SIAM J. Sci. Comput.
Vol. 32, No. 2, pp. 656–683

© 2010 Society for Industrial and Applied Mathematics

ON TWO-DIMENSIONAL SPARSE MATRIX PARTITIONING: MODELS, METHODS, AND A RECIPE*

ÜMİT V. ÇATALYÜREK†, CEVDET AYKANAT‡, AND BORA UÇAR§

Abstract. We consider two-dimensional partitioning of general sparse matrices for parallel sparse matrix-vector multiply operation. We present three hypergraph-partitioning-based methods, each having unique advantages. The first one treats the nonzeros of the matrix individually and hence produces fine-grain partitions. The other two produce coarser partitions, where one of them imposes a limit on the number of messages sent and received by a single processor, and the other trades that limit for a lower communication volume. We also present a thorough experimental evaluation of the proposed two-dimensional partitioning methods together with the hypergraph-based one-dimensional partitioning methods, using an extensive set of public domain matrices. Furthermore, for the users of these partitioning methods, we present a partitioning recipe that chooses one of the partitioning methods according to some matrix characteristics.

Key words. sparse matrix partitioning, parallel matrix-vector multiplication, hypergraph partitioning, two-dimensional partitioning, combinatorial scientific computing

AMS subject classifications. 05C50, 05C65, 65F10, 65F50, 65Y05

DOI. 10.1137/080737770

1. Introduction.

Sparse matrix-vector multiply operation forms the computational core of many iterative methods including solvers for linear systems, linear programs, eigensystems, and least squares problems. In these solvers, the computations $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$ are performed repeatedly with the same large, sparse, possibly unsymmetric or rectangular matrix $\mathbf{A}$ and with a changing input vector $\mathbf{x}$. Our aim is to efficiently parallelize these multiply operations by two-dimensional (2D) partitioning of the matrix $\mathbf{A}$ in such a way that the computational load per processor is balanced and the communication overhead is low.

Graph and hypergraph partitioning models have been used for one-dimensional (1D) partitioning of sparse matrices [4, 5, 8, 9, 19, 20, 25, 26, 30, 32, 37]. In these models, a K-way partition of the vertices of a given graph or hypergraph is computed. The partitioning constraint is to maintain a balance criterion on the number of vertices in each part; if the vertices are weighted, then the constraint is to maintain a balance criterion on the sum of the vertex weights in each part. The partitioning objective is to minimize the cutsize of the partition defined over the edges or hyperedges. The partitioning constraint and objective relate, respectively to, maintaining a computational

*Received by the editors October 10, 2008; accepted for publication (in revised form) October 21, 2009; published electronically February 24, 2010.
http://www.siam.org/journals/sisc/32-2/73777.html

†Departments of Biomedical Informatics and Electrical & Computer Engineering, The Ohio State University, Columbus, OH 43210 (catalyurek.1@osu.edu). The work of this author is supported in part by the National Science Foundation grants CNS-0643969, OCI-0904809, OCI-0904802, and CNS-0403342, and the U.S. DOE SciDAC Institute grant DE-FC02-06ER2775.

‡Computer Engineering Department, Bilkent University, Ankara, Turkey (aykanat@cs.bilkent.edu.tr). The work of this author is partially supported by The Scientific and Technical Research Council of Turkey (TÜBİTAK) under project EEEAG-109E019.

§Centre National de la Recherche Scientifique, Laboratoire de l'Informatique du Parallélisme, (UMR CNRS -ENS Lyon-INRIA-UCBL), Université de Lyon, 46, allée d'Italie, ENS Lyon, F-69364, Lyon Cedex 7, France (bora.ucar@ens-lyon.fr). The work of this author is partially supported by Agence Nationale de la Recherche through SOLSTICE project ANR-06-CIS6-010.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

load balance and minimizing the total communication volume. The limitations of the graph model have been shown in [8, 9, 19]. First, it tries to minimize a wrong objective function, since edge-cut metric is only an approximation to the total communication volume. Second, it can only model square matrices. Alternative models such as bipartite graph model [21], multi-constraint and multi-objective partitionings [43, 44], skewed partitioning [23], and those based on hypergraph partitioning [8, 9] have been proposed. All these new models address some of the limitations of the standard model, but only in hypergraph-partitioning based ones, the partitioning objective is an exact measure of the total communication volume.

Earlier works on 2D matrix partitioning [22, 34, 35, 39] are based on checkerboard partitioning. These works are typically suited to dense matrices or sparse matrices with structured nonzero patterns that are difficult to exploit. Later works [7, 11, 12, 52] are specifically targeted to sparse matrices. These works are based on different hypergraph models and produce matrix partitionings with differing characteristics. These 2D partitioning models, as hypergraph-partitioning based models for 1D partitioning, encode the total communication volume exactly with the partitioning objective. The hypergraph model in [11] is used to partition the matrices on nonzero basis. In other words, it produces fine-grain partitionings in which assignment decisions are made in nonzero basis. The hypergraph model in [12] is used to obtain checkerboard partitionings. In other words, the matrix is divided into blocks, and the blocks are assigned to processors. The partitioned matrix maps naturally onto a 2D mesh of processors. Therefore, the communication along a matrix row or column is confined to a subset of processors, and hence the total number of messages is limited. The approach presented in [52] applies recursive bisection in which each step partitions the current submatrix along the rows or columns using the hypergraph models for 1D partitioning. This approach does not limit the total number of messages.

In order to parallelize the matrix-vector multiply $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$, we have to partition the vectors $\mathbf{x}$ and $\mathbf{y}$ as well. There are two alternatives in partitioning the vectors $\mathbf{x}$ and $\mathbf{y}$. The first one, symmetric partitioning, is to have the same partition on $\mathbf{x}$ and $\mathbf{y}$. The second one, nonsymmetric partitioning, is to have different partitions on $\mathbf{x}$ and $\mathbf{y}$. There are three groups of methods to obtain vector partitionings. The methods in the first group perform the vector partitioning implicitly using the partitions on the matrix for symmetric partitioning [9, 11, 12]. The methods in the second group perform the vector partitioning in an additional stage after partitioning the matrix for nonsymmetric [3, 46, 47, 52] and symmetric [46, 52] partitionings. The methods in the third group [50] enhance the previously proposed hypergraph models in order to obtain vector and matrix partitionings simultaneously both for the symmetric and nonsymmetric partitioning cases. A common goal pursued by all these techniques is to assign a vector entry to a processor that has nonzeros in the corresponding row or column of the matrix. In this paper, we are only interested in matrix partitioning, and we do not make use of any of those vector partitioning methods. However, we use a simple vector partitioning method achieving the common goal stated above.

We present some background material on parallel matrix-vector multiply operation based on 2D matrix partitioning, hypergraph partitioning, and hypergraph models for 1D matrix partitioning in the next section. Section 3 presents three methods with different assignment granularity and communication patterns for 2D matrix partitioning: *fine-grain*, *jagged-like*, and *checkerboard-like* partitioning methods. The fine-grain and the checkerboard-like models were briefly discussed, respectively, in [11] and [12]. The jagged-like partitioning model was only described in the first author's thesis [7]. Section 4 contains further investigations on partitioning methods, including

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

a recipe on matrix partitioning alternatives. In section 5, we present experimental results.

