SIAM J. Sci. Comput. © 2022 Society for Industrial and Applied Mathematics
Vol. 44, No. 2, pp. C99–C123

PARTITIONING AND REORDERING FOR SPIKE-BASED DISTRIBUTED-MEMORY PARALLEL GAUSS–SEIDEL[*]

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

Abstract. Gauss–Seidel (GS) is a widely used iterative method for solving sparse linear systems of equations and also known to be effective as a smoother in algebraic multigrid methods. Parallelization of GS is a challenging task since solving the sparse lower triangular system in GS constitutes a sequential bottleneck at each iteration. We propose a distributed-memory parallel GS (dmpGS) by implementing a parallel sparse triangular solver (stSpike) based on the Spike algorithm. stSpike decouples the global triangular system into smaller systems that can be solved concurrently and requires the solution of a much smaller reduced sparse lower triangular system which constitutes a sequential bottleneck. In order to alleviate this bottleneck and to reduce the communication overhead of dmpGS, we propose a partitioning and reordering model consisting of two phases. The first phase is a novel hypergraph partitioning model whose partitioning objective simultaneously encodes minimizing the reduced system size and the communication volume. The second phase is an in-block row reordering method for decreasing the nonzero count of the reduced system. Extensive experiments on a dataset consisting of 359 sparse linear systems verify the effectiveness of the proposed partitioning and reordering model in terms of reducing the communication and the sequential computational overheads. Parallel experiments on 12 large systems using up to 320 cores demonstrate that the proposed model significantly improves the scalability of dmpGS.

Key words. parallel Gauss–Seidel, distributed-memory, Spike algorithm, parallel sparse triangular solve, hypergraph partitioning, sparse matrix reordering

AMS subject classifications. 68W10, 05C65, 65F50, 65F10

DOI. 10.1137/21M1411603

1. Introduction.

A wide range of applications in science and engineering require the solution of a sparse linear system of equations

$$Ax = f, \tag{1.1}$$

where $A \in \mathbb{R}^{m\times m}$ is a general large sparse invertible matrix and x and $f \in \mathbb{R}^m$ are the unknown and right-hand-side vectors, respectively. Depending on the numerical and structural properties of the coefficient matrix, various solvers have been proposed.

Direct solvers require a sequence of operations: reordering and partitioning, symbolic factorization, numerical factorization, and finally obtaining the solution, typically via forward and backward sweeps. The reordering and partitioning schemes are used both to reduce the amount of fill-in and to enhance the parallel scalability. Symbolic factorization is used to determine the sparsity structure of the factors, and finally the numerical factorization (such as sparse LU [23], QR [34], SVD [13], and

[*]Submitted to the journal's Software and High-Performance Computing section April 12, 2021; accepted for publication (in revised form) October 7, 2021; published electronically March 14, 2022.
https://doi.org/10.1137/21M1411603

Funding: This work was supported by the National Center for High Performance Computing of Turkey (UHeM) under grant 4007772020. The work of the third author was supported by the BAGEP award of the Science Academy.

[†]Department of Computer Engineering, Bilkent University, 06800 Ankara, Turkey (tugba.uzluer@bilkent.edu.tr, aykanat@cs.bilkent.edu.tr).

[‡]Department of Computer Engineering, Ankara Yildirim Beyazit University, 06010 Ankara, Turkey (ftorun@ybu.edu.tr).

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

WZ [17]) is computed. Direct solvers are robust and, in general, are known to be very scalable during the factorization phase [5, 43] but not so much during the triangular solution phase [45].

Iterative solvers, on the other hand, are known to be more scalable but not as robust as direct solvers. Nevertheless, they are still preferred for large sparse systems due to their lower memory requirements. Starting with an initial guess for the solution vector, these methods improve the solution at each iteration. There are two main types of iterative solvers: stationary and nonstationary methods. Stationary methods have the general form $x^{(k+1)} = \phi(x^{(k)})$, where $x^{(k)}$ is the solution vector at the kth iteration and $\phi(\cdot)$ is a function which does not change during the iterations. Some examples are Jacobi, Gauss–Seidel, successive overrelaxation (SOR), and symmetric SOR [34, 58]. Nonstationary methods have the form $x^{(k+1)} = \phi^{(k)}(x^{(k)})$ in which the function $\phi^{(k)}(\cdot)$ changes at each iteration. Some examples are projection methods, Krylov subspace methods, and Chebyshev iterations [9, 35, 58].

In practice, linear systems are preconditioned to reduce the required number of iterations of the iterative solvers and to improve their robustness. There could be a variety of choices of preconditioners; some are problem specific and others are more general. General classical preconditioners include incomplete factorization–based preconditoners (such as incomplete LU [57, 58]), sparse approximate inverse [11], algebraic multigrid (AMG) [51, 56], and others. We refer the reader to [10] for a detailed survey of preconditioners. Among these preconditioners, AMG has been widely used recently in many applications [12, 30, 53] and is a generalization of geometric multigrid (GMG) [69]. GMG requires some knowledge of the physical problem and/or its geometry, while there is no such requirement for AMG. AMG can be also used as a direct solver [36, 70]. Furthermore, AMG typically uses another iterative method as a "smoother" which is required to reduce the error at each level, and the smoother itself can also be preconditioned. More recently a preferred smoother for AMG is Gauss–Seidel [3, 16, 66], as in BoomerAMG [36] and Trilinos-ML [32].

Gauss–Seidel (GS) is a well-known stationary iterative method which solves the linear system (1.1) by splitting the coefficient matrix into its lower and strictly upper triangular parts, $A = L + U$. Then the solution is obtained iteratively by

$$x^{(k+1)} = L^{-1}(f - Ux^{(k)}).$$

At each iteration of GS, both a lower triangular system is required to be solved and an upper triangular sparse matrix-vector multiplication (SpMV) is performed. GS is guaranteed to converge if A is strictly or irreducibly diagonal dominant [7] or symmetric positive definite [34]. It is known to be effective and preferred as a smoother for a wide variety of problems [3, 71]. However, a true distributed-memory parallelization of GS is considered to be a challenging task [3].

In the literature, parallel GS implementations are proposed either to solve the original problem (1.1) [6, 42, 61] or to use it as a smoother in multigrid schemes [38, 63, 72]. A commonly used method to parallelize GS by finding independent subtasks is the red-black coloring strategy [2, 31, 41], which has been extended to multicoloring [33, 52, 4] to attain more parallelism for complicated regular problems. However, multicolored GS is not feasible for some cases such as unstructured finite element applications since the number of colors becomes too large [42]. Another approach is to use a processor-localized GS in which each processor performs GS as a subdomain solver, but its convergence rate is low and may diverge for a large number of processors [3].

The main difficulty in parallelizing GS arises from the sequential nature of triangular solves included in GS [71]. Along with its importance in several applications, solving triangular systems often constitutes a sequential bottleneck because of the dependencies between unknowns in forward or backward substitutions. In [59], a parallel banded triangular solver is proposed. This algorithm is extended for solving banded linear systems [21, 28] and further improved by implementing various alternatives in each step of the factorization including the solution of the reduced system in [55, 54, 62]. At this point, the algorithm is called the Spike algorithm. For sparse linear systems, Spike is also proposed as a solver for a banded preconditioner that is sparse within the band [49, 60], and it is generalized for sparse linear systems [15, 47, 48]. In [68], a Spike-based parallel solver for general tridiagonal systems is implemented for graphics processing unit architectures. A recent study [22] proposes a multithreaded parallel solver for sparse triangular systems by extending the Spike algorithm [59].

We propose a distributed-memory parallel GS (dmpGS) by implementing and using a distributed-memory version of the sparse triangular Spike (stSpike) algorithm. stSpike enables obtaining the solution of the system by solving independent sparse triangular subsystems and a smaller reduced triangular system. Solving this reduced system constitutes a sequential computational bottleneck in dmpGS. The size of this reduced system is equal to the number of nonzero columns in the lower off-diagonal blocks of the coefficient matrix. The computational cost of solving the reduced system is proportional to its nonzero count. The communication volume of dmpGS is equal to the number of nonzero column segments in the off-diagonal blocks plus the reduced system size. Both of these communication and computational overheads highly depend on the sparsity structure of the coefficient matrix.

We note that solving the reduced system is embarrassingly parallel if the coefficient matrix is banded and diagonally dominant [50, 54]. In case the coefficient matrix is not diagonally dominant, another way to alleviate the cost of solving the reduced system is to further parallelize the solution of the reduced system, which has been done iteratively [55] or recursively [15, 54]. Instead, we propose to minimize the size and the nonzero count of the reduced system, together with the communication volume, and show that the resulting reduced system is so small that further parallelization of the solution of the reduced system is often no longer needed. For attaining these minimization objectives, we propose a partitioning and reordering model that exploits the sparsity of the coefficient matrix. The proposed model consists of two phases. The first phase is a row-wise partitioning of the coefficient matrix, whereas the second phase is a row reordering within the row blocks induced by the partition obtained in the first phase.

For the first phase, we propose a novel hypergraph model that extends and enhances the conventional column-net model for simultaneously decreasing the reduced system size and the communication volume. We introduce vertex fixing, net anchoring, and net splitting schemes within the recursive bipartitioning framework to encode the minimization of the number of nonzero column segments in the lower triangular part of the resulting partition.

For the second phase, we propose an intelligent in-block row reordering method with the aim of decreasing the computational costs of both forming the coefficient matrix of the reduced system once and solving the reduced system at each iteration.

The rest of the paper is organized as follows. Section 2 provides the background information on hypergraph and sparse matrix partitioning and stSpike. In section 3, we discuss the dmpGS algorithm along with its communication and computational costs. The proposed partitioning and reordering model for dmpGS is introduced in section 4. We provide the experimental results in section 5 and conclude in section 6.

2. Background.

2.1. Hypergraph partitioning. A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ consists of a set of vertices $\mathcal{V}=\{v_i\}_{1\leq i\leq n}$ and a set of nets $\mathcal{N}=\{n_j\}_{1\leq j\leq m}$. Each net $n_j\in\mathcal{N}$ connects a subset of vertices in $\mathcal{V}$, which is referred to as the *pins* of n_j and denoted by $Pins(n_j,\mathcal{H})$. Each vertex v_i is assigned a weight $w(v_i)$, and each net n_j is assigned a cost $cost(n_j)$. $\Pi=\{\mathcal{V}_1,\mathcal{V}_2,\dots,\mathcal{V}_k\}$ denotes a K-way partition of $\mathcal{H}$, where parts are mutually disjoint and exhaustive. The weight of a part is the sum of the weights of vertices in that part. For a given partition, if a net connects at least one vertex in a part, it is said to *connect* that part. *Connectivity* $\lambda(n_j)$ of net n_j is the number of parts connected by n_j. If a net n_j connects multiple parts (i.e., $\lambda(n_j)>1$), it is called *cut*; and otherwise *internal* (i.e., $\lambda(n_j)=1$). The set of cut nets is denoted by $\mathcal{N}_{cut}$. The *cutsize* of Π is defined in various ways. Two most commonly used cutsize definitions are the *cut-net* and the *connectivity* metrics [18], which are, respectively, defined as

$$cs_{cutn}(\Pi)=\sum_{n\in\mathcal{N}_{cut}} cost(n) \quad\text{and}\quad cs_{conn}(\Pi)=\sum_{n\in\mathcal{N}_{cut}}(\lambda(n)-1)cost(n). \tag{2.1}$$

Hypergraph partitioning (HP) is the problem of finding a K-way partition which minimizes the cutsize and satisfies the balance criterion $W_{max}\leq W_{avg}(1+\epsilon)$. Here, ϵ is the given maximum allowable imbalance ratio, and W_{max} and W_{avg}, respectively, denote the maximum and average part weights. HP with fixed vertices ensures the assignment of some preassigned vertices which are called *fixed vertices* to the respective parts.

The *recursive bipartitioning (RB)* is a widely used paradigm to obtain a K-way HP. It first partitions the hypergraph into two, and then each part is further bipartitioned recursively until reaching the desired number of parts K. In order to encode the cut-net and connectivity metrics, cut-net removal and cut-net splitting methods are utilized in the RB-based HP, respectively [18].

