}^{(j)} \alpha^{(j)}$
13: **end for**
14: **function** $[\gamma, \bar{R}] = stab\_1(R^T R))$
15: $\quad \gamma = chol(R^T R)$ // Cholesky Decomposition
16: $\quad \bar{R} = R\gamma^{-1}$
17: **end function**
18: **function** $[\beta, \bar{P}, \Psi] = stab\_2(P, HP\ )$
19: $\quad \beta = chol(P^T HP)$ // Cholesky Decomposition
20: $\quad \bar{P} = P\beta^{-1}$
21: $\quad \Psi = HP\beta^{-1}$
22: **end function**

frontal parallel sparse direct solver. At line 11, rather than using simultaneous CG-BC iterations with multiple rhs vectors, we opt for the block version of the CG accelerated block Cimmino since block CG takes advantage of a better detection of clusters of eigenvalues [7, 36]. We use the stabilized version of the block CG accelerated block Cimmino (BCG-BC) [7, 37] implementation available in the open-source software package ABCD Solver [51]. The pseudocode of BCG-BC is shown in Algorithm 3.3. Since the stabilized block CG enforces $\bar{R}^{(j+1)}$ to have orthogonal columns by utilizing Cholesky decomposition (lines 6 and 10 in Algorithm 3.3), it does not have some of the convergence issues that non-stabilized block CG has, such as; a breakdown can occur due to division by zero [14], ${R^{(j+1)}}^T R^{(j+1)}$ matrices can become ill-conditioned or close to zero when one of the unknown vectors converges much faster than the others [37, 50].

If the systems at lines 15 and 19 in Algorithm 3.3 are ill-conditioned then the Cholesky decomposition may fail. If this happens, the ABCD solver employs a more stable alternative, the modified Gram-Schmidt process [12]. Rarely, even the modified Gram-Schmidt process can fail if the matrix is extremely ill-conditioned which is more likely to happen when s is large since large s values have the potential of increasing the likelihood of linear dependence between rhs vectors as also observed in [7, 37, 50]. Therefore, we choose relatively small s values in the light of those studies and our observations. Alternatively, a breakdown–free block CG [32] is also available.

In Algorithm 3.2, at line 13, p_1 constructs $S = (D - C^T F)$ in (3.7) via sparse

TABLE 4.1
Matrix properties

matrix name	kind	n	nnz	DC Density %	#parts
dc1	circuit simulation prb. sequence	116,835	766,396	98	8
trans4	circuit simulation prb. sequence	116,835	749,800	98	8
ASIC_100k	circuit simulation problem	99,340	940,621	93	8
mult_dcop_02	subsequent circuit simulation prb.	25,187	193,276	90	8
rajat30	circuit simulation problem	643,994	6,175,244	71	33
shermanACb	2D/3D problem	18,510	145,149	56	8
TSOPF_RS_b39_c30	power network problem	60,098	1,079,986	50	8
coupled	circuit simulation problem	11,341	97,193	21	8
nxp1	circuit simulation problem	414,604	2,655,880	13	21
circuit_4	circuit simulation problem	80,209	307,604	11	8
para-4	semiconductor device problem	153,226	2,930,882	4	8
appu	directed weighted random graph	14,000	1,853,104	2	8
ohne2	semiconductor device problem	181,343	6,869,939	2	10
av41092	2D/3D problem	41,092	1,683,902	2	8
ns3Da	computational fluid dynamics prb.	20,414	1,679,599	1	8
ted_A	thermal problem	10,605	424,587	1	8
hcircuit	circuit simulation problem	105,676	513,072	1	8
barrier2-10	subsequent semiconductor dev. prb.	115,625	2,158,759	1	8
torso1	2D/3D problem	116,158	8,516,500	1	8
std1_Jac3_db	chemical process simulation prb.	21,982	531,826	1	8

n: number of rows/columns, nnz: number of nonzeros, DC Density.: ratio of the number of nonzero in the densest column over n, #parts: number of row-blocks.

matrix kernels and then computes z by solving the much smaller system in (3.8) via the double precision general dense linear system solver *dgesv* subroutine in Linear Algebra PACKage (LAPACK). At line 14, y is computed via *dgemv* Basic Linear Algebra Subprograms (BLAS) Level 2 subroutine since in our case B is selected among the most dense columns and solving the linear system at line 11 is likely to introduce further nonzeros thus leading to a rather dense F.

4. Numerical Experiments. We conduct extensive numerical experiments to validate the performance of the proposed scheme. As a baseline method for the comparison, we use the classical Conjugate Gradient accelerated block Cimmino (CG-BC) algorithm since we assume there is only one *rhs* vector (1.1). For a fair comparison, for the baseline method, we also use *hsl_mc64* to permute the coefficient matrices to maximize the product of the diagonal entries and GRIP partitioning method to determine row-blocks.

4.1. Dataset. We use real $n \times n$ unsymmetric matrices with $n > 10{,}000$ and having at least one dense column that has more nonzeros than 1% of n from SuiteSparse Matrix Collection [16]. In the collection, there are 76 matrices which satisfy this condition, however, some of them have similar nonzero patterns. We observe that these similar matrices belong to the same problem groups and they often give similar results which may create a bias in the performance analysis. For instance, out of 76 matrices, there are 14, 11, and 8 matrices in the `TSOPF_RS`, `rajat`, and `barrier` problem groups, respectively. For this type of matrices, we use the largest matrix in the same problem group if the associated linear system converges in 10,000 iterations by using at least one of the methods. Otherwise, we use the next largest matrix in the same group and so on. Table 4.1 shows the properties of 20 matrices which satisfy these criteria. In the table, the matrices are given in decreasing sorted order by the ratio of the number of nonzeros in their densest column to n.