Our contributions in this paper are fourfold: first, to present the jagged-like method for the first time, and the fine-grain and checkerboard-like methods in a more accessible venue (section 3); second, to investigate the merits of these three partitioning approaches with respect to each other (section 4.1); third, to propose a recipe (section 4.4) which suggests a partitioning method among the existing 1D and the proposed 2D partitioning methods based on some easily computable matrix characteristics; fourth, a thorough and conclusive experimental evaluation (section 5) of the 1D and 2D partitioning methods as well as the effectiveness of the proposed recipe. We also discuss (sections 4.2 and 4.3) how communication requirements can be modeled when collective communication primitives are used in the matrix-vector multiply operations and characterize a class of applications whose efficient parallelization can be obtained by using hypergraph partitioning models.

2. Preliminaries.

Here, we give an overview of algorithms for parallel matrix-vector multiplies, hypergraph partitioning problem and its variations, and remind the reader of the hypergraph models for 1D sparse matrix partitioning.

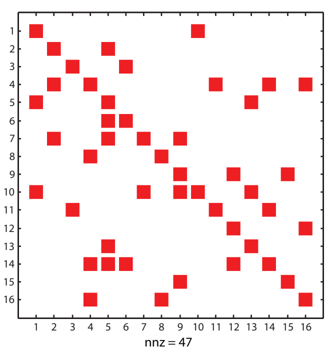
2.1. Row-column-parallel matrix-vector multiply.

Consider the computations $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}$, where the nonzeros of the $M \times N$ matrix $\mathbf{A}$ are partitioned arbitrarily among K processors such that each processor P_k owns a mutually disjoint subset of nonzeros, $\mathbf{A}^{(k)}$. Then, $\mathbf{A}$ can be written as $\mathbf{A} = \sum_k \mathbf{A}^{(k)}$. The vectors $\mathbf{y}$ and $\mathbf{x}$ are also partitioned among processors, where the processor P_k holds $\mathbf{x}^{(k)}$, a dense vector of size N_k, and it is responsible for computing $\mathbf{y}^{(k)}$, a dense vector of size M_k. We note that the vectors $\mathbf{x}^{(k)}$ for $k = 1, \ldots, K$ are disjoint and hence $\sum_k N_k = N$; similarly the vectors $\mathbf{y}^{(k)}$ for $k = 1, \ldots, K$ are disjoint and hence $\sum_k M_k = M$. In this setting, the sparse matrix $\mathbf{A}^{(k)}$ owned by processor P_k can be permuted and written as

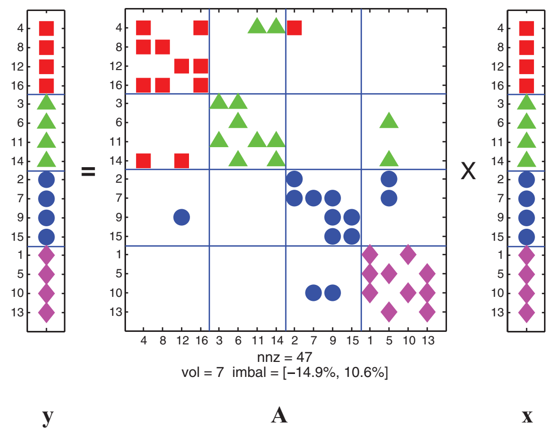
$$\mathbf{A}^{(k)} = \begin{bmatrix} \mathbf{A}_{11}^{(k)} & \cdots & \mathbf{A}_{1\ell}^{(k)} & \cdots & \mathbf{A}_{1K}^{(k)} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{A}_{\ell 1}^{(k)} & \cdots & \mathbf{A}_{\ell\ell}^{(k)} & \cdots & \mathbf{A}_{\ell K}^{(k)} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{A}_{K1}^{(k)} & \cdots & \mathbf{A}_{K\ell}^{(k)} & \cdots & \mathbf{A}_{KK}^{(k)} \end{bmatrix}. \tag{2.1}$$

Here, the blocks in the row-block stripe $\mathbf{A}_{k*}^{(k)} = \{\mathbf{A}_{k1}^{(k)}, \ldots, \mathbf{A}_{kk}^{(k)}, \ldots, \mathbf{A}_{kK}^{(k)}\}$ have row dimension of M_k, and similarly the blocks in the column-block stripe $\mathbf{A}_{*k}^{(k)} = \{\mathbf{A}_{1k}^{(k)}, \ldots, \mathbf{A}_{kk}^{(k)}, \ldots, \mathbf{A}_{Kk}^{(k)}\}$ have column dimension of N_k. The $\mathbf{x}$-vector entries that are needed by processor P_k are represented as $\hat{\mathbf{x}}^{(k)} = [\hat{\mathbf{x}}_1^{(k)}, \ldots, \hat{\mathbf{x}}_k^{(k)}, \ldots, \hat{\mathbf{x}}_K^{(k)}]$, a sparse column vector (we omit the transpose sign for column vectors for simplicity in the notation), where $\hat{\mathbf{x}}_\ell^{(k)}$ contains only those entries of $\mathbf{x}^{(\ell)}$ of processor P_ℓ corresponding to the nonzero columns in $\mathbf{A}_{*\ell}^{(k)}$. Here, the vector $\hat{\mathbf{x}}_k^{(k)}$ is equivalent to $\mathbf{x}^{(k)}$, defined according to the given partition on the $\mathbf{x}$-vector (hence the vector $\hat{\mathbf{x}}^{(k)}$ is of size at least N_k). The $\mathbf{y}$-vector entries for which the processor P_k computes partial results are represented as a sparse vector $\hat{\mathbf{y}}^{(k)} = [\hat{\mathbf{y}}_k^{(1)}, \ldots, \hat{\mathbf{y}}_k^{(k)}, \ldots, \hat{\mathbf{y}}_k^{(K)}]$, where $\hat{\mathbf{y}}_k^{(\ell)}$ contains only the partial results for $\mathbf{y}^{(\ell)}$ corresponding to the nonzero rows in $\mathbf{A}_{\ell *}^{(k)}$. Since the parallelism is achieved on a nonzero basis, we derive a nonzero-based sparse matrix-vector multiply (SpMxV) algorithm. This algorithm, which we call the *row-column-parallel* algorithm, executes at each processor P_k the steps that follow.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.



(a)



(b)

FIG. 2.1. (a) *A* 16×16 *unsymmetric matrix* **A** *with* `nnz` = 47 *nonzeros.* (b) *Sparse matrix-vector multiplication* $\mathbf{y} \leftarrow \mathbf{Ax}$ *of the sample matrix* **A**. *The matrix and the input and output vectors are partitioned among four processors. The four disjoint sets of nonzeros and vector entries that are assigned to the four processors are shown with four distinct shapes and colors. The average number of nonzeros per processor is* 47/4 = 11.75. *The maximum number of nonzeros of a processor is* 13, *giving an imbalance ratio of* 10.6%; *i.e., the maximally loaded processor has* 10.6% *more nonzeros than the average number of nonzeros. The minimum number of nonzeros of a processor is* 10, *being* 14.9% *less than the average number of nonzeros. We indicate the imbalance among the parts,* `imbal`, *using these two marginal percentages. The total communication volume is denoted with* `vol`.