2.2. Sparse matrix partitioning with HP. Several HP models and methods have been proposed and successfully utilized for obtaining matrix partitioning [8, 14, 19, 20, 25, 39, 64, 67]. Among these, the most relevant one is the *column-net* model [18] that represents a given sparse matrix A as a hypergraph $\mathcal{H}_{CN}(A)$ in which nets and vertices, respectively, represent columns and rows. In this model, vertex v_i is added to the pin list of net n_j for each nonzero $A(i,j)$ in A. Throughout the paper, r_i and c_j, respectively, denote row i and column j.

A K-way ordered partition $\Pi=\langle\mathcal{V}_1,\mathcal{V}_2,\dots,\mathcal{V}_K\rangle$ of the column-net model $\mathcal{H}_{CN}(A)$ is decoded as a partial reordering of the rows of A in such a way that the rows corresponding to vertices in $\mathcal{V}_k$ are ordered before the rows corresponding to the vertices in $\mathcal{V}_\ell$ for $k<\ell$. This is a partial reordering since the rows corresponding to the vertices in the same part can be ordered arbitrarily. Let $\mathcal{B}^r_k$ denote the kth row block which contains the rows corresponding to the vertices in $\mathcal{V}_k$. We consider a symmetric row-column reordering that yields a 2D grid structure of A. The submatrix consisting of the rows of $\mathcal{B}^r_k$ and columns of ℓth column block $\mathcal{B}^c_\ell$ is referred as block-(k,ℓ) of A. A column is said to *connect* a row block $\mathcal{B}^r_k$ if it contains at least one nonzero in $\mathcal{B}^r_k$. A column is called *cut* if it connects more than one row block. For a matrix with nonzero diagonal entries, each column connects a diagonal block and becomes a cut column if it connects at least one off-diagonal block.

In the column-net model with unit net cost, the partitioning objective using the connectivity and cut-net metrics (2.1), respectively, encode the minimization of the number of nonzero column segments in off-diagonal blocks and the number of cut columns. The former partitioning objective is successfully utilized in encoding the minimization of the row parallel SpMV operations [18].

2.3. stSpike algorithm. We describe stSpike for lower triangular systems since the algorithm for the upper triangular case is similar. Given a lower triangular linear system of equations

$$Ly = b, \tag{2.2}$$

a DS factorization of sparse lower triangular matrix L is computed as $L = DS$, where D is the lower block diagonal of L and S is the spike matrix. These blocks are assumed to be obtained by matrix partitioning. Multiplying both sides of (2.2) from the left by D^{-1}, we obtain a modified system

$$Sy = g, \tag{2.3}$$

where $g = D^{-1}b$ and $S = D^{-1}L$. By splitting $L = D + R$, we obtain $S = I + G$, where $G = D^{-1}R$ and R is the block off-diagonal part of L. The sparse triangular system $DG = R$ with multiple right-hand-side vectors can be solved for the block rows of G independently with perfect parallelism.

The nonzero column segments of R constitute dense column segments (called *spikes*) in the off-diagonal blocks of S. The block diagonal of S is identity. Additional nonzeros (fill-in) are introduced within the off-diagonal blocks of S only in the locations below the top nonzero (having the smallest row index) for each nonzero column segment of R. The submatrix consisting of rows and columns $\mathcal{C}$ of S, namely, $\widehat{S} = S(\mathcal{C}, \mathcal{C})$, constitutes an independent reduced system where $\mathcal{C}$ is the set of nonzero columns of R, i.e., cut columns of L. Then the reduced system is of the form

$$\widehat{S}\widehat{y} = \widehat{g}, \tag{2.4}$$

where $\widehat{g} = g(\mathcal{C})$ and $\widehat{y} = y(\mathcal{C})$, which can be solved independently from the rest of the unknowns in y. After solving the reduced system, the only remaining computation for retrieving the solution of the original system is

$$y = g - D^{-1}(\widehat{R}\widehat{y}), \tag{2.5}$$

which can be obtained in perfect parallelism where $\widehat{R} = R(:, \mathcal{C})$ (in MATLAB notation). We only partially compute S just to form $\widehat{S}$, since forming S explicitly is expensive and requires a large amount of memory. Partial computation of S constitutes the factorization phase, whereas computation of $\widehat{g}$, solving (2.4) and (2.5) constitutes the solution phase of stSpike.

An example L matrix and the corresponding S and $\widehat{S}$ matrices are shown in Figure 1. The reduced system indices $\mathcal{C} = \{1, 3, 4, 6, 7, 9, 11\}$ are colored in red and circled. The nonzeros that constitute the reduced system are bold and colored in red. The background colors of the original nonzeros and possible fill-in are green and blue, respectively. Depending on the sparsity structure of the corresponding column and block diagonal, spikes may not fill the entire column segment. For example, nonzero $L(4, 1)$ in block-(2,1) of L leads to the spike consisting of three nonzeros in the first column of block-(2,1) of S.

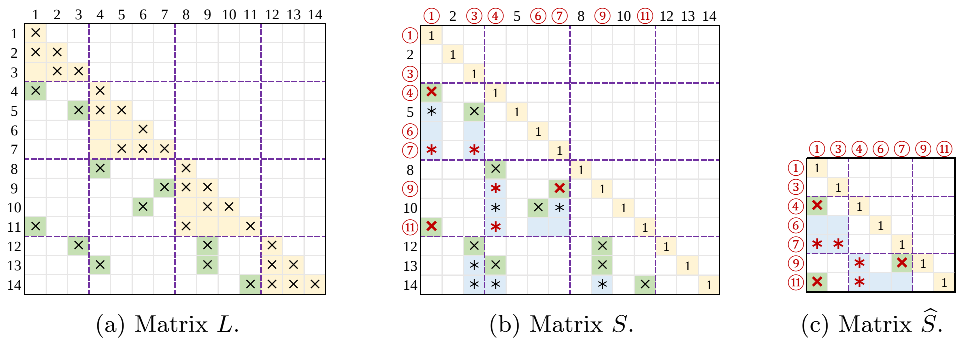


FIG. 1. *Sparsity structure of L and resulting S and $\widehat{S}$ matrices derived from stSpike.*

3. dmpGS algorithm. The pseudo-code of dmpGS is given in Algorithm 3.1 for processor P_k in a K-processor system. Matrix A is assumed to be partitioned into K row blocks, where m_k denotes the number of rows in the kth row block. In the algorithm, R_k, D_k, and U_k, respectively, denote the kth row block of the strictly block lower triangular, lower triangular part of the block diagonal, and strictly upper triangular parts of A as shown in Figure 2. The number of columns in R_k, D_k, and U_k are, respectively, $\sum_{i=1}^{k-1} m_i$, m_k, and $\sum_{i=k}^{K} m_i$. f_k, g_k, x_k, h_k, w_k, and z_k denote the local subvectors of size m_k that are computed by P_k. These subvectors are partitioned conformably with row-wise partitioning of A as shown in Figure 2. $\widehat{S}$, $\widehat{x}$, and $\widehat{g}$, respectively, denote the $|\mathcal{C}|\times|\mathcal{C}|$ coefficient matrix, $|\mathcal{C}|\times 1$ unknown, and $|\mathcal{C}|\times 1$ right-hand-side vectors of the reduced system in stSpike. $\mathcal{C}_k$ denotes the subset of $\mathcal{C}$ corresponding to the row indices in R_k.

In Algorithm 3.1, lines 2–7 denote the factorization phase of stSpike which computes $\widehat{S}$. This phase is done only once after which we proceed with the GS iterations in lines 8–22. Each dmpGS iteration involves two SpMVs at lines 11 and 20, two vector subtraction operations at lines 12 and 22, an independent sparse triangular solve at line 13, and a reduced system solution at line 17, which enables independent sparse triangular solves at line 21. The upper and lower triangular SpMV operations are incurred by the GS and stSpike algorithms, respectively. These two SpMV operations incur communication of x-vector entries depending on the sparsity structures of the upper triangular U and lower triangular L matrices, respectively. Conformable partitioning of the vectors avoids communication during vector subtraction operations.

At lines 9–10, communication operations are performed for local SpMV (line 11). After P_k receives all necessary nonlocal x-vector entries, it forms its augmented vector $\breve{x}$. Each processor sends the selected entries of its g_k vector to P_1 (line 14) to form

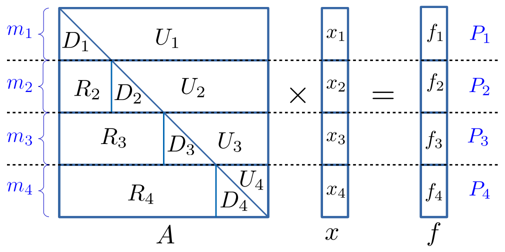


FIG. 2. *Four-way row-wise partition of matrix A and vectors x and f.*

Algorithm 3.1 dmpGS for processor P_k.

```
Input: Submatrices $R_k, D_k, U_k$, and right-hand-side subvector $f_k$
Output: Subvector $x_k$
1: Choose an initial guess for $x_k$
2: if $2 \le k \le K-1$ then
3:     $G_k \leftarrow D_k^{-1} R_k$     ▷ local partial sparse triangular solve with multiple right-hand sides
4:     Form and send $\widehat{G}_k$ to processor $P_1$
5: if $k = 1$ then
6:     Receive $\widehat{G}_\ell$ from $P_\ell$ for $2 \le \ell \le K-1$ to form $\widehat{G}$
7:     $\widehat{S} \leftarrow \widehat{G} + I$
8: while not converged do
9:     Send required local $x_k$ entries to respective processors in $\{P_1, \ldots, P_{k-1}\}$
10:    Receive nonlocal $x_\ell$ entries from processors in $\{P_{k+1}, \ldots, P_K\}$ to form $\breve{x}_k$
11:    $h_k \leftarrow U_k \breve{x}_k$     ▷ local SpMV
12:    $h_k \leftarrow f_k - h_k$
13:    $g_k \leftarrow D_k^{-1} h_k$     ▷ local sparse triangular solve
14:    if $2 \le k \le K-1$ then Send $\{g_k(i)\}_{i \in \mathcal{C}_k}$ to processor $P_1$
15:    if $k = 1$ then
16:        Receive $\{g_\ell(i)\}_{i \in \mathcal{C}_k}$ from $P_\ell$ for $2 \le \ell \le K-1$ to form $\widehat{g}$
17:        $\widehat{x} \leftarrow \widehat{S}^{-1} \widehat{g}$     ▷ solve reduced system
18:        Send $\widehat{x}$ entries to requiring processors
19:    if $k \ne 1$ then Receive required $\widehat{x}$-entries to form $\bar{x}_k$
20:    $z_k \leftarrow R_k \bar{x}_k$     ▷ local SpMV
21:    $w_k \leftarrow D_k^{-1} z_k$     ▷ local sparse triangular solve
22:    $x_k \leftarrow g_k - w_k$
```

the right-hand-side vector $\widehat{g}$ (line 16) for the sequential solution of the reduced system to obtain $\widehat{x}$ (line 17). Here $\widehat{x}$ corresponds to those unknowns in x which are at the interface of the partitioning of L, and obtaining them decouples the global lower triangular system into independent much smaller systems. P_1 sends only those x-vector entries that are required by other processors (line 18) so that each processor P_k forms its $\bar{x}$ vector (line 19) to perform local SpMV (line 20).

The communication overhead in each iteration of dmpGS is as follows. The communication volumes incurred by $h = Ux$ (line 11) and $z = Rx$ (line 20) are equal to the number of nonzero column segments in the off-diagonal blocks of U and L, respectively. Thus the communication volume required by these two SpMVs is equal to the total number of off-diagonal nonzero column segments in A ($\texttt{offD\_nzCol\_seg}(A)$). The volume of communication incurred at line 16 is equal to the size of the reduced system, $|\mathcal{C}|$. Therefore, the total communication volume of dmpGS is

$$\texttt{commVol} = \texttt{offD\_nzCol\_seg}(A) + |\mathcal{C}|. \tag{3.1}$$

Note that the different row blocks (R_k) seem to vary in the number of columns because of the triangular structure of the problem. On the other hand, the computational load imbalance is alleviated by the proposed partitioning model which also gathers most of the nonzeros to the diagonal blocks.