In the experiments, we partition the system into a number of row-blocks according to the size of $\mathcal{A}$. As seen in Table 4.1, we partition systems with $n < 160,000$ into 8 row blocks and larger systems into a number of row blocks where each block contains approximately 20,000 rows.

4.2. Experimental Framework. In the experiments, we use a shared-memory system and a distributed-memory system. We conduct extensive experiments with large number of matrices on the shared-memory system and limited experiments with a smaller number of matrices on the distributed-memory system due to limited core hours.

The shared-memory machine has four NUMA sockets each of which has an AMD Opteron 6376 processor running at 2.3GHz with 32GB memory. Since each processor has 16 cores, there are 64 cores in total. We use MPI implementation of OpenMPI v1.10.2 and gcc v4.7.2 compiler. We utilize BLAS and LAPACK implementations of Intel Math Kernel Library (MKL) v2019. Experiments are performed with 32 cores due to memory bandwidth limitations of the platform.

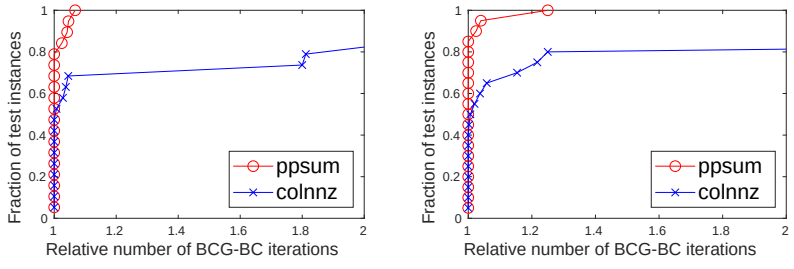
Each node of the distributed-memory system has two NUMA sockets each of which has a 14-core Intel Xeon E5-2680 processor running at 2.4Ghz with 64GB memory. Nodes are interconnected with a high-bandwidth low-latency switch network (56 Gbit/s Infiniband). We use MPI implementation of OpenMPI v1.10.0 and gcc v4.8.5 compiler. We utilize BLAS and LAPACK implementations of Intel MKL v2019.4. We use 8 distributed nodes and 16 cores of each node for performance analysis to study the details of the communication statistics and parallel running time of the steps of the proposed scheme.

In both parallel CG-BC and BCG-BC, mapping of row-blocks to processors is performed in the same way as in the ABCD solver [51]. If there are equal number of row-blocks and processors, then each row-block is assigned to a processor. If there are fewer row-blocks than processors, multiple processors may work on the same row-block. The decision of how many processors are assigned to a row-block is done according to the FLOP count of the analysis phase of MUMPS. If the computation on a row-block requires relatively more FLOPs than the others, more processors could be assigned to that row-block. If there are more row-blocks than processors, the row-blocks are distributed among processors while maintaining the load balance among the processors in terms of the sizes of the row-blocks.

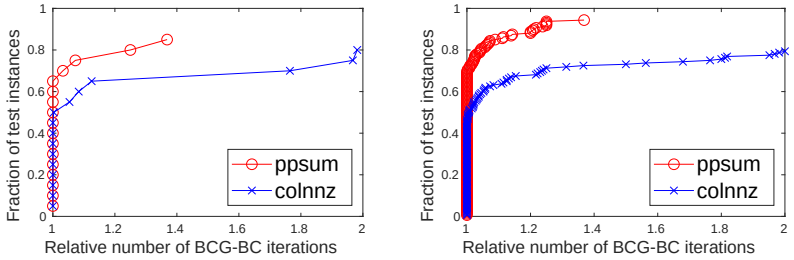
We use the same stopping criterion in [19, 51] for the algorithms i.e., backward error $\frac{\|\mathcal{A}x^{(j)}-f\|_\infty}{\|\mathcal{A}\|_\infty\|x^{(j)}\|_1+\|f\|_\infty} < 10^{-12}$. We set the maximum number of iterations as 10,000. We use the right-hand side vector that is provided with the matrix from the original problem in the SuiteSparse Matrix Collection. For some matrices, right–hand–side vectors are not provided. For those matrices, we use randomly generated right–hand–side vectors.

4.3. Experiments on the shared-memory system. Figure 4.1 compares two column selection metrics by using performance profiles [18] on the 20-matrix dataset using the number of BCG-BC iterations for the convergence as the comparison metric. In each figure, two performance profile curves compare two column selection metrics relative to the best performing one for each data instance. A point (x, y) on a performance profile curve denotes that the respective column selection metric requires at most x times more iterations than the best performing metric in y percent of the instances.

Figures 4.1a, 4.1b, 4.1c and 4.1d respectively display the performance profiles of



(a) $s = 1$ (b) $s = 5$



(c) $s = 15$ (d) $s = 1, 2, 3, 4, 5, 10, 15, 20$

FIG. 4.1. *Performance profiles of the comparison of colnnz and ppsum for the required BCG-BC iterations relative to the best.*

$s=1$, $s=5$, $s=15$, and for all s values ($s=1,2,3,4,5,10,15,20$) up to two times the iteration count of the best. As seen in Figure 4.1a, *ppsum* attains the best convergence rate in 80% (16 out of 20) of the test instances for $s=1$. In all of the test instances, *ppsum* leads to convergence within 1.1x iterations of the best metric for $s=1$. For $s=5$ (Figure 4.1b), *ppsum* attains the best convergence rate in 85% (17 out of 20) of the test instances. However, for $s=15$, the success rate of both methods decrease slightly due to numerical instability during the BCG-BC for `ohne2` and `torso1` test instances. Figure 4.1d shows the overall performance comparison for all s values. As seen in Figure 4.1d, *ppsum* achieves the best convergence rate in approximately 70% of all test instances. Therefore, we decide to use *ppsum* metric for selecting columns in the rest of the experiments.