1. For each $\ell \neq k$, form and send sparse vector $\hat{\mathbf{x}}_k^{(\ell)}$ to processor P_ℓ, where $\hat{\mathbf{x}}_k^{(\ell)}$ contains only those entries of $\mathbf{x}^{(k)}$ corresponding to the nonzero columns in $\mathbf{A}_{*k}^{(\ell)}$.
2. In order to form $\hat{\mathbf{x}}^{(k)} = [\hat{\mathbf{x}}_1^{(k)}, \dots, \hat{\mathbf{x}}_k^{(k)}, \dots, \hat{\mathbf{x}}_K^{(k)}]$, first define $\hat{\mathbf{x}}_k^{(k)} = \mathbf{x}^{(k)}$. Then, for each $\ell \neq k$ where $\mathbf{A}_{*\ell}^{(k)}$ contains nonzeros, receive $\hat{\mathbf{x}}_\ell^{(k)}$ from processor P_ℓ, corresponding to the nonzero columns in $\mathbf{A}_{*\ell}^{(k)}$.
3. Compute $\hat{\mathbf{y}}^{(k)} \leftarrow \mathbf{A}^{(k)}\hat{\mathbf{x}}^{(k)}$.
4. For each $\ell \neq k$, send the sparse partial-results vector $\hat{\mathbf{y}}_k^{(\ell)}$ to processor P_ℓ, where $\hat{\mathbf{y}}_k^{(\ell)}$ contains only those partial results for $\mathbf{y}^{(\ell)}$ corresponding to the nonzero rows in $\mathbf{A}_{\ell *}^{(k)}$.
5. Receive the partial-results vector $\hat{\mathbf{y}}_\ell^{(k)}$ from each processor P_ℓ which has computed a partial result for $\mathbf{y}^{(k)}$, i.e., from each processor P_ℓ where $\mathbf{A}_{k*}^{(\ell)}$ has nonzeros.
6. Compute $\mathbf{y}^{(k)} \leftarrow \sum_\ell \hat{\mathbf{y}}_\ell^{(k)}$, adding all the partial-results $\hat{\mathbf{y}}_\ell^{(k)}$ received in the previous step to its own partial results for $\mathbf{y}^{(k)}$.

There are two communication phases in this algorithm. The first one is just before the local matrix-vector multiply, and it is due to the communication of the **x**-vector entries (steps 1 and 2). We refer to this operation as *expand*. The second communication phase is just after the local matrix-vector multiply, and it is due to the communication of the partial results on **y**-vector entries (steps 4 and 5). We refer to this operation as *fold*. It is possible to restructure this algorithm in order to take full advantage of communication and computation overlap [48].

Figure 2.1 shows a sample matrix and input- and output-vectors of a matrix-vector multiply operation, partitioned among four processors. The matrix is permuted

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

such that the rows and the columns of the matrix are aligned conformably with the partition on the output- and input-vectors, respectively. The disjoint sets of nonzeros $\mathbf{A}^{(1)}$ to $\mathbf{A}^{(4)}$ are assigned to the processors P_1 to P_4 and each such set is shown with a distinct symbol in the figure. Processor P_1 holds the (red) squares; processor P_2 holds the (green) triangles; processor P_3 holds the (blue) circles; and processor P_4 holds the (magenta) diamonds. The number of nonzeros of the four processors are, respectively, 12, 12, 13, and 10. The average number of nonzeros is $47/4 = 11.75$, the maximum is 13, being 10.6% more than the average, and the minimum is 10, being 14.9% less than the average. Consider the processor P_1, to see the steps of the multiply algorithm and the communication operations performed by P_1. Among all blocks of $\mathbf{A}^{(1)}$, only three are nonempty: $\mathbf{A}_{11}^{(1)}$, containing the nine nonzeros of the $(1,1)$-block in the figure; $\mathbf{A}_{21}^{(1)}$, containing the two nonzeros of the $(2,1)$-block in the figure; and $\mathbf{A}_{13}^{(1)}$ containing the one nonzero of the $(1,3)$-block. Processor P_1 holds the vector $\mathbf{x}^{(1)} = [x_4, x_8, x_{12}, x_{16}]$ and has to compute the final result of the multiplication for $\mathbf{y}^{(1)} = [y_4, y_8, y_{12}, y_{16}]$. It needs the $\mathbf{x}$-vector entries $\hat{\mathbf{x}}^{(1)} = [\hat{\mathbf{x}}_1^{(1)}, \hat{\mathbf{x}}_3^{(1)}]$, where $\hat{\mathbf{x}}_1^{(1)} = \mathbf{x}^{(1)}$ and $\hat{\mathbf{x}}_3^{(1)}$ contain only those entries of $\mathbf{x}^{(3)}$ of processor P_3 corresponding to the nonzero columns in $\mathbf{A}_{*3}^{(3)}$, i.e., $\hat{\mathbf{x}}_3^{(1)} = [x_2]$. During the course of the multiply operation, P_1 sends $\hat{\mathbf{x}}_1^{(3)} = [x_{12}]$ to processor P_3 in step 1; receives $\hat{\mathbf{x}}_3^{(1)} = [x_2]$ from P_3 to form $\hat{\mathbf{x}}^{(1)}$ in step 2; performs the multiplication operations in step 3; sends the partial-result vector $\hat{\mathbf{y}}_1^{(2)}$ for y_{14} to processor P_2 in step 4; receives partial result for y_4 from processor P_2 in step 5; and finally adds up the partial result received in the previous step to its own results to compute $\mathbf{y}^{(1)} = [y_4, y_8, y_{12}, y_{16}]$.

It is implicit in the algorithm that row *coherence* and column *coherence* are important factors in a matrix partition for parallel SpMxV. Column coherence relates to the fact that nonzeros on the same column require the same $\mathbf{x}$-vector entry. Row coherence relates to the fact that nonzeros on the same row generate partial results for the same $\mathbf{y}$-vector entry. In a partitioning, disturbing column coherence incurs expand communication of $\mathbf{x}$-vector entries, and disturbing row coherence incurs fold communication of partial $\mathbf{y}$-vector results.

If the sparsity structure of $\mathbf{A}$ is ignored, in the worst case, the total communication volume of the nonzero-based parallel matrix-vector multiply algorithm is $(K-1)N + (K-1)M$ units, and the total number of messages is $2K(K-1)$. The worst case occurs when there is at least one nonzero in every single row and every single column of $\mathbf{A}^{(k)}$ for all k. By restricting the partitioning of nonzeros to 1D, i.e., partitioning such that only row-block stripe $\mathbf{A}_{k*}^{(k)}$ (or column-block stripe $\mathbf{A}_{*k}^{(k)}$) would have all of the nonzeros in $\mathbf{A}^{(k)}$, one can reduce the worst-case communication requirements to $K(K-1)$ messages with a total volume of $(K-1)N$, or $(K-1)M$ units. By further restricting the partitioning such that only a subset of blocks in row-block stripe $\mathbf{A}_{k*}^{(k)}$ and column-block stripe $\mathbf{A}_{*k}^{(k)}$ have nonzeros, it is also possible to achieve a 2D distribution [22, 34, 35, 39], called *transpose-free blocked 2D partitioning*, that would reduce the worst-case communication requirements to $2K(\sqrt{K}-1)$ messages with a total volume of $(\sqrt{K}-1)N + (\sqrt{K}-1)M$ units.