4. The proposed partitioning and reordering model.

We propose a two-phase model for reducing the communication overhead of dmpGS while maintaining computational balance as well as reducing the sequential computational overhead incurred by solving the reduced system at each iteration. This computational overhead is proportional to the number of nonzeros in the off-diagonals of $\widehat{S}$. In subsection 4.1, we propose a novel HP model as the first phase which simultaneously encodes the minimization of the reduced system size $|\mathcal{C}|$ and the communication volume. Decreasing $|\mathcal{C}|$ is important because it not only directly contributes to reducing the communication volume but also relates to decreasing the computational overhead. In subsection 4.2, we propose an in-block reordering method as the second phase which refines the improvement further by decreasing the number of nonzeros in $\widehat{S}$. We provide the illustrations showing the effect of the proposed partitioning and reordering model on a sample matrix in subsection 4.3.

4.1. HP model.

The partitioning objective in this phase is minimizing the sum of communication volume overhead (3.1) and sequential overhead costs with proper scaling:

$$\begin{aligned} PartObj &= commVol + (\alpha - 1)|\mathcal{C}| \\ &= (\texttt{offD\_nzCol\_segs}(A) + |\mathcal{C}|) + (\alpha - 1)|\mathcal{C}| \\ &= \texttt{offD\_nzCol\_segs}(A) + \alpha|\mathcal{C}|. \end{aligned} \tag{4.1}$$

Here α denotes the scaling factor between the effect of the reduced system size and the number of off-diagonal nonzero column segments on the overall overhead.

4.1.1. Definitions and layout.

We define a column as *L-cut* if it connects at least one off-diagonal block in the lower triangular part. That is, a column c_i in kth column block $\mathcal{B}^c_k$ is L-cut if it connects a row block $\mathcal{B}^r_\ell$ with $\ell > k$. Since L-cut columns of A are the nonzero columns of R, the number of L-cut columns (L-$\texttt{cut\_cols}$(A)) is equal to the reduced system size, $|\mathcal{C}|$. Therefore, the partitioning objective (4.1) can be rewritten as

$$PartObj = \texttt{offD\_nzCol\_segs}(A) + \alpha(L\text{-}\texttt{cut\_cols}(A)). \tag{4.2}$$

Let $\mathcal{H}_{CN}(A) = (\mathcal{V}, \mathcal{N})$ be the column-net hypergraph of an $m \times m$ sparse matrix A with nonzero diagonal entries. An ordered partition $\Pi_K = \langle \mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K \rangle$ of $\mathcal{H}_{CN}(A)$ is decoded as a partial symmetric row and column reordering of A as explained in section 2.2. Each net n_i of $\mathcal{H}_{CN}(A)$ connects vertex v_i since $A(i,i) \neq 0$ for each $1 \leq i \leq m$. A net n_i with $v_i \in \mathcal{V}_k$ is called *L-cut* if it connects at least one vertex part $\mathcal{V}_\ell$ such that $\ell > k$. The set of L-cut nets is denoted as $\mathcal{N}_{Lcut}$. We define a new type of cutsize, which we call the *L-cut-net metric*, as

$$cs_{Lcut}(\Pi_K) = \sum_{n \in \mathcal{N}_{Lcut}} cost(n). \tag{4.3}$$

Finally, the cost of partition Π_K is defined as the sum of connectivity metric with unit net cost and L-cut-net metric with net cost α, i.e.,

$$cost_{conn+Lcut}(\Pi_K) = \sum_{n \in \mathcal{N}_{cut}} (\lambda(n) - 1) + \alpha|\mathcal{N}_{Lcut}|. \tag{4.4}$$

Here, each cut net n incurs $\lambda(n) - 1$, and each L-cut net incurs α to the cutsize.

LEMMA 4.1. *A column c_i of A is L-cut iff net n_i of $\mathcal{H}_{CN}(A)$ is L-cut.*

Proof. Due to symmetric row-column ordering, c_i is in $\mathcal{B}^c_k$ iff r_i is in $\mathcal{B}^r_k$, which corresponds to $v_i \in \mathcal{V}_k$. Furthermore, c_i connects $\mathcal{B}^r_\ell$ iff n_i connects $\mathcal{V}_\ell$. Therefore, c_i in $\mathcal{B}^c_k$ connects $\mathcal{B}^r_\ell$ iff n_i with $v_i \in \mathcal{V}_k$ connects $\mathcal{V}_\ell$, where $\ell > k$. ∎

PROPOSITION 4.2. *Minimizing $cost_{conn+Lcut}(\Pi_K)$ for a K-way partition Π_K of $\mathcal{H}_{CN}(A)$ corresponds to minimizing the partitioning objective* (4.2).

Proof. By Lemma 4.1, the number of L-cut nets in $\mathcal{H}_{CN}(A)$ is equal to the number of L-cut columns in A. Thus $\alpha|\mathcal{N}_{Lcut}| = \alpha(L\texttt{-cut\_cols}(A))$. Furthermore, it is known by [18] that $\sum_{n\in\mathcal{N}_{cut}}(\lambda(n)-1) = \texttt{offD\_nzCol\_segs}(A)$. ∎

Each vertex is associated with a weight equal to the number of nonzeros in the respective row of the matrix, i.e., $w(v_i) = nnz(A(i,:))$. Thus, the partitioning constraint of maintaining balance on part weights approximately encodes the computational load balance during aggregate two triangular SpMVs (lines 11 and 20) and two triangular solves (lines 13 and 21).

The cut-net splitting technique has been successfully used within the RB framework to encode the minimization of the connectivity metric [18]. However, to the best of our knowledge, there exists no tool or model for encoding the minimization of the L-cut-net metric in the literature. We propose to use the RB framework with novel net anchoring and splitting schemes to encode the minimization of the L-cut-net metric.

4.1.2. RB model. At each RB step, an ordered bipartition $\Pi_2 = \langle \mathcal{V}_U, \mathcal{V}_L\rangle$ of $\mathcal{V}$ is decoded as ordering the vertices of $\mathcal{V}_L$ after those of $\mathcal{V}_U$. Here $\mathcal{V}_U$ and $\mathcal{V}_L$ denote the upper and lower vertex parts, respectively. In RB, the concept of L-cut net takes a special form. In a bipartition $\Pi_2 = \langle \mathcal{V}_U, \mathcal{V}_L\rangle$, a net n_i is L-cut if v_i is assigned to $\mathcal{V}_U$ and n_i connects at least one vertex v_j such that $v_j \in \mathcal{V}_L$. The partitioning objective at each RB step is to minimize

$$cost_{RB}(\Pi_2) = |\mathcal{N}_{cut}| + \alpha|\mathcal{N}_{Lcut}|. \tag{4.5}$$

For encapsulating the connectivity and L-cut-net metrics simultaneously, each net n_i in $\mathcal{H}_{CN}(A)$ is replicated as two different kinds of nets, namely, *conn-net* n^c_i and *lcn-net* n^ℓ_i. Here, conn-nets encapsulate the connectivity metric, whereas lcn-nets encapsulate the L-cut-net metric. The motivation for net replication is the requirement of different net splitting and net removal procedures for encoding the connectivity and L-cut-net metrics at each RB step. In order to encapsulate the RB objective (4.5), we assign unit cost to the conn-nets and cost α to the lcn-nets. We refer to the hypergraph formed by these replicated nets as $\mathcal{H}$.

We extend $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ into a hypergraph $\mathcal{H}' = (\mathcal{V}', \mathcal{N}')$ so that minimizing the number of conventional cut nets in $\mathcal{H}'$ encodes minimizing (4.5). We introduce new fixed vertices $v_U \in \mathcal{V}_U$ and $v_L \in \mathcal{V}_L$ to form the extended vertex set $\mathcal{V}' = \mathcal{V} \cup \{v_U, v_L\}$. We represent each lcn-net n^ℓ_i in $\mathcal{H}$ as a pair of nets $\hat{n}^\ell_i$ and $\check{n}^\ell_i$ in $\mathcal{H}'$. $\hat{n}^\ell_i$ is the same as n^ℓ_i except it is U-anchored (connects v_U). $\check{n}^\ell_i$ is a 2-pin L-anchored net which connects v_L and v_i. That is, for each net n_i in $\mathcal{H}_{CN}(A)$, $\mathcal{H}'$ contains nets n^c_i, $\hat{n}^\ell_i$, and $\check{n}^\ell_i$, where

$$\begin{aligned}
&Pins(n^c_i, \mathcal{H}') = Pins(n_i, \mathcal{H}_{CN}(A)),\\
&Pins(\hat{n}^\ell_i, \mathcal{H}') = Pins(n_i, \mathcal{H}_{CN}(A)) \cup \{v_U\}, \text{ and}\\
&Pins(\check{n}^\ell_i, \mathcal{H}') = \{v_i, v_L\}.
\end{aligned}$$

The nets in the extended hypergraph for a sample 3-pin net are shown in Figure 3.

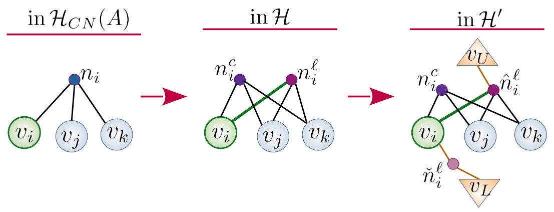


FIG. 3. *Net n_i in $\mathcal{H}_{CN}(A)$ is replicated as conn-net n_i^c and lcn-net n_i^ℓ to form $\mathcal{H}$. Net n_i^ℓ in $\mathcal{H}$ is represented by a pair of nets $\hat{n}_i^\ell$ and $\check{n}_i^\ell$ in $\mathcal{H}'$.*

We form $\mathcal{H}'$ at the beginning and apply RB steps until reaching the desired part count, K. The resulting K-way partition Π'_K of $\mathcal{H}'$ induces a K-way partition Π_K of $\mathcal{H}_{CN}(A)$. $\mathcal{H}$ is an in-between hypergraph introduced for the sake of clarity of presentation and is not constructed during implementation. We explain the proposed net splitting and removal methods on $\mathcal{H}$ and show the correspondence on $\mathcal{H}'$. We consider that each bipartition $\Pi'_2 = \langle \mathcal{V}'_U, \mathcal{V}'_L \rangle$ of $\mathcal{H}'$ induces a bipartition $\Pi_2 = \langle \mathcal{V}_U, \mathcal{V}_L \rangle$ of $\mathcal{H}$. Here $\mathcal{H}$ and $\mathcal{H}'$ refer to the respective hypergraphs just before the current RB step. New hypergraphs $\mathcal{H}_U$ and $\mathcal{H}_L$ are constructed according to $\Pi_2 = \langle \mathcal{V}_U, \mathcal{V}_L \rangle$ as follows. For both conn- and lcn-nets, each internal net in $\mathcal{V}_L$ and $\mathcal{V}_U$ is, respectively, included in $\mathcal{N}_L$ and $\mathcal{N}_U$ as is. In the net splittings, a new conn- or lcn-net is added to the net list of $\mathcal{H}_U$ or $\mathcal{H}_L$ only if it has more than one pin. The single-pin nets are discarded since they cannot contribute to the cutsize in the following RB steps.