Table 4.2 compares the proposed scheme against the baseline CG-BC algorithm in terms of the number of iterations required for convergence and the parallel solution time for different s values. In the proposed scheme, the number of iterations refers to the number of BCG-BC iterations and the timings include all of the solution steps (lines 12-15 as well as line 11 in Algorithm 3.2). The GRIP method, which is used in both proposed and baseline CG-BC algorithms, achieves row-block partitioning via using the well-known multilevel graph partitioning tool METIS [34]. As METIS involves randomized algorithms during the coarsening phase, for each matrix, five row-block partitions are obtained by using different seeds. The geometric means of the resulting iteration counts and running time are reported in Table 4.2. For each matrix, the best parallel solution time is shown in bold.

As seen in Table 4.2, the proposed algorithm achieves fewer iterations with increasing s in almost all matrices. There are two reasons for this. First, with the increasing s, the eigenvalues of H are expected to be clustered better around one due to the removal of selected columns. We note that those columns have the potential

TABLE 4.2
The number of iterations and parallel solution time in seconds.

Matrix		Baseline CG-BC	Proposed scheme for different s s=1	s=2	s=3	s=4	s=5	s=10	s=15	s=20
dc1	itr.	113	53	36	35	30	26	26	26	23
	time	72.5	20.3	15.4	16.5	15.7	**14.1**	21.2	24.1	28.8
trans4	itr.	16	11	6	6	6	4	4	4	4
	time	10.3	4.1	2.8	2.9	3.3	**2.5**	3.6	4.5	5.7
ASIC_100k	itr.	52	19	19	18	19	18	17	16	16
	time	27.5	**11.5**	13.2	13.3	14.4	14.9	18.8	22.4	26.1
mult_dcop_02	itr.	11	5	5	5	5	5	5	5	4
	time	1.4	**0.9**	0.9	1.0	1.1	1.1	1.3	1.5	2.0
rajat30	itr.	40	17	16	9	9	9	9	8	8
	time	125.4	54.6	26.3	**13.6**	15.2	17.9	29.3	33.3	52.4
shermanACb	itr.	1913	123	80	70	63	61	19	17	16
	time	101.3	8.5	5.5	5.2	5.1	4.9	2.1	2.6	2.8
TSOPF_RS_b39_c30	itr.	521	275	208	123	110	86	44	24	16
	time	12.8	10.1	10.3	8.4	9.3	8.9	8.6	6.7	**6.0**
coupled	itr.	114	91	76	61	55	52	41	32	28
	time	2.3	2.2	2.0	1.7	1.7	**1.6**	1.8	1.9	2.2
nxp1	itr.	8450	4172	2775	1939	1543	1288	688	stab.	stab.
	time	5201.9	2338.7	1975.6	1602.3	1441.6	1421.8	**1421.1**		
circuit_4	itr.	NC	NC	NC	8075	2988	1983	51	28	19
	time				1665.1	700.5	519.9	16.6	11.4	**10.9**
para-4	itr.	2236	845	449	342	51	38	33	31	32
	time	372.4	189.1	131.2	124.0	22.8	**20.2**	28.8	30.6	42.1
appu	itr.	422	405	404	396	394	388	378	369	363
	time	**43.7**	55.0	66.3	65.6	77.6	82.9	106.8	113.6	149.1
ohne2	itr.	5114	1965	1181	740	578	505	269	stab.	stab.
	time	1212.4	707.2	573.6	435.8	429.7	422.5	**369.8**		
av41092	itr.	553	332	249	202	180	151	101	80	stab.
	time	20.2	16.3	15.7	**15.0**	15.9	15.7	18.0	19.9	
ns3Da	itr.	161	143	133	121	113	105	84	74	68
	time	**3.0**	3.5	3.9	4.7	5.2	5.6	7.5	9.0	11.2
ted_A	itr.	7374	828	460	303	265	212	101	55	41
	time	38.2	5.5	3.7	3.0	3.1	2.8	2.5	**2.0**	2.3
hcircuit	itr.	241	1	1	1	1	1	1	1	1
	time	24.4	**0.3**	0.3	0.4	0.4	0.6	0.7	1.0	1.3
barrier2-10	itr.	3831	276	264	199	164	135	77	56	47
	time	490.7	49.5	57.5	54.3	53.3	50.6	48.3	**40.5**	47.3
torso1	itr.	NC	5886	3732	2822	2106	1706	945	stab.	stab.
	time		1133.7	969.4	902.1	818.0	791.6	**741.7**		
std1_Jac3_db	itr.	NC	1	1	1	1	1	1	1	1
	time		**0.1**	0.1	0.1	0.1	0.1	0.2	0.2	0.3
Avg. impr.:	iter.	-	3.4x	4.5x	5.6x	6.8x	7.8x	10.7x	12.1x	13.8x
	time	-	2.8x	3.2x	3.5x	3.8x	3.7x	3.4x	3.1x	2.8x

itr.: number of CG-BC and BCG-BC iterations, time: parallel solution time, Avg. impr.: Average (geometric mean) speedup obtained by the proposed algorithm against CG-BC in terms of parallel solution time (excluding the matrices where either method failed to converge), NC: does not converge in 10,000 iterations, stab.: BCG-BC fails due to numerical instability.

of deteriorating the orthogonality among subspaces. This results in a matrix with a better condition number thus leading to fewer iterations for the BCG-BC algorithm. Second, the block CG method used in the proposed scheme identifies multiple eigenvalues [7, 36, 37] and thus improves the detection of clusters of eigenvalues which leads to fewer iterations. However, there is also a trade-off between increasing s and parallel solution time since increasing s also increases computational cost per iteration while it decreases the number of iterations. In Table 4.2, the reduction in the number of iterations reflects to a decrease in the total parallel solution time for 18 out of 20 test problems. Only in `appu` and `ns3Da`, the reduction in the number of BCG-BC iterations does not lead to faster parallel solution time.