2.2. Hypergraph partitioning.

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$. Every net $n_j \in \mathcal{N}$ is a subset of vertices, i.e., $n_j \subseteq \mathcal{V}$. The vertices in a net n_j are called its *pins*. The number of pins of a net defines its size. Weights can be associated with the vertices. We use w_i to denote the weight of the vertex v_i.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

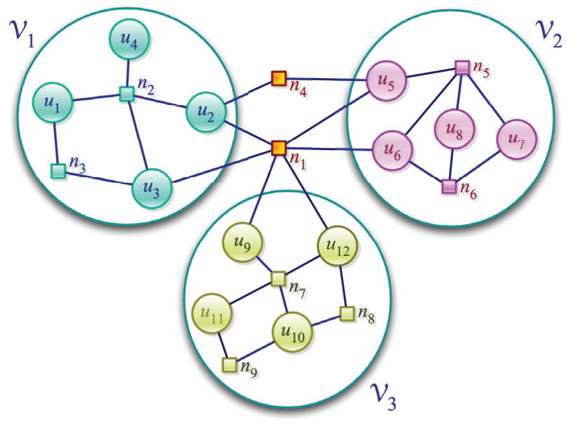


FIG. 2.2. *A hypergraph and a partition on its vertices. The vertices are labeled from u_1 to u_{12} and are represented by circles. The nets are labeled from n_1 to n_9 and are represented by the small squares. The vertices are partitioned into three parts, and the parts are labeled as $\mathcal{V}_1, \mathcal{V}_2$, and $\mathcal{V}_3$.*

Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ is called a K-way partition of the vertex set $\mathcal{V}$ if each part is nonempty, i.e., $\mathcal{V}_k \neq \emptyset$ for $1 \leq k \leq K$; parts are pairwise disjoint, i.e., $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for $1 \leq k < \ell \leq K$; and the union of parts gives $\mathcal{V}$, i.e., $\bigcup_k \mathcal{V}_k = \mathcal{V}$. A K-way vertex partition of $\mathcal{H}$ is said to satisfy the partitioning constraint if

$$W_k \leq W_{avg}(1+\varepsilon) \quad \text{for } k = 1, 2, \ldots, K. \tag{2.2}$$

In (2.2), the weight W_k of a part $\mathcal{V}_k$ is defined as the sum of the weights of the vertices in that part (i.e., $W_k = \sum_{v_i \in \mathcal{V}_k} w_i$), W_{avg} is the average part weight (i.e., $W_{avg} = (\sum_{v_i \in \mathcal{V}} w_i)/K$), and ε represents the allowable imbalance ratio.

In a partition Π of $\mathcal{H}$, a net that has at least one pin (vertex) in a part is said to *connect* that part. *Connectivity set* Λ_j of a net n_j is defined as the set of parts connected by n_j. *Connectivity* $\lambda_j = |\Lambda_j|$ of a net n_j denotes the number of parts connected by n_j. A net n_j is said to be *cut* (*external*) if it connects more than one part (i.e., $\lambda_j > 1$), and *uncut* (*internal*) otherwise (i.e., $\lambda_j = 1$). The set of external nets of a partition Π is denoted as $\mathcal{N}_E$. The partitioning objective is to minimize the cutsize defined over the cut nets. There are various cutsize definitions. Two relevant definitions are:

$$cutsize(\Pi) = \sum_{n_j \in \mathcal{N}_E} 1, \tag{2.3}$$

$$cutsize(\Pi) = \sum_{n_j \in \mathcal{N}_E} (\lambda_j - 1). \tag{2.4}$$

In (2.3), each cut net contributes one to the cutsize. In (2.4), each cut net n_j contributes $\lambda_j - 1$ to the cutsize. If costs are associated with the nets, then a cut net contributes its cost multiples of the above quantities to the cutsize. The hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more parts such that the cutsize is minimized, while a given balance criterion (2.2) is met. The hypergraph partitioning problem is known to be NP-hard [33].

Figure 2.2 shows a sample hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ with 12 vertices and 9 nets. The vertices are labeled from u_1 to u_{12} and are represented by circles. The nets are

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

labeled from n_1 to n_9 and are represented by the small squares. The pins are shown with lines. For example, net n_2 contains vertices u_1 to u_4. The vertices are partitioned into three parts, each shown by a large cycle encompassing the vertices in that part and labeled as $\mathcal{V}_1, \mathcal{V}_2$, and $\mathcal{V}_3$. The nets n_1 and n_4 connect, respectively, three and two parts, and hence they are in the cut: the other nets are internal to a part. The cutsize according to (2.3) is 2, as there are two nets in the cut, whereas the cutsize according to (2.4) is 3, where n_1 and n_4 contribute, respectively, 2 and 1. Assuming vertices of unit weights, the partition has a perfect balance.

A recent variant of the above problem is the multi-constraint hypergraph partitioning [1, 7, 12, 27, 44] in which each vertex has a vector of weights associated with it. The partitioning objective is the same as above, and the partitioning constraint is to satisfy a balancing constraint associated with each weight. We use the notation $w_{i,g}$ to denote the G weights of a vertex v_i for $g = 1, \ldots, G$. Hence, the balance criterion (2.2) can be rewritten as

$$W_{k,g} \leq W_{avg,g}\,(1+\varepsilon) \text{ for } k = 1, \ldots, K \text{ and } \; g = 1, \ldots, G, \tag{2.5}$$

where the gth weight $W_{k,g}$ of a part $\mathcal{V}_k$ is defined as the sum of the gth weights of the vertices in that part (i.e., $W_{k,g} = \sum_{v_i \in \mathcal{V}_k} w_{i,g}$), and $W_{avg,g}$ is the average part weight for the gth weight (i.e., $W_{avg,g} = (\sum_{v_i \in \mathcal{V}} w_{i,g})/K$), and ε again represents allowed imbalance ratio.

2.3. Hypergraph models for 1D sparse matrix partitioning. In the *column-net hypergraph model* [8, 9] $\mathcal{H}_\mathcal{R} = (\mathcal{V}_\mathcal{R}, \mathcal{N}_\mathcal{C})$ of matrix $\mathbf{A}$, there exist one vertex $v_i \in \mathcal{V}_\mathcal{R}$ and one net $n_j \in \mathcal{N}_\mathcal{C}$ for each row r_i and column c_j, respectively. Net $n_j \subseteq \mathcal{V}_\mathcal{R}$ contains the vertices corresponding to the rows that have a nonzero entry in column c_j. That is, $v_i \in n_j$ if and only if $a_{ij} \neq 0$. Weight w_i of a vertex $v_i \in \mathcal{V}_\mathcal{R}$ is set to the total number of nonzeros in row r_i. This model is used for rowwise partitioning.

In the *row-net hypergraph model* [8, 9] $\mathcal{H}_\mathcal{C} = (\mathcal{V}_\mathcal{C}, \mathcal{N}_\mathcal{R})$ of matrix $\mathbf{A}$, there exist one vertex $v_j \in \mathcal{V}_\mathcal{C}$ and one net $n_i \in \mathcal{N}_\mathcal{R}$ for each column c_j and row r_i, respectively. Net $n_i \subseteq \mathcal{V}_\mathcal{C}$ contains the vertices corresponding to the columns that have a nonzero entry in row r_i. That is, $v_j \in n_i$ if and only if $a_{ij} \neq 0$. Weight w_j of a vertex $v_j \in \mathcal{V}_\mathcal{R}$ is set to the total number of nonzeros in column c_j. This model is used for columnwise partitioning.

The use of the hypergraphs $\mathcal{H}_\mathcal{R}$ and $\mathcal{H}_C$ in 1D sparse matrix partitioning for parallelization of matrix-vector multiply operation is described in [8, 9]. In particular, it has been shown that the partitioning objective of minimizing the cutsize (2.4) corresponds exactly to minimizing the total communication volume, and the partitioning constraint of maintaining balance on part weights (2.2) corresponds to maintaining a computational load balance for a given number K of processors.