For cut conn-nets, we apply the conventional cut-net splitting procedure [18] to encapsulate the connectivity metric. If a conn-net n_i^c is cut, then n_i^c is split into two pinwise disjoint nets in $\mathcal{H}_U$ and $\mathcal{H}_L$ such that

$$Pins(n_i^c, \mathcal{H}_U) = Pins(n_i^c, \mathcal{H}) \cap \mathcal{V}_U \text{ and } Pins(n_i^c, \mathcal{H}_L) = Pins(n_i^c, \mathcal{H}) \cap \mathcal{V}_L.$$

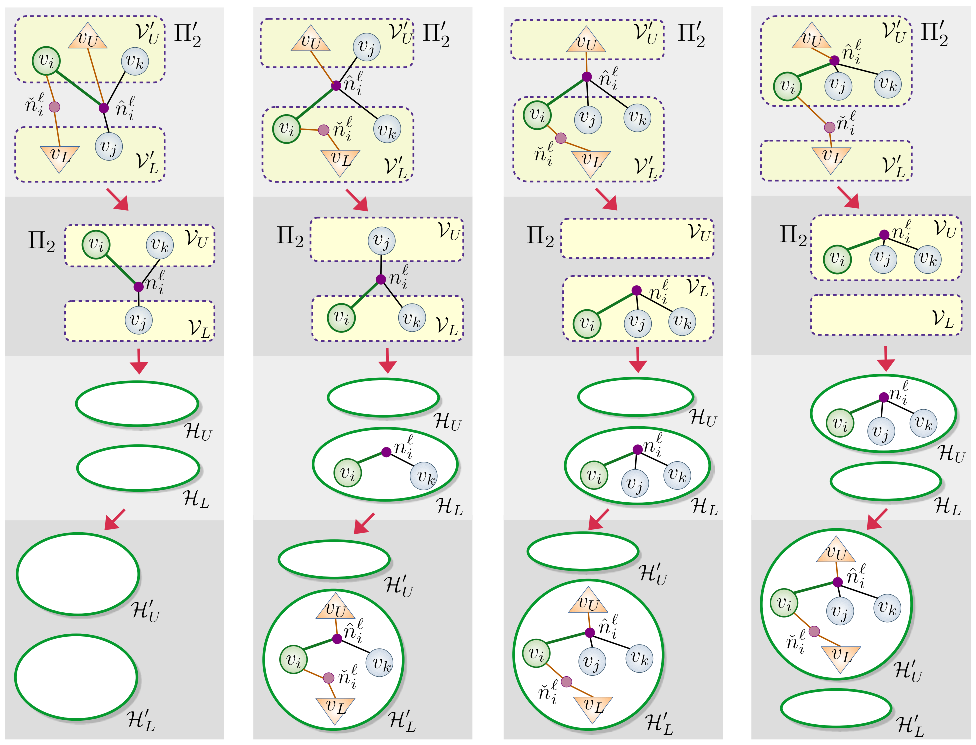
For lcn-nets, we introduce a hybrid cut-net splitting/removal method in order to correctly encapsulate the L-cut-net metric. At each RB step, for each net pair $(\hat{n}_i^\ell, \check{n}_i^\ell)$ in a bipartition Π', we consider the state of n_i^ℓ in Π where $Pins(n_i^\ell, \mathcal{H}) = Pins(\hat{n}_i^\ell, \mathcal{H}') - \{v_U\}$ for ease of understanding. If an lcn-net n_i^ℓ is not internal, then it can be L-cut or "cut but not L-cut."

If n_i^ℓ is L-cut, then we apply cut-net removal for n_i. This is because when n_i is L-cut in an RB step, it also becomes L-cut in the final K-way partition. Hence there is no need to track this net anymore, and we do not include it in further bipartitions.

If n_i^ℓ is cut but not L-cut, then we apply net removal towards $\mathcal{H}_U$ and *net-L-splitting* towards $\mathcal{H}_L$. That is, n_i^ℓ is added to $\mathcal{H}_L$ as $Pins(n_i^\ell, \mathcal{H}_L) = Pins(n_i^\ell, \mathcal{H}) \cap \mathcal{V}_L$. This is because n_i^ℓ cannot be L-cut in further bipartitionings of $\mathcal{H}_U$ but it has the potential of becoming L-cut in further bipartitionings of $\mathcal{H}_L$. In the extended hypergraph context, this corresponds to adding lcn-net pair $(\hat{n}_i^\ell, \check{n}_i^\ell)$ to $\mathcal{H}'_L$ such that

$$Pins(\hat{n}_i^\ell, \mathcal{H}'_L) = (Pins(n_i^\ell, \mathcal{H}') \cap \mathcal{V}'_L) \cup \{v_U\} \text{ and } Pins(\check{n}_i^\ell, \mathcal{H}'_L) = \{v_i, v_L\}.$$

Figure 4 shows all possible cases for a sample lcn-net. The first, second, third, and last horizontal layers, respectively, show the bipartition Π'_2 of $\mathcal{H}'$, the corresponding bipartition Π_2 of $\mathcal{H}$, $\mathcal{H}_U$ and $\mathcal{H}_L$ induced by Π_2, and the corresponding $\mathcal{H}'_U$ and $\mathcal{H}'_L$ induced by Π'_2. If n_i^ℓ is L-cut in $\mathcal{H}$ as in Figure 4(a), both $\hat{n}_i^\ell$ and $\check{n}_i^\ell$ are cut in Π'_2. If n_i^ℓ is cut but not L-cut as in Figure 4(b), or if n_i^ℓ is internal to $\mathcal{V}_L$ as in Figure 4(c), then only $\hat{n}_i^\ell$ is cut. Otherwise, if n_i^ℓ is internal to $\mathcal{V}_U$ as in Figure 4(d), then only $\check{n}_i^\ell$ is cut.



(a) n_i^ℓ is L-cut. (b) n_i^ℓ is cut, not L-cut. (c) n_i^ℓ is internal to $\mathcal{V}_L$. (d) n_i^ℓ is internal to $\mathcal{V}_U$.

FIG. 4. *All cases for n_i^ℓ and corresponding net pair $(\hat{n}_i^\ell, \check{n}_i^\ell)$ after bipartition $\langle \mathcal{V}_U', \mathcal{V}_L' \rangle$.*

LEMMA 4.3. *Consider the bipartition $\Pi_2 = \langle \mathcal{V}_U, \mathcal{V}_L \rangle$ of $\mathcal{H}$ induced by a bipartition $\Pi_2' = \langle \mathcal{V}_U', \mathcal{V}_L' \rangle$ of $\mathcal{H}'$ in an RB step. If a net is L-cut in Π_2, then it incurs 2 cut nets in Π_2'. Conversely, if a net is not L-cut in Π_2, then it incurs 1 cut net in Π_2'.*

Proof. If n_i^ℓ is L-cut in Π_2 of $\mathcal{H}$, then $v_i \in \mathcal{V}_U$ and n_i^ℓ connects a vertex v_j such that $v_j \in \mathcal{V}_L$. In Π_2' of $\mathcal{H}'$, $\hat{n}_i^\ell$ is cut since it connects $v_i \in \mathcal{V}_U'$ and $v_j \in \mathcal{V}_L'$, and $\check{n}_i^\ell$ is also cut since it connects $v_i \in \mathcal{V}_U'$ and $v_L \in \mathcal{V}_L'$.

If n_i^ℓ is not L-cut and $v_i \in \mathcal{V}_L$ in Π_2, then $\hat{n}_i^\ell$ is cut in Π_2' because it connects $v_U \in \mathcal{V}_U'$ and $v_i \in \mathcal{V}_L'$, but $\check{n}_i^\ell$ is not cut since both v_i and v_L are in $\mathcal{V}_L'$.

If n_i^ℓ is not L-cut and $v_i \in \mathcal{V}_U$ in Π_2, then n_i^ℓ should be internal to $\mathcal{V}_U$, because otherwise any pin in $\mathcal{V}_L$ would make n_i^ℓ to be L-cut. In Π_2', net $\hat{n}_i^\ell$ is internal to $\mathcal{V}_U'$ since both v_i and v_U are in $\mathcal{V}_U'$, but $\check{n}_i^\ell$ is cut since it connects $v_i \in \mathcal{V}_U'$ and $v_L \in \mathcal{V}_L'$. ◻

PROPOSITION 4.4. *Minimizing the conventional cut-net metric for the bipartition Π_2' of $\mathcal{H}'$ encodes minimizing $cost_{RB}(\Pi_2)$ defined in* (4.5).

Proof. By Lemma 4.3, each L-cut net in Π_2 incurs 2 cut nets in Π_2', whereas all remaining nets in Π_2 incur 1 cut net in Π_2'. Since the cost of lcn-nets is α, the cutsize incurred by lcn-nets in Π_2' is $\alpha(|\mathcal{N}_{Lcut}| + |\mathcal{N}|)$. Since conn-nets are of unit cost, they incur $|\mathcal{N}_{cut}|$ to the cutsize of Π_2'. Hence the total cutsize of Π_2' is $|\mathcal{N}_{cut}| + \alpha|\mathcal{N}_{Lcut}| + \alpha|\mathcal{N}|$. Since $\alpha|\mathcal{N}|$ is constant, minimizing the cutsize of Π_2' is equivalent to minimizing $|\mathcal{N}_{cut}| + \alpha|\mathcal{N}_{Lcut}|$, which is $cost_{RB}(\Pi_2)$. ◻

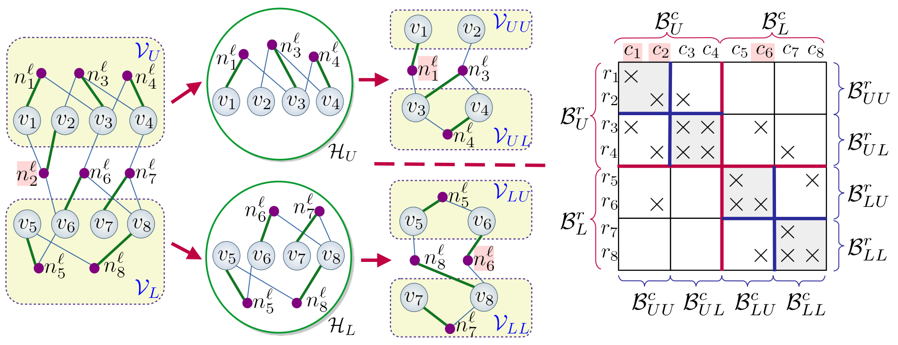


FIG. 5. *Sample 2-level RB showing lcn-nets and corresponding matrix partitioning.*

Figure 5 shows an example 2-level RB in terms of lcn-nets in $\mathcal{H}$ and the corresponding 4-way matrix partitioning. The L-cut nets n_1^ℓ, n_2^ℓ, and n_6^ℓ and the corresponding L-cut columns c_1, c_2, and c_6 of A are colored in red background. n_2^ℓ is L-cut in the first level RB and discarded in the future bipartitions. This is because column c_2 is already counted as L-cut due to nonzero $A(6,2)$ and should not be counted as L-cut again due to nonzero $A(4,2)$ in further bipartitions.

Note that the L-cut net definition can be considered to be similar to the left-cut net defined in [1] for encapsulating the profile minimization, but the net splitting and removal strategies are quite different for encapsulating the objective of our problem.

THEOREM 4.5. *Recursively bipartitioning $\mathcal{H}'$ by minimizing the cutsize according to the cut-net metric and applying the proposed net splitting and removal strategies until reaching K parts encode minimizing the partitioning objective* (4.2).

Proof. By Proposition 4.4, recursively bipartitioning $\mathcal{H}'$ by minimizing the conventional cut-net metric encodes minimizing $cost_{RB}(\Pi_2)$ at each RB step. We show that this encodes minimizing $cost_{conn+Lcut}(\Pi_K)$. Proposed net splitting and removal strategies ensure that an L-cut net in Π_K is also L-cut in Π_2 in exactly one RB step. Since an L-cut net contributes α to both $cost_{RB}(\Pi_2)$ and $cost_{conn+Lcut}(\Pi_K)$, minimizing $\alpha|\mathcal{N}_{Lcut}|$ in each bipartition Π_2 encodes minimizing $\alpha|\mathcal{N}_{Lcut}|$ in Π_K. Furthermore, minimizing the number of cut nets $|\mathcal{N}_{cut}|$ at each RB step and applying the cut-net splitting procedure encodes minimizing the connectivity metric $\sum_{n\in\mathcal{N}_{cut}}(\lambda(n){-}1)$ [18]. Therefore, minimizing the cutsize for each bipartition Π_2' of $\mathcal{H}'$ encodes minimizing $cost_{conn+Lcut}(\Pi_K)$; hence by Proposition 4.2, this corresponds to the partitioning objective (4.2). ∎

4.2. Reordering within row blocks. Consider the K-way block structure (e.g., Figure 2) of A induced by the partial symmetric row-column permutation obtained by the HP model (section 4.1). We perform row reordering within the kth row block of A by considering nonzeros of the kth row block R_k of R. The resulting row reordering within the kth row block of A is symmetrically applied to the columns of the kth column block of A. R_k is an $m_k \times z_k$ matrix, where $z_k = \sum_{i=1}^{k-1} m_i$. For simplicity, we assume a local indexing for the rows of R_k so that R_k consists of rows r_i with $1 \le i \le m_k$.