The last two rows of Table 4.2 respectively show the average speedup obtained by the proposed algorithm (for different s values) against CG-BC in terms of the number

of iterations and parallel solution time for each s. The average speedup values obtained by the proposed algorithm for a given s are computed as the average of the ratios of the number of iterations and parallel solution time attained by the proposed algorithm over CG-BC. In computing average speedup values, we prefer using geometric mean rather than arithmetic mean to smooth out the very large speedup values. A notable example is `hcircuit` for which large speedup values are obtained since the proposed scheme requires only one BCG-BC iteration for all s, whereas the CG-BC requires 241 iterations, respectively. The geometric mean values of the number of iterations displayed in the table vary between 3.4 and 13.8, the arithmetic mean values vary between 17.9 and 53.8. Similarly, the geometric mean values of parallel solution time vary between 2.8 and 3.8, and the arithmetic mean values vary between 7.4 and 8.9. As seen in the table, the average performance improvement of the proposed algorithm against CG-BC in terms of the number of iterations increases with increasing s up to 13.8 times fewer iterations. On the other hand, the average speedup of the proposed algorithm against CG-BC in terms of parallel solution time initially increases with increasing s peaking at 3.8x for $s = 4$, then it gradually decreases.

As also seen in Table 4.2, in 18 out of 20 test matrices the proposed scheme obtains a faster parallel solution time for $s = 1, 2, 3, 4, 5, 10$. For $s = 15$ and 20, the proposed scheme obtains a better solution time in 13 out of 20 test matrices. This relative performance degradation of the proposed scheme is mainly due to numerical instability during BCG-BC for large block-sizes. Here and hereafter, block-size refers to the number of rhs vector. In [7, 50], no stability issues are reported for smaller than 32 rhs vectors for their dataset. However, because of our selection criterion the matrices we have included in our dataset such as `nxp1`, `ohne2`, `torso1`, and `av41092` are more challenging. Hence, they cause stability issues even when the number of rhs vectors smaller than 32.

Results of some extensive experiments using BCG-BC with increasing block-sizes for various problems are given in [50]. It is stated there, for some problems, using larger block-size leads to an improvement in the total solution time, but for some problems such as `torso3`, there is no improvement in the total solution time. Even though `torso3` is not included in our dataset due to our selection criterion, we have performed additional experiments using this matrix to verify the effectiveness of the proposed scheme. These experiments indicate that the proposed scheme achieves improvements in parallel solution time with increasing s. This is due to the main contribution of the proposed scheme which aims at attaining better eigenvalue cluster in H by handling some columns and respective rows separately via forming the Schur complement system.

4.4. Experiments on the distributed-memory system. In this section, we show the performance of the proposed scheme in parallel factorization and parallel solution stages. We then show the robustness of the proposed scheme through experiments conducted on the distributed-memory system. For having sufficiently large granularity on the target distributed-memory system, we consider the largest matrices from Table 4.1 with $n > 100{,}000$. Among the ten large matrices satisfying this criterion we selected five matrices having more number of nonzeros than 10% of n in their densest column. The matrices that satisfy these two criteria are `dc1`, `trans4`, `ASIC_100k`, `rajat30` and `nxp1`. To see the effect of the communication costs clearly we set the number of row blocks to 128 for 128 MPI processes so that each row-block is assigned to a distinct processor.

4.4.1. Parallel factorization. We compare the proposed scheme against the baseline algorithm in terms of factorization time of the augmented systems in (2.4). Table 4.3 shows the maximum factorization time among 128 processors in seconds using MUMPS. Here, we run 128 embarrassingly parallel MUMPS instances each running sequentially. In general, the proposed scheme achieves less time in parallel factorization. This experimental finding is expected since handling the "dense" columns and respective rows separately via forming the Schur complement system in the proposed scheme is likely to incur less fill-in as well as better load balance. For instance, in `rajat30`, the proposed scheme decreases the number of nonzeros in the factors (including the fill-in) of the augmented systems of the most heavily loaded processor from 1,145,772 to 460,683. This contributes to the decrease in the factorization time from 4.93 seconds of the baseline algorithm to 0.29 seconds of the proposed scheme for $s = 20$. On average, the proposed scheme achieves 6.7 times faster factorization time than the baseline algorithm for $s = 20$.

TABLE 4.3
Parallel factorization time in seconds

Matrix	Baseline	Proposed scheme for different s							
		1	2	3	4	5	10	15	20
dc1	0.62	0.56	0.26	0.26	0.26	0.27	0.26	0.27	0.27
trans4	1.02	0.21	0.25	0.22	0.23	0.24	0.23	0.27	0.26
ASIC_100k	0.54	0.58	0.54	0.59	0.57	0.61	0.59	0.58	0.52
rajat30	4.93	3.21	0.66	0.40	0.31	0.36	0.34	0.35	0.29
nxp1	0.64	0.50	0.24	0.27	0.26	0.07	0.06	0.07	0.07
Avg. Impr.	-	1.9x	3.6x	4.5x	5.2x	6.0x	6.4x	6.2x	6.7x

4.4.2. Parallel solution. We compare the proposed scheme against the baseline algorithm in terms of iterative solution stage using BCG-BC against CG-BC in the baseline algorithm. Table 4.4 illustrates this comparison in terms of the number of iterations required for convergence, per-iteration and total parallel solution times as well as per-iteration communication statistics. The per-iteration time is the average time per-iteration including communication and computation, which is obtained through dividing the total parallel solution time of BCG-BC by the number of iterations for convergence in BCG-BC. For communication statistics, we use two metrics for measuring the communication requirements of each iteration; the average number of messages and the average message volume sent by a processor. The former and latter metrics respectively refer to the latency and bandwidth overheads. The message volume is given in terms of the number of floating point words (divided by 1,000) transmitted between processors.