3. Models and methods for 2D matrix partitioning.

Here, we propose three hypergraph-partitioning based methods for 2D sparse matrix partitioning for parallel $\mathbf{y} \leftarrow \mathbf{Ax}$ computations. These three methods produce nonzero-to-processor assignments, i.e., $map(a_{ij}) = P_k$ if a_{ij} is assigned to processor P_k. They do not address the vector partitioning problem. However, they rely on vector partitions being consistent with the matrix partitions; consistent in the sense that each vector entry x_j or y_i will be assigned to a processor having at least one nonzero in the corresponding column (the jth column) or row (the ith row), respectively, of $\mathbf{A}$. If the vector partitioning is consistent, then the cutsize (2.4) in the proposed hypergraph-partitioning based models will be equivalent to the total communication volume. The

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

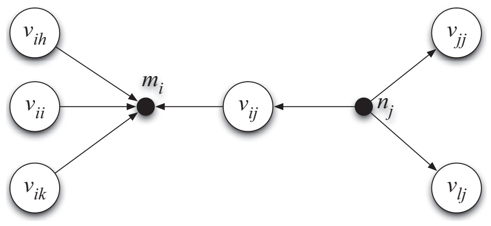


FIG. 3.1. *Dependency relation of 2D fine-grain hypergraph model.*

consistency is easy to achieve for the nonsymmetric vector partitioning; x_j can be assigned to any of the processors in $\{map(a_{ij}) : 1 \leq i \leq M \text{ and } a_{ij} \neq 0\}$, and y_i can be assigned to any of the processors in $\{map(a_{ij}) : 1 \leq j \leq N \text{ and } a_{ij} \neq 0\}$. If a symmetric partitioning is sought, then special care must be taken to assign a pair of matching input- and output-vector entries, e.g., x_i and y_i, to a processor having nonzeros in the corresponding row and column. In order to have such a processor for all vector entry pairs, the sparsity pattern of the matrix $\mathbf{A}$ can be modified to have a zero-free diagonal. In such cases, a consistent vector partition is guaranteed to exist, because the processors that own the diagonal entries can also own the corresponding input- and output-vector entries; x_i and y_i can be assigned to $map(a_{ii})$. Therefore, throughout this section, we assume that a consistent vector partitioning is always possible after partitioning the matrix $\mathbf{A}$.

3.1. Fine-grain model and partitioning method. In the fine-grain model, an $M \times N$ matrix $\mathbf{A}$ with Z nonzeros is represented as a unit-weight hypergraph $\mathcal{H}_\mathcal{Z} = (\mathcal{V}_\mathcal{Z}, \mathcal{N}_\mathcal{RC})$ with $|\mathcal{V}_\mathcal{Z}| = Z$ vertices and $|\mathcal{N}_\mathcal{RC}| = M + N$ nets for 2D partitioning. There exists one vertex $v_{ij} \in \mathcal{V}_\mathcal{Z}$ corresponding to each nonzero a_{ij} in matrix $\mathbf{A}$. For each row and for each column there exists a net in $\mathcal{N}_\mathcal{RC}$. Let $\mathcal{N}_\mathcal{RC} = \mathcal{N}_\mathcal{R} \cup \mathcal{N}_\mathcal{C}$ such that $\mathcal{N}_\mathcal{R} = \{r_1, \ldots, r_M\}$ represents the set of nets corresponding to the rows, and $\mathcal{N}_\mathcal{C} = \{c_1, \ldots, c_N\}$ represents the set of nets corresponding to the columns of the matrix $\mathbf{A}$. The net r_i contains the vertices corresponding to the nonzeros in the ith row, and the net c_j contains the vertices corresponding to the nonzeros in the jth column. That is, $v_{ij} \in r_i$ and $v_{ij} \in c_j$ if and only if $a_{ij} \neq 0$. Note that each vertex v_{ij} is a pin of exactly two nets. Each vertex v_{ij} corresponds to the scalar multiply operation $y_i^j = a_{ij}x_j$. Therefore, each column-net c_j represents the set of scalar multiply operations that need x_j during the expand phase, and each row-net r_i represents the set of scalar multiply results needed to accumulate y_i in the fold phase. Each vertex v_{ij} has unit computational weight $w_{ij} = 1$. Figure 3.1 illustrates the dependency relation view of the 2D fine-grain model. As seen in this figure, column-net $c_j = \{v_{ij}, v_{jj}, v_{lj}\}$ of size 3 represents the three scalar multiply operations $y_i^j = a_{ij}x_j$, $y_j^j = a_{jj}x_j$, and $y_l^j = a_{lj}x_j$ which need x_j. In this figure, row-net $r_i = \{v_{ih}, v_{ii}, v_{ik}, v_{ij}\}$ of size 4 represents the four scalar multiply results $y_i^h = a_{ih}x_h$, $y_i^i = a_{ii}x_i, y_i^k = a_{ik}x_k$ and $y_i^j = a_{ij}x_j$ which are needed to accumulate $y_i = y_i^h + y_i^i + y_i^k + y_i^j$.

The fine-grain partitioning method partitions the hypergraph $\mathcal{H}_\mathcal{Z}$ given above. Consider a partition $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_K\}$ of the vertices of $\mathcal{H}_\mathcal{Z}$. Without loss of generality, we assign part $\mathcal{V}_k$ to processor P_k for $k = 1, \ldots, K$. That is, for each $k = 1, \ldots, K$ we set $map(a_{ij}) = P_k$ for all $v_{ij} \in \mathcal{V}_k$. Since Π satisfies the balance constraint (2.2), it achieves a computational load balance among processors under the vertex weight definition given above. Consider an $\mathbf{x}$-vector entry x_j needed by only one processor.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Then all of the nonzeros in column j should have been assigned to a single processor. This implies that the column-net c_j connects only one part. Hence the contribution of that net to the cutsize is zero. Consider an $\mathbf{x}$-vector entry x_j needed by more than one, say p, processors. Then the nonzeros in column j should have been partitioned among these p processors. This implies that the column-net c_j connects p parts. That is, $\lambda_j = p$ holds. The contribution of this net to the cutsize is equal to $p - 1$. Due to the consistency of the vector partitioning, x_j is assigned to one of those p processors. Therefore, we have the equivalence between $\lambda_j - 1$ and the communication volume regarding x_j. Similar arguments hold for the $\mathbf{y}$-vector entries, since the sets $\mathcal{N}_\mathcal{R}$ and $\mathcal{N}_\mathcal{C}$ are disjoint.

Note that the nonzeros in the same row or column are treated independently by the fine-grain partitioning method. Therefore, neither row coherence nor column coherence is respected.

3.2. Jagged-like partitioning method. Jagged partitioning has been successively used in partitioning 2D spatial computational domains (2D workload arrays) for load balancing in the parallelization of several irregular computations including SpMxV computations on processor meshes [31, 38, 40, 41, 42]. In this method, for a $P \times Q$ processor mesh, the matrix is first partitioned into P horizontal (vertical) strips and every horizontal (vertical) strip is independently partitioned into Q submatrices. That is, splits span the entire array/matrix in one dimension, while they are jagged in the other dimension. Asymptotically and run-time efficient exact algorithms are proposed and implemented for producing jagged partitions with optimal balance [31, 36, 40]. However, the jagged partitioning methods adopted in sparse matrix partitioning unnecessarily restrict the search space since they do not utilize the flexibility of disturbing the integrity and original ordering of the rows/columns of the matrices, and furthermore, they do not consider the minimization of communication volume explicitly.