Recall that in stSpike, fill-in may arise below the top nonzero of each spike in R_k. The top nonzero of a spike c_j in R_k is the nonzero with the minimum row index, i.e., $top(c_j, R_k) = \min\{i : R_k(i,j) \neq 0,\ 1 \le i \le m_k\}$. We define the *height* of a spike c_j in R_k

as the number of reduced system row indices between $top(c_j, R_k)$ and m_k inclusively, i.e.,

$$height(c_j, R_k) = |\{i : top(c_j, R_k) \le i \le m_k,\ i \in \mathcal{C}_k\}|, \tag{4.6}$$

since only the rows with indices in $\mathcal{C}_k$ may contribute to the nonzero count of $\widehat{S}$. The height of a spike in R_k constitutes an upper bound on the nonzero count (including the fill-in) of the corresponding column in $\widehat{S}$. In Figure 1(b), the heights of the spikes are as follows: $height(c_1, R_2)$=3, $height(c_3, R_2)$=2; $height(c_1, R_3)$=1, $height(c_4, R_3)$=2, $height(c_6, R_3)$=1, and $height(c_7, R_3)$=2. The height of a nonspike column is assumed to be zero. The objective of in-block reordering is to minimize the *total height*

$$\sum_{k=2}^{K-1} \sum_{j=1}^{z_k} height(c_j, R_k), \tag{4.7}$$

which constitutes an upper bound on the nonzero count in off-diagonal blocks of $\widehat{S}$. The last block R_K does not contribute nonzeros to $\widehat{S}$ since $\mathcal{C}_K$ is empty. Reorderings within different blocks are completely independent and can be done concurrently.

One straightforward approach is placing the rows whose indices are not among $\mathcal{C}_k$ to the bottom of R_k to avoid the nonzeros of the rows that are not in $\mathcal{C}_k$ to contribute to (4.7). Let $\overline{R}_k = R_k(\mathcal{C}_k, :)$ be the $|\mathcal{C}_k| \times z_k$ submatrix of R_k consisting of the rows with indices in $\mathcal{C}_k$. Then the problem is reduced to reordering only those rows of $\overline{R}_k$ since the rest of the rows at the bottom of R_k do not have an impact on (4.7). The reordering objective for each $\overline{R}_k$ is to minimize $\sum_{j=1}^{z_k} height(c_j, \overline{R}_k)$ with a simplified height definition, $height(c_j, \overline{R}_k) = |\mathcal{C}_k| + 1 - top(c_j, \overline{R}_k)$. Then the total height minimization problem is formulated in general as follows: Given any sparse matrix $H \in \mathbb{R}^{\ell \times n}$, find a row reordering P that minimizes $\sum_{j=1}^{n} (\ell + 1 - top(c_j, PH))$.

THEOREM 4.6. *The total height minimization problem (THMP) is NP-hard.*

Proof. We reduce the profile minimization problem (PMP) [1, 46], which is known to be NP-hard [26, 44], to THMP as follows. Given a symmetric matrix $V \in \mathbb{R}^{n \times n}$ with nonzero diagonal entries, the objective of PMP is finding a symmetric row-column reordering P that minimizes $\sum_{j=1}^{n} (j - top(c_j, PVP^T))$. This minimization objective is equivalent to maximizing $\sum_{j=1}^{n} top(c_j, PVP^T)$, since $\sum_{j=1}^{n} j$ is constant. Any instance PVP^T of PMP can be mapped to an instance PV of THMP by simply removing the column reordering as $(PVP^T)P = PV$. Note that the minimization objective of THMP, which is $\sum_{j=1}^{n} (n+1 - top(c_j, PV))$, is equivalent to maximizing $\sum_{j=1}^{n} top(c_j, PV)$, since $\sum_{j=1}^{n} (n+1)$ is constant. Thus, PVP^T is a solution of PMP iff PV is a solution of THMP since the column reordering itself has no effect on $\sum_{j=1}^{n} top(c_j, PVP^T) = \sum_{j=1}^{n} top(c_j, PV)$. If there were a polynomial-time solution to THMP, then one could solve PMP in polynomial time by just applying the row reordering obtained by THMP on the columns as well. Therefore, PMP can be reduced to THMP in polynomial time, and since PMP is NP-hard, then so is THMP. $\square$

Algorithm 4.1 presents the pseudocode of the proposed heuristic for reordering the rows of R_k. The efficient implementation of this algorithm requires accessing the nonzeros of both rows and columns of $\overline{R}_k$, so it is stored in both compressed sparse row and compressed sparse column formats. $Cols(r_i)$ denotes the set of columns in row r_i, whereas $Rows(c_j)$ denotes the set of rows in column c_j. Degree of a row or column is defined as the number of nonzeros in that row or column, i.e., $deg(r_i) = |Cols(r_i)|$

Algorithm 4.1 Proposed in-block reordering for R_k where $2 \le k \le K-1$.

```
Input: $R_k \in \mathbb{R}^{m_k \times z_k}$ and set of reduced system row indices $\mathcal{C}_k$ of $R_k$.
Output: the permutation vector $perm$ of $R_k$.
1: Place the rows $r_i$ with $i \notin \mathcal{C}_k$ to the last $m_k - |\mathcal{C}_k|$ indices in any order
2: Consider submatrix $\overline{R}_k = R_k(\mathcal{C}_k, :)$ of $R_k$ consisting of rows $r_i$ with $i \in \mathcal{C}_k$
3: for each row $r_i$ of $\overline{R}_k$ do
4:     $load(r_i) \leftarrow 0$
5:     for each column $c_j \in Cols(r_i)$ do $load(r_i) \leftarrow load(r_i) + deg(c_j)$
6: for $d \leftarrow 0$ to max_row_deg do $\mathcal{S}(d) \leftarrow \{r_i : deg(r_i) = d\}$
7: $indx \leftarrow 0$
8: while $indx < |\mathcal{C}_k|$ do
9:     $d^* \leftarrow \min\{d : \mathcal{S}(d) \neq \varnothing\}$
10:    $r_{i^*} \leftarrow \text{argmax}_{r_i \in \mathcal{S}(d^*)} load(r_i)$        ▷ Select $r_{i^*} \in \mathcal{S}(d^*)$ with maximum load
11:    $indx \leftarrow indx + 1$
12:    $perm(indx) \leftarrow r_{i^*}$
13:    $\mathcal{S}(d^*) \leftarrow \mathcal{S}(d^*) - \{r_{i^*}\}$
14:    for each column $c_j \in Cols(r_{i^*})$ do
15:        $Rows(c_j) \leftarrow Rows(c_j) - \{r_{i^*}\}$
16:        $Cols(r_{i^*}) \leftarrow Cols(r_{i^*}) - \{c_j\}$
17:        for each row $r_{i'} \in Rows(c_j)$ do
18:            $Cols(r_{i'}) \leftarrow Cols(r_{i'}) - \{c_j\}$
19:            $load(r_{i'}) \leftarrow load(r_{i'}) - deg(c_j)$
20:            $\mathcal{S}(deg(r_{i'})) \leftarrow \mathcal{S}(deg(r_{i'})) - \{r_{i'}\}$
21:            $deg(r_{i'}) \leftarrow deg(r_{i'}) - 1$
22:            $\mathcal{S}(deg(r_{i'})) \leftarrow \mathcal{S}(deg(r_{i'})) \cup \{r_{i'}\}$
```

and $deg(c_j) = |Rows(c_j)|$. In lines 3–5, we define the *load* of each row r_i as the sum of degrees of columns c_j such that $\overline{R}_k(i,j) \neq 0$.

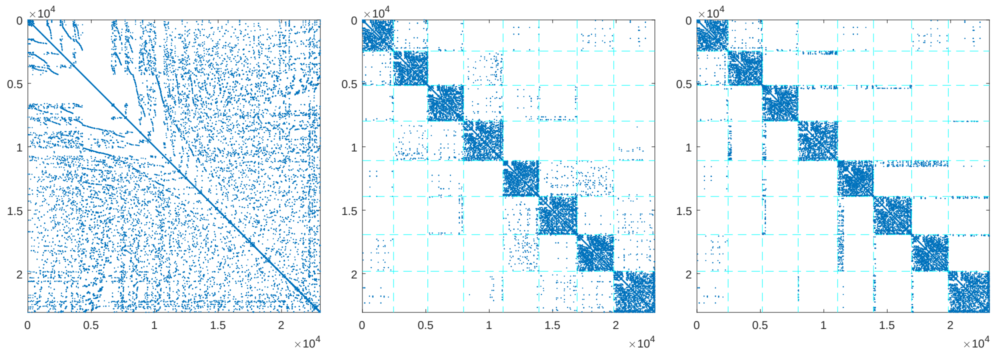
The greedy choice utilized in the proposed heuristic is to order the rows with smaller degrees to upper positions of $\overline{R}_k$ since placing denser rows to upper positions incurs more height in (4.7). We further improve our greedy approach by using dynamic row degrees during the row selection process. When a row is selected, the degree of each unselected row is decremented by the number of its nonzeros having the same column index with the nonzeros in the selected row. Since the nonzeros in a selected row already determine the heights of the respective columns, we do not need to consider the rest of the nonzeros of these columns in future row selections. When selecting a row among rows with the same degree, load values of the rows are used as a tie-breaking strategy. A row with a higher load is preferred to be selected since it will lead to a larger amount of decrease on the degrees of unselected rows.

In Algorithm 4.1, $\mathcal{S}(d)$ denotes the set of rows with degree d. Due to dynamic row degrees, at each iteration we find the minimum degree d^* (line 9). Then we choose the row r_{i^*} in $\mathcal{S}(d^*)$ with the maximum load (line 10). After r_{i^*} is selected, all remaining nonzeros in each column c_j with $R_k(i^*, j) \neq 0$ are deleted as in lines 15–18. For each unselected row $r_{i'}$ with $R_k(i', j) \neq 0$, we dynamically update the load and degree of $r_{i'}$ and the respective degree sets (lines 19–22).

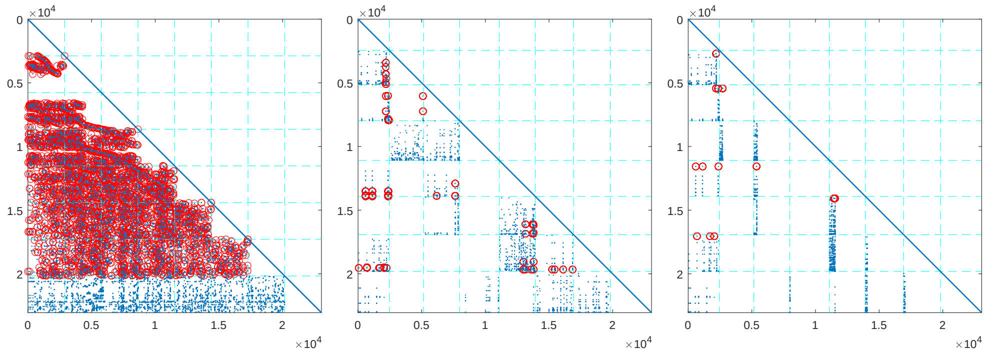
Recall that forming $\widehat{S}$ in dmpGS requires the computation of nonzeros up to the largest reduced system row index and any entry beyond that is not required to be computed for each row block. Hence the total height (4.7) also gives the computational cost of forming $\widehat{S}$ since we place $\overline{R}_k$ at the top of R_k for each $1 < k < K$.

4.3. Illustration. Figure 6 illustrates the effect of applying the proposed partitioning and reordering model for K=8 on a sample matrix (`msc23052`) from the SuiteSparse Matrix Collection [24]. The nonzero structure of the original matrix, the structure obtained after applying the proposed HP model, and the final structure after the proposed in-block reordering are shown in order. Below each ordering of A, the resulting spike matrix (S) is shown, including the nonzeros of the reduced system ($\widehat{S}$) which are highlighted with red circles. As seen in the figure, the proposed partitioning and reordering model significantly reduces the nonzero count of the reduced system. For example, the number of nonzeros in $\widehat{S}-I$ in Figure 6(d), (e), and (f) are 277,113, 3,593, and 811, respectively. Note that these numbers may seem to be much larger than the ones appearing in the figures because of the overlapping red circles.