Since the proposed scheme removes "dense" columns, it has the potential of decreasing the latency overhead through reducing the number of messages. It also has the potential of decreasing bandwidth overhead per rhs vector. However, since the number of rhs vectors in BCG-BC is s+1, the message volume increases by a factor of s+1. Thus, although the proposed scheme has the potential of decreasing bandwidth overhead for small s, it might increase the bandwidth overhead for large s.

As seen in Table 4.4, the proposed scheme significantly reduces the number of messages in all matrices except `ASIC_100k` for which there is no improvement. This performance gap between the proposed and baseline methods increases in general with increasing s in the other four matrices. The proposed scheme achieves smaller message volume for small s values in `dc1` (for $s \leq 2$) and `trans4` (for $s \leq 2$) and

TABLE 4.4
Per-iteration communication statistics and parallel running time details of CG-BC and BCG-BC on the distributed-memory system.

Matrix		Baseline	Proposed Scheme for varying s							
	Metric	CG-BC	s=1	s=2	s=3	s=4	s=5	s=10	s=15	s=20
dc1	Avg msg cnt	127	63	59	61	62	61	61	**59**	60
	Avg vol (x10^3)	4.15	**2.56**	3.81	5.04	6.27	7.51	13.62	19.31	25.13
	Per-iter time	0.13	**0.09**	0.10	0.11	0.12	0.13	0.18	0.23	0.31
	# of iters	118	66	51	44	41	39	40	38	**37**
	Total time	15.82	6.32	5.53	5.45	**5.40**	5.54	8.22	10.10	13.40
trans4	Avg msg cnt	127	58	60	59	58	57	57	59	**56**
	Avg vol (x10^3)	4.08	**2.46**	3.73	4.95	6.02	7.29	13.15	18.75	24.12
	Per-iter time	0.12	**0.08**	0.09	0.10	0.10	0.11	0.15	0.20	0.26
	# of iters	27	18	11	8	6	4	**4**	5	6
	Total time	3.21	1.64	1.23	1.01	0.90	**0.74**	1.07	1.68	2.41
ASIC_100k	Avg msg cnt	127	127	127	127	127	127	127	127	127
	Avg vol (x10^3)	**6.27**	12.47	18.67	24.89	31.14	37.28	67.77	97.87	127.82
	Per-iter time	**0.14**	0.16	0.19	0.21	0.22	0.23	0.29	0.37	0.41
	# of iters	87	28	26	26	26	25	24	24	**23**
	Total time	11.79	**4.62**	5.12	5.82	5.99	6.04	7.50	9.75	10.46
rajat30	Avg msg cnt	115	114	91	30	29	27	23	23	**23**
	Avg vol (x10^3)	28.86	27.27	14.02	**12.27**	16.05	17.32	35.73	48.03	63.08
	Per-iter time	0.66	0.79	0.39	**0.33**	0.38	0.52	0.95	1.31	1.74
	# of iters	53	28	20	12	12	10	8	8	**8**
	Total time	35.24	22.77	8.98	**4.89**	5.72	6.59	10.08	13.42	18.38
nxp1	Avg msg cnt	30	**23**	23	23	23	24	23	23	23
	Avg vol (x10^3)	**2.35**	2.86	4.30	5.73	7.30	8.69	15.72	23.25	29.43
	Per-iter time	**0.08**	0.09	0.10	0.11	0.13	0.15	0.27	0.80	0.90
	# of iters	NC	NC	9439	6921	5553	4369	**2440**	stab.	stab.
	Total time			907.33	770.58	742.11	**675.33**	681.46		

Avg msg cnt/vol: average number of messages/volume sent by a processor, Per-iter/Total time: parallel per-iteration/total solution time in seconds, # of iters: number of iterations, NC: does not converge in 10,000 iterations, stab.: BCG-BC fails due to numerical instability.

`rajat30` (for $s \leq 5$). For small s values ($s \leq 5$), the per-iteration parallel running time of the proposed scheme remains comparable with those of CG-BC. Therefore, the significant amount of decrease achieved by the proposed scheme in terms of the number of iterations required for convergence leads to significantly faster parallel solution time.

We also provide a matrix-specific analysis on the correlation between per-iteration communication statistics and per-iteration running time. We first consider `dc1` matrix. For $s \leq 4$, the proposed scheme yields faster parallel per-iteration time than CG-BC although the proposed scheme involves more computational work due to multiple rhs vectors. This improvement in the parallel per-iteration time mainly comes from the decrease in communication overhead. For $s=1$ and 2, the proposed scheme achieves faster per-iteration time because of less overhead incurred in both latency and bandwidth metrics compared to CG-BC. For $s=3$ and 4, although the message volume is higher than that of CG-BC, the proposed scheme achieves faster time per iteration because of less latency overhead incurred due to the average message count values of 61 and 62 for $s = 3$ and $s = 4$, respectively, instead of 127 in CG-BC. For larger s values ($s = 5, 10, 15, 20$), since the increased average message volume and doing more work in an iteration dominate the improvement from the latency overhead, the parallel per-iteration time in the proposed scheme becomes slower than that of CG-BC. A similar correlation can also be observed for `trans4` and `rajat30` which

leads the proposed scheme to achieve faster per-iteration time for small s ($s \leq 5$ for `trans4` and $2 \leq s \leq 5$ for `rajat30`).