The proposed jagged-like partitioning method uses the row-net and column-net hypergraph models proposed in [8, 9]. The proposed partitioning method is a two-step method, in which each step models either the expand phase or the fold phase of the parallel SpMxV algorithm. Therefore, we have two alternative schemes for this partitioning method. We present the one which models the expands in the first step and the folds in the second step. A similar discussion holds for the other scheme.

Given an $M \times N$ matrix $\mathbf{A}$ and the number K of processors organized as a $P \times Q$ mesh, the jagged-like partitioning model proceeds as shown in Figure 3.2. The algorithm has two main steps. First, $\mathbf{A}$ is partitioned rowwise into P parts using the column-net hypergraph model $\mathcal{H}_\mathcal{R}$ discussed in section 2.3 (lines 1 and 2 of Figure 3.2). Consider a P-way partition $\Pi_\mathcal{R}$ of $\mathcal{H}_\mathcal{R}$. From the partition $\Pi_\mathcal{R}$, we obtain P submatrices $\mathbf{A}_p$ for $p = 1, \ldots, P$ each having roughly equal number of nonzeros. For each p, the rows of the submatrix $\mathbf{A}_p$ correspond to the vertices in $\mathcal{R}_p$. Hence, $\mathbf{A}_p$ is of size $M_p \times N$, where $M_p = |\mathcal{R}_p|$ (lines 6 and 7 of Figure 3.2). We assign the submatrix $\mathbf{A}_p$ to the pth row of the processor mesh. Second, each submatrix $\mathbf{A}_p$ for $1 \le p \le P$ is independently partitioned columnwise into Q parts using the row-net hypergraph $\mathcal{H}_p$ (lines 8 and 9 of Figure 3.2). Observe that the nonzeros in the ith row of $\mathbf{A}$ are partitioned among the Q processors in a row of the processor mesh. In particular, if $v_i \in \mathcal{R}_p$ at the end of line 2 of the algorithm, then the nonzeros in the ith row of $\mathbf{A}$ are partitioned among the processors in the pth row of the processor mesh. After partitioning the submatrix $\mathbf{A}_p$ columnwise, we fill the *map* array for the nonzeros residing in $\mathbf{A}_p$.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

```
JAGGED-LIKE-PARTITIONING($A, K = P \times Q, \varepsilon_1, \varepsilon_2$)
Input: a matrix $A$, the number of processors $K = P \times Q$, and the imbalance ratios $\varepsilon_1, \varepsilon_2$.
Output: $map(a_{ij})$ for all $a_{ij} \neq 0$ and totalVolume.
 1: $\mathcal{H}_\mathcal{R} = (\mathcal{V}_\mathcal{R}, \mathcal{N}_\mathcal{C}) \leftarrow$ columnNet($A$)
 2: $\Pi_\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_P\} \leftarrow$ partition($\mathcal{H}_\mathcal{R}, P, \varepsilon_1$)      ▷ rowwise partitioning of $A$
 3: expandVolume← $cutsize(\Pi_\mathcal{R})$
 4: foldVolume← 0
 5: for $p = 1$ to $P$ do
 6:     $R_p = \{r_i : v_i \in \mathcal{R}_p\}$
 7:     $A_p \leftarrow A(R_p, :)$                              ▷ submatrix indexed by rows $R_p$
 8:     $\mathcal{H}_p = (\mathcal{V}_p, \mathcal{N}_p) \leftarrow$ rowNet($A_p$)
 9:     $\Pi_p^\mathcal{C} = \{\mathcal{C}_p^1, \ldots, \mathcal{C}_p^Q\} \leftarrow$ partition($\mathcal{H}_p, Q, \varepsilon_2$)      ▷ columnwise partitioning of $A_p$
10:     foldVolume←foldVolume $+cutsize(\Pi_p^\mathcal{C})$
11:     for all $a_{ij} \neq 0$ of $A_p$ do
12:         $map(a_{ij}) = P_{p,q} \Leftrightarrow c_j \in \mathcal{C}_p^q$
13: return totalVolume←expandVolume+foldVolume
```

FIG. 3.2. *Jagged-like partitioning.*

Consider processor loads obtained according to the *map* array at the end of the algorithm. The Q processors in a row of the processor mesh are assigned a roughly equal number of nonzeros, i.e., each having at most $(1+\varepsilon_2)\frac{nnz(\mathbf{A}_p)}{Q}$ nonzeros, due to the balance constraint (2.2) met while partitioning $\mathcal{H}_p$. Furthermore, we have $nnz(\mathbf{A}_p) \leq (1+\varepsilon_1)\frac{nnz(\mathbf{A})}{P}$ for all p, due to the balance constraint met while partitioning $\mathcal{H}_\mathcal{R}$. Therefore, a processor can get as many as $(1+\varepsilon_1+\varepsilon_2+\varepsilon_1\varepsilon_2)\frac{nnz(\mathbf{A})}{K}$ nonzeros. In other words, the resulting K-way partitioning of $\mathbf{A}$ is guaranteed to satisfy a balance constraint with an imbalance ratio of $\varepsilon = (\varepsilon_1+\varepsilon_2+\varepsilon_1\varepsilon_2)$.

Consider a $\mathbf{y}$-vector entry y_i. As noted above, the nonzeros in the row r_i are partitioned in the columnwise partitioning of the submatrix $\mathbf{A}_p$ containing r_i (line 9 of the algorithm). Hence, the processors that contribute to y_i exactly correspond to the parts in the connectivity set of the row-net r_i in $\mathcal{H}_p$. That is, the volume of communication required to fold y_i is accurately represented as a part of "foldVolume" in the algorithm. Consider an $\mathbf{x}$-vector entry x_j. Suppose it is needed by q processors. Then, the nonzeros in the jth column of $\mathbf{A}$ should have been partitioned among those q processors. Note that due to the row-net model representing the columns as vertices, the nonzeros of the jth column in a submatrix $\mathbf{A}_p$ are assigned to exactly one processor. In other words, the jth column of $\mathbf{A}$ should have nonzeros in q submatrices after the rowwise partitioning in line 2 of the algorithm. Therefore, the connectivity of the net $n_j \in \mathcal{N}_\mathcal{C}$ should be q. Due to the consistency of the vector partitioning, the volume of communication regarding x_j is equal to $(q-1) = (\lambda_j - 1)$. Therefore, the volume of communication regarding x_j is accurately represented as a part of "expandVolume" in the algorithm.

As an example run of the algorithm, consider the 16×16 matrix shown in Figure 2.1 to be partitioned among the processors of a 2×2 mesh. Figure 3.3(a) illustrates the column-net representation of the sample matrix. For simplicity of presentation, we labeled the vertices and the nets of the hypergraphs with letters "r" and "c" to denote the rows and columns of the matrix. We first partition the matrix rowwise into two parts, and assign each part to a row of the processor mesh, namely, to processors $\{P_1, P_2\}$ and $\{P_3, P_4\}$. The resulting permuted matrix is displayed in Figure 3.3(b).