Notice that the proposed HP model gathers most of the nonzeros to the diagonal blocks so that the off-diagonal blocks become very sparse. Then, the proposed in-block reordering method gathers the reduced system nonzeros to the upper left corner of the respective off-diagonal block (Figure 6(f)). This is because we agglomerate the reduced system row indices to the top within each block, and we apply the resulting row reordering to the columns symmetrically. Within each off-diagonal block, gathering the rows with reduced system indices to the top corresponds to agglomerating the columns with these indices, which are actually all the columns having nonzeros, to



(a) A: original. (b) A: HP ordering. (c) A: HP&in-block ordering.



(d) S: original. (e) S: HP ordering. (f) S: HP&final ordering.

Fig. 6. *Nonzero structure of* `msc`23052*:* (a) *before ordering,* (b) *after HP for* $K = 8$, (c) *after HP and in-block reordering;* (d), (e), (f) *the respective spike (S) matrices (the reduced system ($\widehat{S}$) nonzeros are circled in red color).*

the left. An exception is the first column block since no row reordering is performed for the first row block.

5. Experiments.

We use the HSL software package MC64 [29] for scaling and permuting the coefficient matrices to avoid a singular L. We select the MC64 option that maximizes the product of the diagonal entries and then scales to make the absolute value of diagonal entries one and the off-diagonal entries less than or equal to one. For symmetric matrices, in order not to destroy the symmetry, we apply the symmetric MC64 if the main diagonal is already zero-free. Otherwise, we apply the nonsymmetric MC64 to obtain a zero-free main diagonal. For unsymmetric matrices, we just apply the nonsymmetric MC64.

The experiments are conducted on an extensive dataset obtained from the SuiteSparse Matrix Collection [24]. For sufficiently coarse-grained parallel processing, we select real square matrices that have more than 20,000 rows and between 100,000 and 20,000,000 nonzeros. There are 199 symmetric and 208 unsymmetric matrices in SuiteSparse satisfying these properties at the time of experimentation. 44 symmetric and 4 unsymmetric matrices are eliminated because they are singular. The remaining are 155 symmetric and 204 unsymmetric, a total of 359 sparse matrices on which we conduct experiments. Table 1 shows the number of instances for each matrix kind. Kinds are sorted in decreasing order of instance count. The kinds having fewer than 5 instances in our dataset (acoustics, chemical oceanography, counter-example, and data analytics) are grouped as one kind.

5.1. Partitioning quality.

We tested the performance of the proposed partitioning algorithm described in subsection 4.1 against the partitioning quality of the conventional column-net HP with connectivity metric (cnHP) and graph partitioning (GP) models. For both cnHP and GP, vertex weights are set as the number of nonzeros in the respective rows, whereas nets and edges are assigned unit cost. In cnHP, the objective is to minimize the number of nonzero off-diagonal column segments. In GP, the objective is to minimize the number of nonzeros in the off-diagonal blocks. For unsymmetric matrices, GP is applied on $|A| + |A^T|$. The well-known partitioning tools METIS [40] and PaToH [19] are used for GP and cnHP models, respectively.

TABLE 1
Number of instances among different matrix kinds in the dataset.

Kind ID	Kind name	Sym	Unsym	Total
1	structural	48	4	52
2	circuit simulation	2	46	48
3	economic	1	33	34
4	semiconductor device	0	33	33
5	computational fluid dynamics	6	27	33
6	2D/3D	19	9	28
7	power network	14	13	27
8	optimization	20	3	23
9	model reduction	13	3	16
10	chemical process simulation	0	15	15
11	theoretical/quantum chemistry	14	0	14
12	electromagnetics	6	4	10
13	thermal	5	4	9
14	materials	2	4	6
15	weighted graph	1	5	6
16	acoustics, oceanography, counter-ex., analytics	4	1	5
	All	155	204	359

In the proposed model, we use PaToH as the HP tool in each bipartitioning step. Experiments are conducted with different scaling factors $\alpha = 1, 2, 5$, and 10 for lcn-net cost assignment. We set the maximum allowable imbalance ratio in each bipartitioning as $\epsilon = 0.05$. As both METIS and PaToH involve randomized algorithms in the coarsening phase, five partitioning runs are performed for each instance with different seeds, and the averages are reported. We conduct experiments for $K = 8, 16, 32, 64, 128$, and 256 parts (processors).

Table 2 shows the results of the comparison experiments in terms of the communication volume and the reduced system size metrics for dmpGS utilizing the partitions generated by GP, cnHP, and the proposed model. For each test instance, these metrics are normalized with respect to the number of rows, and the average for all matrices are given for each K. Here and hereafter, all averages are given as geometric means.

As seen in Table 2, cnHP achieves considerably lower communication volume and reduced system size compared with GP, as expected. The average improvement of cnHP over GP is approximately 10% for both metrics on $K = 256$. In fact, cnHP is equivalent to the proposed HP model for $\alpha = 0$. As seen in the table, there is a trade-off between the reduced system size and the communication volume for varying values of α for the proposed HP model. Yet the rate of increase in the communication volume is observed to be larger than the rate of decrease in the reduced system size with increasing α. For example, for $K = 64$, compared to the cnHP model, the proposed model slightly increases the communication volume by 0.4%, 0.5%, 2.9%, and 5.9%, whereas it significantly decreases the reduced system size by 21.5%, 25.2%, 30.7%, and 32.0% for $\alpha = 1, 2, 5$, and 10, respectively. Here, $\alpha = 2$ seems to be a balanced choice since it significantly decreases the reduced system size while it slightly increases the communication volume. This is reflected in the parallel scalability of the proposed algorithm, as will be shown in subsection 5.3; thus we set $\alpha = 2$ in the upcoming results.

In Figure 7, we provide the performance profiles comparing GP, cnHP, and the proposed model in terms of the reduced system size. We present the performance profiles only for $K = 16, 64$, and 256 due to lack of space. A performance profile [27] shows the comparison of different models relative to the best performing one for each data instance. On a profile, a point (x, y) means that the respective model is within x factor of the best result for a fraction y of the instances. For example, the point (1.20,

TABLE 2
Averages of total communication volume and the reduced system size in dmpGS, both normalized with respect to the number of rows.

				Proposed HP model (sec. 4.1)			
	K	GP	cnHP	$\alpha = 1$	$\alpha = 2$	$\alpha = 5$	$\alpha = 10$
Comm. vol.	8	0.158	0.132	0.140	0.139	0.139	0.145
	16	0.253	0.217	0.223	0.224	0.224	0.232
	32	0.380	0.329	0.332	0.332	0.337	0.347
	64	0.547	0.477	0.479	0.479	0.491	0.505
	128	0.767	0.681	0.679	0.680	0.697	0.719
	256	1.062	0.955	0.948	0.953	0.977	1.012
Red. sys. size	8	0.048	0.041	0.033	0.032	0.029	0.029
	16	0.075	0.066	0.051	0.049	0.045	0.045
	32	0.109	0.094	0.074	0.070	0.065	0.064
	64	0.149	0.129	0.102	0.097	0.090	0.088
	128	0.197	0.174	0.136	0.129	0.119	0.116
	256	0.252	0.227	0.177	0.168	0.154	0.149

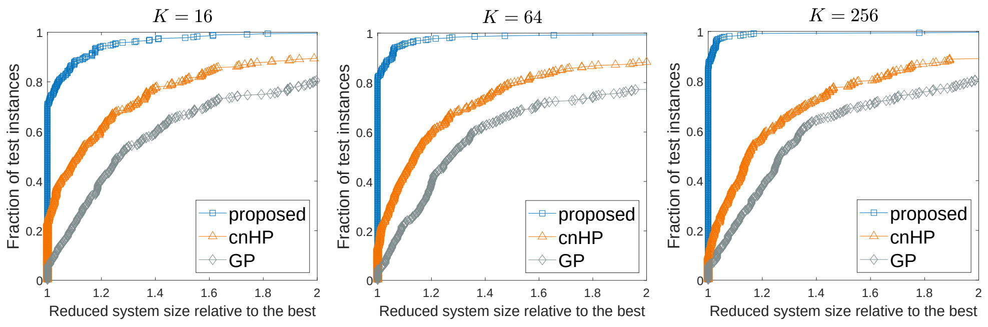


FIG. 7. *Performance profiles comparing GP, cnHP, and the proposed HP model.*

0.60) on the curve of cnHP means that cnHP yields 20% more reduced system size than the smallest reduced system size achieved for 60% of the dataset. Therefore, the model closest to the top left corner is interpreted as the model with best performance.

As seen in Figure 7, the proposed model outperforms the baseline algorithms in terms of the reduced system size in the majority of the test instances. As K increases, the performance gap between GP and cnHP decreases, whereas the performance gap between the proposed model and both of the baseline models increases significantly. The proposed model yields the best performance for 69%, 71%, 75%, 82%, 85%, and 86% of the dataset for $K = 8, 16, 32, 64, 128$, and 256, respectively.

The proposed HP model yields very sparse off-diagonal blocks. The number of nonzeros in any lower off-diagonal block R_k is at most 0.51%, 0.44%, 0.35%, 0.26%, 0.19%, and 0.13% of the total nonzero count of A for K=8, 16, 32, 64, 128, and 256 parts on the average, respectively. As the HP model maintains balance on the nonzero counts of the whole row blocks, these low nonzero counts in off-diagonal blocks do not disturb the computational load balance among processors considerably.

5.2. In-block reordering quality. To our knowledge, no in-block reordering method has been proposed or tested for stSpike in the literature. Therefore, we compare the improvement gained by applying the proposed in-block ordering method against a baseline algorithm which does not apply an in-block reordering. In this comparison, both the proposed and the baseline reordering methods utilize the partitions obtained by the HP model (section 4.1). Two quality metrics used in this comparison are total height and nonzero count in the off-diagonal blocks of $\widehat{S}$.

Table 3 shows the ratios of these quality metrics of the in-block reorderings generated by the baseline to those of the proposed method. For each K value, the results are given as averages grouped by different matrix kinds, and the last row shows the average of all instances in the dataset.

As seen in Table 3, the proposed reordering method achieves significant improvement in terms of both quality metrics against the baseline reordering. For example, for $K = 64$, on overall average, the proposed method achieves 39× and 18.7× improvement against the baseline ordering in terms of height and nonzero counts, respectively. The improvement rate attained in height does not always directly reflect to the improvement rate in the nonzero counts since height is an upper bound for fill-in and the fill-in also depends on the sparsity of the diagonal blocks.

Although the improvement of the proposed reordering against the baseline ordering tends to degrade with increasing K, this is expected since there are fewer rows per block and there is less room for improvement. For example, on overall average,

TABLE 3
Total height and nonzero count averages in the off-diagonal blocks of $\widehat{S}$.

Kind ID	K = 8		K = 16		K = 32		K = 64		K = 128		K = 256	
	Height	nnz	Height	nnz	Height	nnz	Height	nnz	Height	nnz	Height	nnz
1	1,470.1	518.5	554.1	233.6	263.6	143.1	115.9	67.6	65.9	41.1	37.2	25.4
2	71.9	125.2	63.1	100.6	30.5	61.6	15.8	35.0	8.8	18.7	5.5	11.6
3	1,219.2	331.4	321.2	271.5	296.8	197.2	167.8	152.7	88.2	82.9	46.1	47.8
4	27.7	3.8	16.8	5.3	9.9	5.7	8.8	6.2	6.5	4.5	4.6	3.1
5	260.0	10.0	142.6	9.1	90.3	7.6	63.5	6.3	37.6	4.7	24.5	4.0
6	600.0	123.2	298.3	101.5	148.6	61.3	73.5	37.0	38.7	22.3	22.0	13.6
7	131.8	10.1	67.3	7.4	36.7	5.7	22.8	4.7	14.0	3.8	8.5	3.2
8	513.5	97.3	260.4	59.4	92.2	30.9	48.9	18.5	23.8	11.3	17.0	8.2
9	1,547.9	1,101.3	1,010.9	1,221.2	556.7	641.8	248.6	315.4	102.0	141.6	50.7	70.0
10	29.0	4.8	32.9	10.9	15.0	5.6	12.8	5.2	8.6	3.6	6.0	2.7
11	375.3	619.3	213.3	213.3	112.8	136.6	68.8	89.0	43.2	54.1	25.6	32.0
12	241.2	170.7	121.4	109.1	59.9	66.8	31.8	41.2	18.1	25.2	10.6	14.7
13	18.2	2.7	17.7	3.1	12.7	2.7	13.4	2.6	10.1	2.6	8.2	2.8
14	217.7	231.1	116.5	149.6	59.2	94.0	33.0	54.4	19.1	30.6	12.2	17.7
15	610.4	228.9	277.9	164.6	122.0	91.6	61.8	50.9	31.5	28.3	18.0	16.4
16	15.8	62.2	8.5	31.5	6.0	18.9	4.4	11.9	3.4	7.8	2.7	5.3
All	238.1	57.2	127.1	43.0	65.0	27.8	39.0	18.7	22.7	12.1	14.3	8.3

*The values are the ratios of the results attained by the baseline over the proposed in-block reordering.

the proposed in-block reordering method achieves 57.2×, 43.0×, 27.8×, 18.7×, 12.1×, and 8.3× decrease in the nonzero count for $K = 8, 16, 32, 64, 128$, and 256, respectively.