In both `nxp1` and `ASIC_100k`, the proposed scheme cannot reduce the per-iteration time. In `nxp1`, this is because of the increased bandwidth overhead with increasing s despite the decrease in the latency overhead. In `ASIC_100k`, increased bandwidth and latency overheads fails to improve per-iteration time. We should mention that the proposed scheme decreases the total solution time in all matrices including these two matrices since it achieves the significant decrease in the number of iterations.

We present Figure 4.2 to illustrate the performance of the proposed scheme in terms of the number of iterations required for convergence and parallel solution time normalized with respect to those of CG-BC. In the figure, we do not present a bar chart for `nxp1` because the baseline algorithm cannot converge in the maximum number of iterations allowed. As seen in the figure, the relative performance of the proposed scheme in terms of convergence rate increases with increasing s in general. However, the relative performance of the proposed scheme in terms of the parallel solution time starts to deteriorate at $s = 4$, $s = 5$, $s = 1$, and $s = 3$, for `dc1`, `trans4`, `ASIC_100k`, and `rajat30`, respectively. This is because of the increase in the per-iteration parallel running time of the proposed scheme with increasing s.

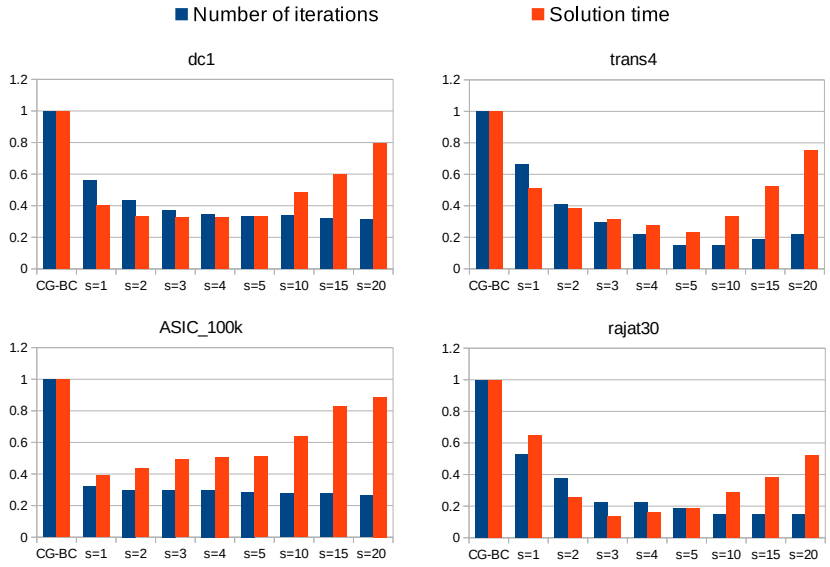


FIG. 4.2. *Number of iterations required for convergence and parallel solution time of the proposed scheme normalized with respect to those of CG-BC.*

4.4.3. Robustness. Another advantage of the block Cimmino method is its robustness compared to classical preconditioned iterative methods [13]. In [13], CG-BC is compared with sequential preconditioned iterative solvers. As seen in Table 4.2, the proposed scheme inherits the robustness of CG-BC. On the other hand, as also shown in the table, s value affects the robustness of the proposed scheme. Yet, for the given test problems with $s = 3, 4, 5$, and 10, the proposed scheme does not fail. We propose a "default" s value of 4 which gives slightly better parallel runtime performance than others on average.

In Table 4.5, we report the results of the experiments performed for comparing

the robustness of the proposed scheme against PETSc's [9] parallel implementation of GMRES [38] and Bi-CGStab [47] methods with state-of-the-art preconditioners using 128 MPI processes on the distributed-memory system. Three parallel preconditioners are adopted with each method; two are block Jacobi preconditioners of PETSc with different levels of fill-in, the other is parallel algebraic multigrid method (BoomerAMG) [49] of the *hypre* library [22]. The BoomerAMG preconditioner is called from PETSc. We employ the default PETSc parameters for GMRES and Bi-CGStab as well as default parameters for the preconditioners. Some of those default parameters are; restart value for GMRES is 30 and the maximum number of iterations is 10,000.

TABLE 4.5
Experiments with parallel preconditioned iterative solvers

		Proposed	GMRES(30)			Bi-CGStab		
Matrix		$s=4$	BJ-ILU(0)	BJ-ILU(1)	BoomerAMG	BJ-ILU(0)	BJ-ILU(1)	BoomerAMG
dc1	itr.	41	268	156	F_1	F_1	F_1	F_2
	time	5.40	0.49	0.55				
trans4	itr.	6	119	124	4	202	101	3
	time	0.90	0.38	0.52	3.54	0.53	0.60	4.69
ASIC_100k	itr.	26	F_3	F_3	F_1	F_3	F_3	F_2
	time	5.99						
rajat30	itr.	12	F_3	F_3	F_1	F_3	F_3	F_2
	time	5.72						
nxp1	itr.	5553	F_3	F_3	NC	F_3	F_3	F_2
	time	742.11						

BJ-ILU(x): Block Jacobi preconditioner each block handled via ILU(x), NC: does not converge in 10,000 iterations, F_1: breakdown, F_2: divergence, F_3: unstable preconditioner, itr.: number of iterations for convergence, time: parallel solution time in seconds.