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

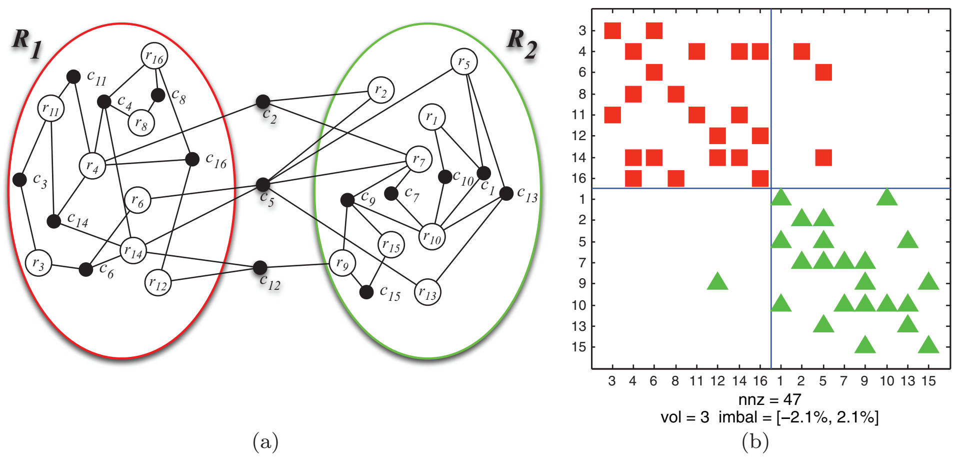


FIG. 3.3. *First step of 4-way jagged-like partitioning of the matrix shown in Figure* 2.1: (a) 2*-way partitioning* $\Pi_{\mathcal{R}}$ *of column-net hypergraph representation* $\mathcal{H}_{\mathcal{R}}$ *of* **A**; (b) 2*-way rowwise partitioning of matrix* $\mathbf{A}^{\Pi}$ *obtained by permuting* **A** *according to the partitioning induced by* Π; *the nonzeros in the same partition are shown with the same shape and color; the deviation of the minimum and maximum numbers of nonzeros of a part from the average are displayed as an interval* `imbal`; `vol` *denotes the number of nonzeros and the total communication volume.*

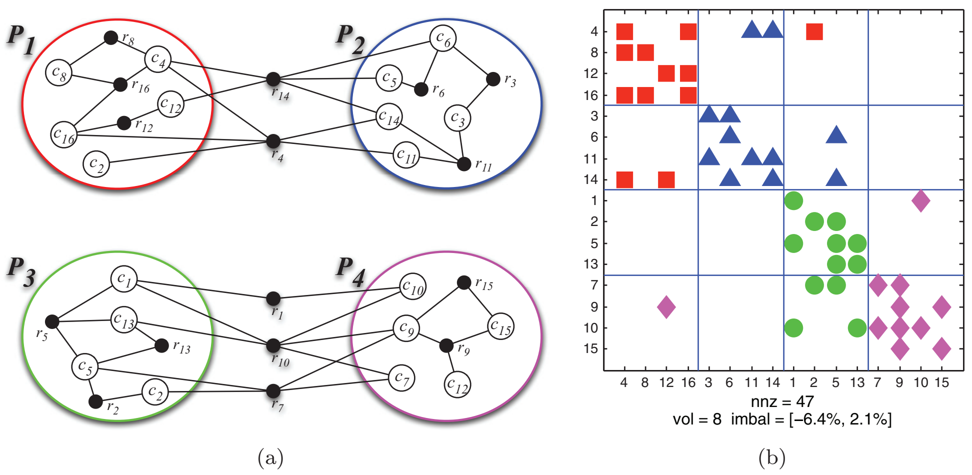


FIG. 3.4. *Second step of* 4*-way jagged-like partitioning:* (a) *Row-net representations of submatrices of* **A** *and* 2*-way partitionings;* (b) *Final permuted matrix; the nonzeros in the same partition are shown with the same shape and color; the deviation of the minimum and maximum numbers of nonzeros of a part from the average are displayed as an interval* `imbal`; `nnz` *and* `vol` *denote, respectively, the number of nonzeros and the total communication volume.*

Figure 3.4(a) displays the two row-net hypergraphs corresponding to each submatrix $\mathbf{A}_p$ for $p = 1, 2$. Each hypergraph is partitioned independently; sample partitions of these hypergraphs are also presented in this figure. As seen in the final symmetric permutation in Figure 3.4(b), the coherences of columns 2 and 5 are not maintained, causing P_3 to communicate with both P_1 and P_2 in the expand phase.

Note that we define $\mathbf{A}_p$ as of size $M_p \times N$ for all p (line 5 of Figure 3.2) for ease of presentation. Normally, $\mathbf{A}_p$ contains only those columns of **A** that have nonzeros in any of the rows in $\mathcal{R}_p$. Some of the columns of **A** are internal to the row part $\mathcal{R}_p$.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

```
CHECKERBOARD-PARTITIONING($A, K = P \times Q, \varepsilon_1, \varepsilon_2$)
Input: a matrix $\mathbf{A}$, the number of processors $K = P \times Q$, and the imbalance ratios $\varepsilon_1, \varepsilon_2$.
Output: $map(a_{ij})$ for all $a_{ij} \neq 0$ and totalVolume.
 1: $\mathcal{H}_\mathcal{R} = (\mathcal{V}_\mathcal{R}, N_\mathcal{C}) \leftarrow$ columnNet($A$)
 2: $\Pi_\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_P\} \leftarrow$ partition($\mathcal{H}_\mathcal{R}, P, \varepsilon_1$)          ▷ rowwise partitioning of $\mathbf{A}$
 3: expandVolume$\leftarrow$ $cutsize(\Pi_\mathcal{R})$
 4: $\mathcal{H}_\mathcal{C} = (\mathcal{V}_\mathcal{C}, \mathcal{N}_\mathcal{R}) \leftarrow$ rowNet($A$)
 5: for $j = 1$ to $|\mathcal{V}_\mathcal{C}|$ do
 6:     for $p = 1$ to $P$ do
 7:         $w_{j,p} = |\{n_j \cap \mathcal{R}_p\}|$
 8: $\Pi_\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_Q\} \leftarrow$ MCPartition($\mathcal{H}_\mathcal{C}, Q, w, \varepsilon_2$)   ▷ columnwise partitioning of $\mathbf{A}$
 9: foldVolume$\leftarrow$ $cutsize(\Pi_\mathcal{C})$
10: for all $a_{ij} \neq 0$ of $A$ do
11:     $map(a_{ij}) = P_{p,q} \Leftrightarrow r_i \in \mathcal{R}_p$ and $c_j \in \mathcal{C}_q$
12: totalVolume$\leftarrow$expandVolume+foldVolume
```

FIG. 3.5. *Checkerboard partitioning.*

These columns appear as a vertex only in $\mathcal{H}_p$. Some other columns have nonzeros in more than one part of $\Pi_\mathcal{R}$. Those columns correspond precisely to the external nets in $\Pi_\mathcal{R}$. That is, each external net n_j in $\Pi_\mathcal{R}$ appears as a vertex in all $\mathcal{H}_q$ for $\mathcal{R}_q \in \Lambda_j$. For example, as seen in Figure 3.3(a), the column-net c_5 is an external net with $\Lambda_5 = \{\mathcal{R}_1, \mathcal{R}_2\}$; hence as displayed in Figure 3.4(a) each hypergraph contains a vertex for column 5, namely, c_5. Note that if $\mathbf{A}_p$ has N_p nonzero columns for $p = 1, \ldots, P$, then $\sum \mathcal{N}_p = N + cutsize(\Pi_R)$.