The proposed partitioning and reordering model yields very small reduced systems whose nonzero counts are significantly low relative to the original system. The average ratios of the nonzero count of the reduced system over the nonzero count of the original coefficient matrix, i.e., $\text{nnz}(\widehat{S})/\text{nnz}(A)$, are 0.05%, 0.12%, 0.26%, 0.49%, 0.87%, and 1.48% for K=8, 16, 32, 64, 128, and 256 parts, respectively. These low nonzero counts of the reduced systems verify the effectiveness of the proposed partitioning and reordering model in terms of alleviating the sequential computational overhead of dmpGS.

5.3. Parallel scalability. Parallel experiments are performed on the Sariyer cluster of UHEM [65] using up to 320 cores over 8 distributed nodes, each containing 40 cores (two Intel Xeon Gold 6148 CPUs) and 192GB memory. The nodes are connected by an InfiniBand EDR 100 Gbps network.

We implement an MPI+OpenMP hybrid parallel dmpGS to demonstrate the effectiveness of using stSpike and the proposed model. Throughout this section, the proposed model refers to the proposed partitioning and in-block reordering model (section 4) applied to dmpGS. The number of MPI processes is the same as the number of parts (K) in a partition. For dmpGS, we experimented with different configurations of number of processes and threads. We found that the best configuration is 8 processes per node and 5 threads per process. Therefore, we conduct parallel experiments for dmpGS using 1, 2, 4, and 8 nodes corresponding to 40, 80, 160, and 320 cores and $K = 8, 16, 32$, and 64 parts (processes), respectively.

To the best of our knowledge, there is no publicly available true distributed-memory parallel GS implementation. For comparing the performance of dmpGS, we also implemented a multithreaded GS (*mtGS*) by using the multithreaded sparse triangular system solver (mkl_sparse_d_trsm) and sparse matrix-vector multiplicator (mkl_sparse_d_mv) of the Intel Math Kernel Library (MKL) [37]. As a baseline, we obtain the results of mtGS on 40 threads/cores (1 node) by using the GP reordering since it is shown in [22] that the triangular solution with MKL benefits most from GP.

We tested the parallel scalability of dmpGS for a subset of the dataset since we have limited core hours on the high performance computing platform. From the dataset, we considered the matrices with at least 100,000 rows and 10,000,000 nonzeros, for which GS converges with a relative residual of less than 10^{-3} in 500 iterations with initial guess $x=[0,\ldots,0]^T$ and right-hand-side vector $f=[1/m,2/m,\ldots,1]^T$. Then we select only those instances with different sparsity structures from each matrix group. There were exactly 12 such matrices in our dataset satisfying these criteria. The properties of those matrices are shown in Table 4, sorted in decreasing order of nonzero counts. The sixth and the last column, respectively, show the relative residual and runtime of mtGS after 500 iterations.

Table 5 shows the average speedup values obtained by dmpGS with GP, cnHP, and the proposed model over mtGS. We run dmpGS with the proposed model for $\alpha=1,2,5$, and 10 to observe the effect of scaling factor (α) on the parallel performance. As seen in the table, the proposed model achieves significantly higher speedup for dmpGS over the baseline models for all α. The speedup performance gap between the proposed and baseline models increases with increasing K, thus confirming the effectiveness of the proposed model.

We also provide Figure 8 which depicts the performance profiles for comparing the dmpGS runtime using the proposed model for varying α and K values. We choose α=2 for better scalability of dmpGS since it yields the best performance for larger part counts ($K=32$ and 64) as seen in both Table 5 and Figure 8. As seen in Table 5, the proposed model with α=2 yields an average of 1.5×, 1.9×, 2.7×, and 3.2× higher speedup relative to the best of the baseline models for $K=8,16,32$, and 64, respectively.

Figure 9 shows the results of the strong scaling experiments as speedup curves of dmpGS with GP, cnHP, and the proposed model. The proposed model significantly

Table 4
The properties of matrices to run dmpGS.

Matrix	Kind ID	Sym	Size	Nnz	Relative residual*	mtGS* time (s)
msdoor	1	✓	415,863	19,173,163	1.9×10^{-4}	23.1
af_shell1	1	✓	504,855	17,562,051	8.2×10^{-4}	23.4
af_1_k101	1	✓	503,625	17,550,675	1.1×10^{-4}	23.4
CoupCons3D	1		416,800	17,277,420	4.0×10^{-9}	21.8
Freescale1	2		3,428,755	17,052,626	3.0×10^{-4}	72.7
circuit5M_dc	2		3,523,317	14,865,409	1.9×10^{-12}	72.7
CurlCurl_3	9	✓	1,219,574	13,544,618	2.8×10^{-4}	35.4
memchip	2		2,707,524	13,343,948	5.4×10^{-5}	57.5
BenElechi1	6	✓	245,874	13,150,496	6.5×10^{-5}	15.2
pwtk	1	✓	217,918	11,524,432	1.5×10^{-4}	13.6
bmw3_2	1	✓	227,362	11,288,630	1.9×10^{-4}	13.6
bmwcra_1	1	✓	148,770	10,641,602	6.0×10^{-4}	11.9

*Relative residual and runtime results of mtGS on 40 cores for 500 iterations.

Table 5
Average speedup obtained by dmpGS over mtGS on 40 cores.

K	Number of		GP	cnHP	Proposed model			
	nodes	cores			$\alpha=1$	$\alpha=2$	$\alpha=5$	$\alpha=10$
8	1	40	9.87	8.71	**14.85**	14.71	14.77	14.51
16	2	80	14.65	13.58	**29.07**	28.51	28.47	28.25
32	4	160	17.41	16.11	47.28	**47.86**	47.24	45.89
64	8	320	15.79	17.60	54.96	**55.54**	50.21	50.65

*The best speedup value obtained for each K is shown in bold.

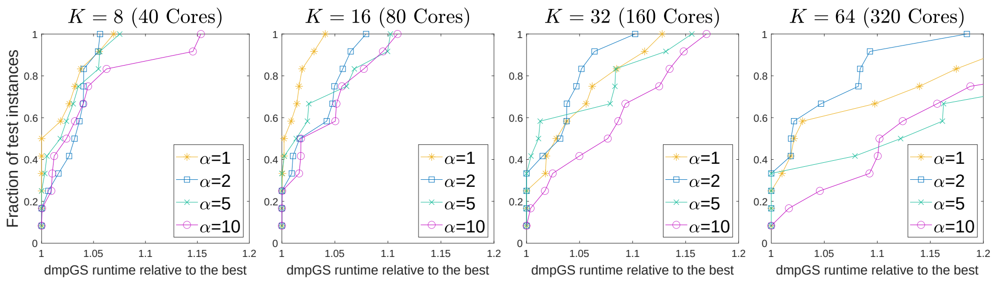


FIG. 8. *Performance profiles in terms of the dmpGS runtime using the proposed model.*

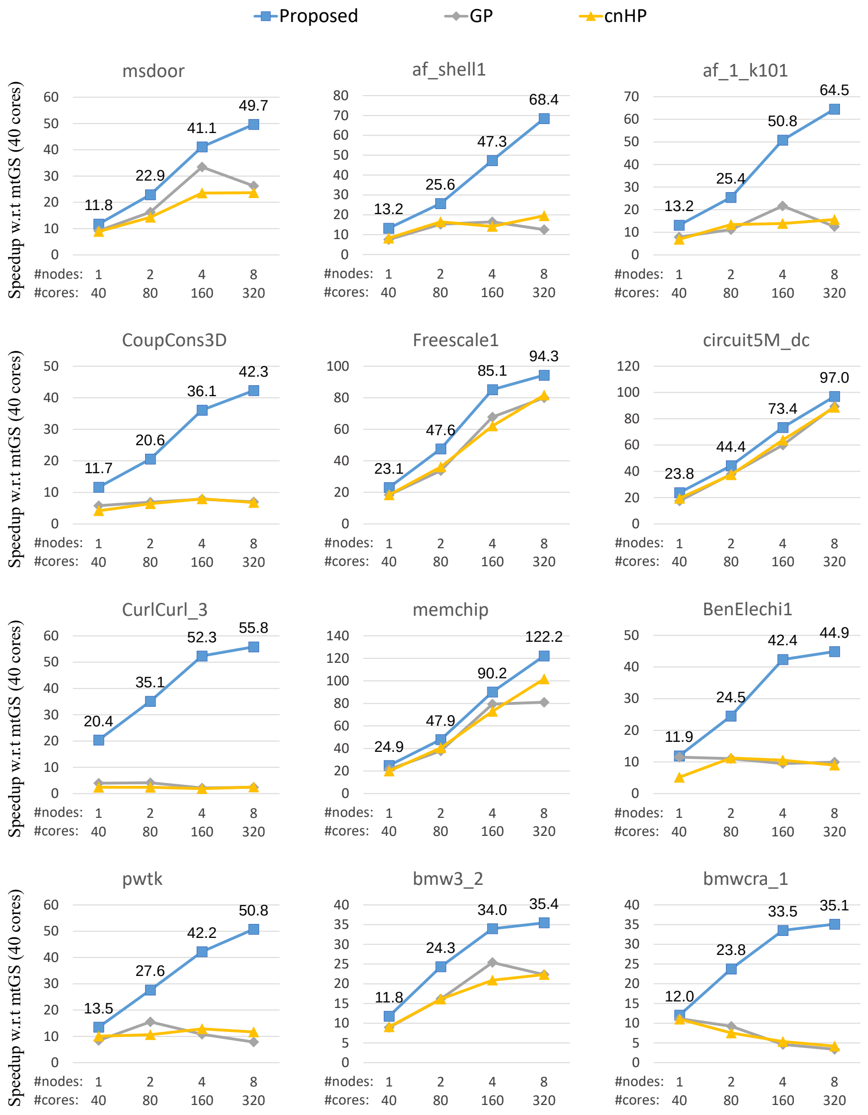


FIG. 9. *Speedup curves of dmpGS with GP, cnHP, and the proposed model (for* $K = 8, 16, 32,$ *and* 64*) relative to mtGS on* 1 *node* (40 *cores).*

enhances the scalability of dmpGS so that dmpGS scales up to 320 cores on all instances. As seen in the figure, the proposed model outperforms GP and cnHP models for all of the test instances, significantly so in 9 out of 12. In Figure 9 for `memchip`, dmpGS using the proposed model achieves up to 122.2 speedup on 320 cores over mtGS on 40 cores.

6. Conclusion. We proposed and implemented an stSpike-based dmpGS algorithm. For improving the scalability of dmpGS, we propose an HP-based partitioning model and an in-block row reordering method. Extensive experiments show that the proposed HP model significantly decreases the reduced system size with respect to the baseline models while attaining comparable communication volume. The proposed in-block reordering method leads to a substantial decrease in the computational cost of both forming and solving the reduced system. Parallel experiments up to 320 cores demonstrate that using the proposed reordering model significantly improves the scalability of dmpGS.