In CG-BC, the normwise backward error of less than 10^{-12} is used as the stopping criterion. In PETSc, iterations are stopped on the basis of relative convergence tolerance (*rtol*) value. For a fair comparison, we use matrix specific *rtol* values in PETSc to obtain a comparable normwise backward errors. Thus, we set *rtol* to 10^{-7}, 10^{-6}, 10^{-1}, 10^{-1} and 10^{-2} for systems with `dc1`, `trans4`, `ASIC_100k`, `rajat30`, and `nxp1` matrices, respectively.

With the block Jacobi preconditioner the number of blocks is set to be equal to the number of MPI processes and each block is handled by the incomplete LU factorization (ILU). We use the default level of fill-in (ILU(0)) and as well as allowing more fill-in by using ILU(1) which will be respectively referred to as BJ-ILU(0) and BJ-ILU(1) in Table 4.5. In the experiments, sub_pc_factor_nonzeros_along_diagonal parameter is enabled to reorder the blocks before factorization to remove zeros from diagonal if possible.

Table 4.5 shows the results of parallel PETSc experiments with two iterative solvers, each with three preconditioners. In addition to PETSc results, the table includes the results of the proposed scheme with default s value of 4. In the table, different modes of failures are indicated by NC, F_1, F_2, and F_3 which respectively denote nonconvergence due to reaching maximum number of iterations, breakdown of the method, divergence due to residual norm increased by a factor of 10^5, and unstable preconditioner.

As seen in Table 4.5, out of five cases, GMRES(30) with BJ-ILU(0) and BJ-ILU(1) preconditioners converge in two cases and GMRES(30) with BoomerAMG preconditioner converges in only one case, whereas Bi-CGStab with BJ-ILU(0), BJ-

ILU(1) and BoomerAMG preconditioners converge in only one case. On the other hand, the proposed method converges in all five cases thus reconfirming its robustness.

5. Conclusion. In this study, we propose a novel scheme which enhances the block Cimmino algorithm via handling "dense" columns separately by forming the Schur complement system. Extensive experiments on a wide range of matrices lead to the following findings. For selecting "dense" columns, the proposed metric that considers the values of the nonzeros in the columns outperforms the metric that considers only the number of nonzeros in terms of the required number of iterations for the convergence. On average, the proposed scheme achieves 13.8 times fewer iterations and 3.8 times faster parallel solution time compared to the classical CG accelerated block Cimmino algorithm on the test matrices. Furthermore, the proposed scheme also reduces the communication requirements of the parallel block Cimmino which leads to faster per-iteration time in the parallel block Cimmino. Performance of the proposed scheme may degrade if the number of selected columns is not chosen carefully. This is because increasing the number of selected columns increases per-iteration computational cost as well as communication volume.

Acknowledgments. Computing resources used in this work were provided by the National Center for High Performance Computing of Turkey (UHeM) under grant number 1008172020.

REFERENCES

[1] M. Adlers and Å. Björck, *Matrix stretching for sparse least squares problems*, Numerical linear algebra with applications, 7 (2000), pp. 51–65.
[2] P. R. Amestoy, I. Duff, J. Koster, and J.-Y. L'Excellent, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM Journal on Matrix Analysis and Applications, 23 (2001), pp. 15–41.
[3] K. D. Andersen, *A modified Schur-complement method for handling dense columns in interior-point methods for linear programming*, ACM Transactions on Mathematical Software (TOMS), 22 (1996), pp. 348–356.
[4] R. Ansorge, *Connections between the Cimmino-method and the Kaczmarz-method for the solution of singular and regular systems of equations*, Computing, 33 (1984), pp. 367–375.
[5] M. Arioli, I. Duff, and P. P. de Rijk, *On the augmented system approach to sparse least-squares problems*, Numerische Mathematik, 55 (1989), pp. 667–684.
[6] M. Arioli, I. Duff, J. Noailles, and D. Ruiz, *A block projection method for sparse matrices*, SIAM Journal on Scientific and Statistical Computing, 13 (1992), pp. 47–70.
[7] M. Arioli, I. Duff, D. Ruiz, and M. Sadkane, *Block Lanczos techniques for accelerating the block Cimmino method*, SIAM Journal on Scientific Computing, 16 (1995), pp. 1478–1511.
[8] H. Avron, E. Ng, and S. Toledo, *Using perturbed QR factorizations to solve linear least-squares problems*, SIAM Journal on Matrix Analysis and Applications, 31 (2009), pp. 674–693.
[9] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, *Efficient management of parallelism in object oriented numerical software libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, eds., Birkhäuser Press, 1997, pp. 163–202.
[10] M. Benzi, *Preconditioning techniques for large linear systems: a survey*, Journal of computational Physics, 182 (2002), pp. 418–477.
[11] Å. Björck, *Numerical methods for least squares problems*, SIAM, 1996.
[12] A. Björck, *Solving linear least squares problems by Gram-Schmidt orthogonalization*, BIT Numerical Mathematics, 7 (1967), pp. 1–21.
[13] R. Bramley and A. Sameh, *Row projection methods for large nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 13 (1992), pp. 168–193.
[14] C. G. Broyden, *A breakdown of the block CG method*, Optimization Methods and Software, 7 (1996), pp. 41–55.
[15] G. Cimmino and C. N. delle Ricerche, *Calcolo approssimato per le soluzioni dei sistemi di equazioni lineari*, Istituto per le applicazioni del calcolo, 1938.