As a future work, we will consider the parallel solution of the reduced system to further alleviate the sequential bottleneck. We will also consider an in-block row reordering which takes the nonzeros of the diagonal blocks into account for further reducing the nonzero count in the reduced system. Finally, the future work will include extending the dmpGS algorithm for multiple right-hand-side vectors as it is very common in modern applications. Using multiple right-hand-side vectors is expected to further enhance the performance of dmpGS since it enables using higher level basic linear algebra subprograms (BLAS) compared to the single right-hand-side case. Moreover, the parallel solution time per right-hand-side vector will further decrease since the parallel factorization is done only once.

REFERENCES

[1] S. Acer, E. Kayaaslan, and C. Aykanat, *A hypergraph partitioning model for profile minimization*, SIAM J. Sci. Comput., 41 (2019), pp. A83–A108.

[2] L. M. Adams and H. F. Jordan, *Is SOR color-blind?*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 490–506.

[3] M. Adams, M. Brezina, J. Hu, and R. Tuminaro, *Parallel multigrid smoothing: Polynomial versus Gauss–Seidel*, J. Comput. Phys., 188 (2003), pp. 593–610.

[4] M. F. Adams, *A distributed memory unstructured Gauss-Seidel algorithm for multigrid smoothers*, in SC'01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, 2001.

[5] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM J. Matrix Anal. Appl., 23 (2001), pp. 15–41.

[6] P. Amodio and F. Mazzia, *A parallel Gauss–Seidel method for block tridiagonal linear systems*, SIAM J. Sci. Comput., 16 (1995), pp. 1451–1461.

[7] R. Bagnara, *A unified proof for the convergence of Jacobi and Gauss–Seidel methods*, SIAM Rev., 37 (1995), pp. 93–97.

[8] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, *Hypergraph partitioning for sparse matrix-matrix multiplication*, ACM Trans. Parallel Comput., 3 (2016), pp. 1–34.

[9] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[10] M. Benzi, *Preconditioning techniques for large linear systems: A survey*, J. Comput. Phys., 182 (2002), pp. 418–477.

[11] M. Benzi and M. Tuma, *A sparse approximate inverse preconditioner for nonsymmetric linear systems*, SIAM J. Sci. Comput., 19 (1998), pp. 968–994.

[12] M. Bernaschi, P. D'Ambra, and D. Pasquini, *AMG based on compatible weighted matching for GPUs*, Parallel Comput., 92 (2020), 102599.

[13] M. W. Berry, *Large-scale sparse singular value computations*, Internat. J. Super Comput. Appl., 6 (1992), pp. 13–49.

[14] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach Using BSP*, Oxford University Press, New York, 2020.

[15] E. S. Bolukbasi and M. Manguoglu, *A multithreaded recursive and nonrecursive parallel sparse direct solver*, in Advances in Computational Fluid-Structure Interaction and Flow Simulation, Springer, Cham, 2016, pp. 283–292.

[16] J. Boyle, M. Mihajlovic, and J. Scott, *HSL MI*20*: An Efficient AMG preconditioner*, Internat. J. Numer. Methods Engrg., 82 (2010), pp. 64–98.

[17] B. Bylina and J. Bylina, *Analysis and comparison of reordering for two factorization methods (LU and WZ) for sparse matrices*, in International Conference on Computational Science, Springer, Cham, 2008, pp. 983–992.

[18] U. V. Catalyurek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Trans. Parallel Distrib. Syst., 10 (1999), pp. 673–693.

[19] Ü. V. Çatalyürek and C. Aykanat, *PaToH (partitioning tool for hypergraphs)*, in Encyclopedia of Parallel Computing, Springer, Cham, 2011, pp. 1479–1487.

[20] Ü. V. Çatalyürek, C. Aykanat, and B. Uçar, *On two-dimensional sparse matrix partitioning: Models, methods, and a recipe*, SIAM J. Sci. Comput., 32 (2010), pp. 656–683.

[21] S.-C. Chen, D. J. Kuck, and A. H. Sameh, *Practical parallel band triangular system solvers*, ACM Trans. Math. Software, 4 (1978), pp. 270–277.

[22] İ. Çuğu and M. Manguoğlu, *A parallel multithreaded sparse triangular linear system solver*, Comput. Math. Appl., 80 (2020), pp. 371–385.

[23] T. A. Davis, *Direct Methods for Sparse Linear Systems*, SIAM, Philadelphia, 2006, pp. 83–95.

[24] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), pp. 1–25.

[25] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, *Parallel hypergraph partitioning for scientific computing*, in Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium, IEEE, 2006, pp. 10–pp.

[26] J. Díaz, J. Petit, and M. Serna, *A survey of graph layout problems*, ACM Comput. Surv., 34 (2002), pp. 313–356.

[27] E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, Math. Program., 91 (2002), pp. 201–213.

[28] J. J. Dongarra and A. H. Sameh, *On some parallel banded system solvers*, Parallel Comput., 1 (1984), pp. 223–235.

[29] I. S. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 973–996.

[30] H. C. Elman, D. J. Silvester, and A. J. Wathen, *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics*, Numer. Math. Sci. Comput., Oxford University Press, Oxford, 2014.

[31] G. C. Fox, *Solving Problems on Concurrent Processors*, Prentice Hall, Old Tappan, NJ, 1988.

[32] M. Gee, C. Siefert, J. Hu, R. Tuminaro, and M. Sala, *ML* 5.0 *Smoothed Aggregation User's Guide*, Tech. Report SAND2006-2649, Sandia National Laboratories, 2006.

[33] G. Golub and J. M. Ortega, *Scientific Computing: An Introduction with Parallel Computing*, Academic Press, New York, 1993.

[34] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed., John Hopkins University Press, Baltimore, MD, 2013, pp. 606–616.

[35] L. Grigori, S. Moufawad, and F. Nataf, *Enlarged Krylov subspace conjugate gradient methods for reducing communication*, SIAM J. Matrix Anal. Appl., 37 (2016), pp. 744–773.

[36] V. E. Henson and U. M. Yang, *BoomerAMG: A parallel algebraic multigrid solver and preconditioner*, Appl. Numer. Math., 41 (2002), pp. 155–177.

[37] Intel, *Intel Math Kernel Library (MKL)*, 2019, https://software.intel.com/en-us/mkl.

[38] K. S. Kang, *Scalable implementation of the parallel multigrid method on massively parallel computers*, Comput. Math. Appl., 70 (2015), pp. 2701–2708.

[39] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, *Multilevel hypergraph partitioning: Applications in VLSI domain*, IEEE Trans. Very Large Scale Integration Syst., 7 (1999), pp. 69–79.

[40] G. Karypis and V. Kumar, *METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version* 5.1, 2013, http://www.cs.umn.edu/~metis.

[41] M. Kawai, T. Iwashita, H. Nakashima, and O. Marques, *Parallel smoother based on block red-black ordering for multigrid Poisson solver*, in International Conference on High Performance Computing for Computational Science, Springer, Cham, 2012, pp. 292–299.

[42] D. P. Koester, S. Ranka, and G. C. Fox, *A parallel Gauss-Seidel algorithm for sparse power system matrices*, in Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing, IEEE, 1994, pp. 184–193.

[43] X. S. Li and J. W. Demmel, *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Software, 29 (2003), pp. 110–140.

[44] Y. Lin and J. Yuan, *Profile minimization problem for matrices and graphs*, Acta Math. Appl. Sin., 10 (1994), pp. 107–112.

[45] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, *A synchronization-free algorithm for parallel sparse triangular solves*, in European Conference on Parallel Processing, Springer, Cham, 2016, pp. 617–630.

[46] L. O. Mafteiu-Scai, *The bandwidths of a matrix. A survey of algorithms*, An. Univ. Vest Timiş. Ser. Mat.-Inform., 52 (2014), pp. 183–223.

[47] M. Manguoglu, *A domain-decomposing parallel sparse linear system solver*, J. Comput. Appl. Math., 236 (2011), pp. 319–325.

[48] M. Manguoglu, *Parallel solution of sparse linear systems*, in High-Performance Scientific Computing, Springer, Cham, 2012, pp. 171–184.

[49] M. Manguoglu, A. H. Sameh, and O. Schenk, *PSPIKE: A parallel hybrid sparse linear system solver*, in Proceedings of the European Conference on Parallel Processing, Springer, 2009, pp. 797–808.

[50] C. C. K. Mikkelsen and M. Manguoglu, *Analysis of the truncated SPIKE algorithm*, SIAM J. Matrix Anal. Appl., 30 (2009), pp. 1500–1519.

[51] Y. Notay, *An aggregation-based algebraic multigrid method*, Electron. Trans. Numer. Anal., 37 (2010), pp. 123–146.

[52] D. P. O'Leary, *Ordering schemes for parallel processing of certain mesh problems*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 620–632.

[53] W. Pazner, *Efficient low-order refined preconditioners for high-order matrix-free continuous and discontinuous Galerkin methods*, SIAM J. Sci. Comput., 42 (2020), pp. A3055–A3083.

[54] E. Polizzi and A. Sameh, *A parallel hybrid banded system solver: The SPIKE algorithm*, Parallel Comput., 32 (2006), pp. 177–194.

[55] E. Polizzi and A. Sameh, *SPIKE: A parallel environment for solving banded linear systems*, Comput. & Fluids, 36 (2007), pp. 113–120.

[56] J. W. Ruge and K. Stüben, *Algebraic multigrid*, in Multigrid Methods, SIAM, Philadelphia, 1987, pp. 73–130.

[57] Y. Saad, *ILUT: A dual threshold incomplete LU factorization*, Numer. Linear Algebra Appl., 1 (1994), pp. 387–402.

[58] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, 2003.

[59] A. H. Sameh and R. P. Brent, *Solving triangular systems on a parallel computer*, SIAM J. Numer. Anal., 14 (1977), pp. 1101–1113.

[60] O. Schenk, M. Manguoglu, A. Sameh, M. Christen, and M. Sathe, *Parallel scalable PDE-constrained optimization: Antenna identification in hyperthermia cancer treatment planning*, Comput. Sci. Res. Dev., 23 (2009), pp. 177–183.

[61] Y. Shang, *A distributed memory parallel Gauss–Seidel algorithm for linear algebraic systems*, Comput. Math. Appl., 57 (2009), pp. 1369–1376.

[62] B. S. Spring, E. Polizzi, and A. H. Sameh, *A feature-complete SPIKE dense banded solver*, ACM Trans. Math. Software, 46 (2020), pp. 1–35.

[63] R. Tavakoli and P. Davami, *A new parallel Gauss–Seidel method based on alternating group explicit method and domain decomposition method*, Appl. Math. Comput., 188 (2007), pp. 713–719.

[64] B. Uçar and C. Aykanat, *Revisiting hypergraph models for sparse matrix partitioning*, SIAM Rev., 49 (2007), pp. 595–603.

[65] UHEM, *National Center for High Performance Computing*, 2021, http://www.uhem.itu.edu.tr.

[66] P. Van, M. Brezina, and J. Mandel, *Convergence of algebraic multigrid based on smoothed aggregation*, Numer. Math., 88 (2001), pp. 559–579.

[67] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Rev., 47 (2005), pp. 67–95.

[68] I. E. Venetis, A. Kouris, A. Sobczyk, E. Gallopoulos, and A. H. Sameh, *A direct tridiagonal solver based on Givens rotations for GPU architectures*, Parallel Comput., 49 (2015), pp. 101–116.

[69] K. Watanabe, H. Igarashi, and T. Honma, *Comparison of geometric and algebraic multigrid methods in edge-based finite-element analysis*, IEEE Trans. Magn., 41 (2005), pp. 1672–1675.

[70] R. Webster, *An algebraic multigrid solver for Navier-Stokes problems*, Internat. J. Numer. Methods Fluids, 18 (1994), pp. 761–780.

[71] U. M. Yang, *Parallel algebraic multigrid methods-high performance preconditioners*, in Numerical Solution of Partial Differential Equations on Parallel Computers, Springer, Cham, 2006, pp. 209–236.

[72] J. Zhang, *Acceleration of five-point red-black Gauss-Seidel in multigrid for Poisson equation*, Appl. Math. Comput., 80 (1996), pp. 73–93.