[16] T. A. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1:1–1:25.
[17] J. W. Demmel and N. J. Higham, *Improved error bounds for underdetermined system solvers*, SIAM Journal on Matrix Analysis and Applications, 14 (1993), pp. 1–14.
[18] E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical programming, 91 (2002), pp. 201–213.
[19] L. Drummond, I. Duff, R. Guivarch, D. Ruiz, and M. Zenadi, *Partitioning strategies for the block Cimmino algorithm*, Journal of Engineering Mathematics, 93 (2015), pp. 21–39.
[20] I. Duff, R. Guivarch, D. Ruiz, and M. Zenadi, *The augmented block Cimmino distributed method*, SIAM Journal on Scientific Computing, 37 (2015), pp. A1248–A1269.
[21] I. S. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM Journal on Matrix Analysis and Applications, 22 (2001), pp. 973–996.
[22] R. D. Falgout and U. M. Yang, *hypre: A library of high performance preconditioners*, in International Conference on computational science, Springer, 2002, pp. 632–641.
[23] A. George and M. T. Heath, *Solution of sparse linear least squares problems using Givens rotations*, Linear Algebra and its applications, 34 (1980), pp. 69–83.
[24] P. Gilbert, *Iterative methods for the three-dimensional reconstruction of an object from projections*, Journal of theoretical biology, 36 (1972), pp. 105–117.
[25] P. E. Gill, W. Murray, D. B. Ponceleon, and M. A. Saunders, *Solving reduced KKT systems in barrier methods for linear and quadratic programming*, tech. report, SOL 91-7, Department of Operations Research, Stanford University, Stanford, CA, 1991.
[26] L. Giraud, A. Haidar, and Y. Saad, *Sparse approximations of the Schur complement for parallel algebraic hybrid solvers in 3D*, Numerical Mathematics, 3 (2010), pp. 276–294.
[27] D. Goldfarb and K. Scheinberg, *A product-form Cholesky factorization method for handling dense columns in interior point methods for linear programming*, Mathematical Programming, 99 (2004), pp. 1–34.
[28] G. H. Golub and M. A. Saunders, *Linear least squares and quadratic programming*, tech. report, Stanford University, Stanford, CA, USA, 1969.
[29] G. H. Golub and C. F. Van Loan, *Matrix computations*, John Hopkins University Press, 1983, p. 530.
[30] J. Gondzio, *Splitting dense columns of constraint matrix in interior point methods for large scale linear programming*, Optimization, 24 (1992), pp. 285–297.
[31] HSL, *A collection of Fortran codes for large scale scientific computation*, See http://www.hsl.rl.ac.uk, (2007).
[32] H. Ji and Y. Li, *A breakdown-free block conjugate gradient method*, BIT Numerical Mathematics, 57 (2017), pp. 379–403.
[33] S. Kaczmarz, *Angenäherte auflösung von systemen linearer gleichungen*, Bulletin International de l'Academie Polonaise des Sciences et des Lettres, 35 (1937), pp. 355–357.
[34] G. Karypis and V. Kumar, *Metis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 5.1*. http://www.cs.umn.edu/~metis, 2013.
[35] C. Mészáros, *Detecting "dense" columns in interior point methods for linear programs*, Computational Optimization and Applications, 36 (2007), pp. 309–320.
[36] D. P. O'Leary, *The block conjugate gradient algorithm and related methods*, Linear algebra and its applications, 29 (1980), pp. 293–322.
[37] D. Ruiz, *Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment*, CERFACS TH/PA/9, 6 (1992).
[38] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on scientific and statistical computing, 7 (1986), pp. 856–869.
[39] M. A. Saunders, *Cholesky-based methods for sparse least squares: The benefits of regularization*, in Adams L, Nazareth JL (eds) Linear and nonlinear conjugate gradient-related methods, SIAM Philadelphia, PA, 1996, pp. 92–100.
[40] J. Scott and M. Tuma, *Solving mixed sparse-dense linear least-squares problems by preconditioned iterative methods*, SIAM Journal on Scientific Computing, 39 (2017), pp. A2422–A2437.
[41] J. Scott and M. Tuma, *A Schur complement approach to preconditioning sparse linear least-squares problems with some dense rows*, Numerical Algorithms, 79 (2018), pp. 1147–1168.
[42] J. A. Scott and M. Tuma, *Sparse stretching for solving sparse-dense linear least-squares problems*, SIAM Journal on Scientific Computing, 41 (2019), pp. A1604–A1625.
[43] F. Sloboda, *A projection method of the Cimmino type for linear algebraic systems*, Parallel computing, 17 (1991), pp. 435–442.
[44] C. Sun, *Dealing with dense rows in the solution of sparse linear least squares problems*, tech. re-

port, Advanced Computing Research Institute, Cornell Theory Center, Cornell University, 1995.
[45] C. Sun, *Parallel solution of sparse linear least squares problems on distributed-memory multiprocessors*, Parallel computing, 23 (1997), pp. 2075–2093.
[46] F. S. Torun, M. Manguoglu, and C. Aykanat, *A novel partitioning method for accelerating the block Cimmino algorithm*, SIAM Journal on Scientific Computing, 40 (2018), pp. C827–C850.
[47] H. A. van der Vorst, *Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems*, SIAM Journal on Scientific and Statistical Computing, 13 (1992), pp. 631–644.
[48] R. J. Vanderbei, *Splitting dense columns in sparse linear systems*, Linear Algebra and its Applications, 152 (1991), pp. 107–117.
[49] U. M. Yang et al., *BoomerAMG: A parallel algebraic multigrid solver and preconditioner*, Applied Numerical Mathematics, 41 (2002), pp. 155–177.
[50] M. Zenadi, *Méthodes hybrides pour la résolution de grands systèmes linéaires creux sur calculateurs parallèles*, PhD thesis, École Doctorale Mathématiques, Informatique et Télécommunications (Toulouse); 142547247, 2013.
[51] M. Zenadi, D. Ruiz, and R. Guivarch, *The Augmented Block Cimmino Distributed Solver*. http://abcd.enseeiht.fr/, 2021. ABCD Solver v1.0.