3.3. Checkerboard partitioning method. The proposed checkerboard partitioning method is also a two-step method, in which each step models either the expand phase or the fold phase of the parallel SpMxV. Similar to jagged-like partitioning, we have two alternative schemes for this partitioning method. Here, we present the one which models the expands in the first step and the folds in the second step. An analogous discussion holds for the other scheme.

Given an $M \times N$ matrix $\mathbf{A}$ and the number K of processors organized as a $P \times Q$ mesh, the checkerboard partitioning method proceeds as shown in Figure 3.5. First, $\mathbf{A}$ is partitioned rowwise into P parts using the column-net model (lines 1 and 2 of Figure 3.5), producing $\Pi_\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_P\}$. Note that this first step is exactly the same as that of the jagged-like partitioning. In the second step, the matrix $\mathbf{A}$ is partitioned columnwise into Q parts by using the multi-constraint partitioning to obtain $\Pi_\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_Q\}$. In comparison to the jagged-like method, we partition the whole matrix $\mathbf{A}$ (lines 4 and 8 of Figure 3.5), not the submatrices defined by $\Pi_\mathcal{R}$. The rowwise and columnwise partitions $\Pi_\mathcal{R}$ and $\Pi_\mathcal{C}$ together define a 2D partition on the matrix $\mathbf{A}$, where $map(a_{ij}) = P_{p,q} \Leftrightarrow r_i \in \mathcal{R}_p$ and $c_j \in \mathcal{C}_q$.

In order to achieve a load balance among processors, we use multi-constraint partitioning in line 8 of the algorithm. Each vertex v_i of $\mathcal{H}_\mathcal{C}$ is assigned G weights: $w_{i,p}$, for $p = 1, \ldots, P$. Here, $w_{i,p}$ is equal to the number of nonzeros of column c_i in rows $\mathcal{R}_p$ (line 7 of Figure 3.5). Consider a Q-way partitioning of $\mathcal{H}_\mathcal{C}$ with P constraints using the vertex weight definition above. Maintaining the P balance constraints (2.5) corresponds to maintaining computational load balance on the processors of each row of the processor mesh. That is, the loads of the Q processors in the pth row of the

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

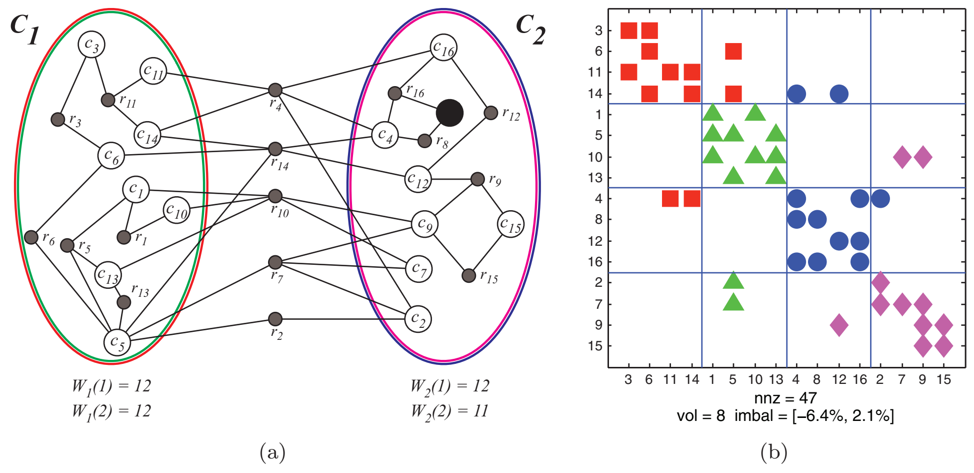


FIG. 3.6. *Second step of* 4*-way checkerboard partitioning:* (a) 2*-way multi-constraint partitioning* $\Pi_{\mathcal{C}}$ *of row-net hypergraph representation* $\mathcal{H}_{\mathcal{C}}$ *of* $\mathbf{A}$*;* (b) *Final checkerboard partitioning of* $\mathbf{A}$ *induced by* $(\Pi_{\mathcal{R}}, \Pi_{\mathcal{C}})$*; the nonzeros in the same partition are shown with the same shape and color; the deviation of the minimum and maximum numbers of nonzeros of a part from the average are displayed as an interval* `imbal`*;* `nnz` *and* `vol` *denote, respectively, the number of nonzeros and the total communication volume.*

processor mesh satisfies $W_{p,q} \le (1+\varepsilon_2)\frac{\sum_i w_{i,p}}{Q}$. We also have $\sum_i w_{i,p} \le (1+\varepsilon_1)\frac{nnz(\mathbf{A})}{P}$, due to the P-way ε_1-balanced partitioning in line 2. As in the jagged-like partitioning, the resulting K-way partitioning of $\mathbf{A}$ is guaranteed to satisfy a balance constraint with an allowable imbalance ratio of $\varepsilon = (\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2)$.

Establishing the equivalence between the total communication volume and the sum of the cutsizes of the two partitions is fairly straightforward. We observe that if the nonzeros in the ith row of $\mathbf{A}$ are partitioned among q processors, then the row-net r_i of $\mathcal{H}_{\mathcal{C}}$ will connect q parts after the columnwise partitioning in line 8 of the algorithm. That is, the volume of communication for the fold operations corresponds exactly to the $cutsize(\Pi_{\mathcal{C}})$. Similarly, if the nonzeros in the jth column of $\mathbf{A}$ are partitioned among p processors, then the net c_j of $\mathcal{H}_{\mathcal{R}}$ should have vertices in the same set of parts after the rowwise partitioning in line 2 of the algorithm. That is, the volume of communication for the expand operations corresponds exactly to the $cutsize(\Pi_{\mathcal{R}})$.

We demonstrate the main steps of the proposed checkerboard partitioning method on the sample matrix shown in Figure 2.1 for a 2×2 processor mesh. First, a 2-way rowwise partition $\Pi_{\mathcal{R}}$ is obtained, giving the same figure as shown in Figure 3.3. Figure 3.6(a) displays the row-net hypergraph representation $\mathcal{H}_{\mathcal{C}}$ of matrix $\mathbf{A}$. It also shows a 2-way multi-constraint partition $\Pi_{\mathcal{C}}$ of $\mathcal{H}_{\mathcal{C}}$. In Figure 3.6(a), $w_{9,1} = 0$ and $w_{9,2} = 4$ for internal column c_9 of row stripe $\mathcal{R}_2$, whereas $w_{5,1} = 2$ and $w_{5,2} = 4$ for external column c_5. Figure 3.6(b) displays the 2×2 checkerboard partition induced by $(\Pi_{\mathcal{R}}, \Pi_{\mathcal{C}})$.

Compared to the jagged-like partitioning method, the checkerboard partitioning method maintains both row and column coherences at the level of the row or columns of the processor mesh. It confines the expand and fold communications to the processors of the same column and row of the processor mesh, respectively. In this way, the number of messages to be sent and received by a processor is limited to $P-1$ and $Q-1$ processors in the expand and fold phases, respectively.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

4. Further investigations and comments.

4.1. Comparison of the models.

4.1. Comparison of the models. 1D rowwise partitioning incurs only expand communication, because it respects row coherence by assigning entire rows to processors while disturbing column coherence. In a dual manner, 1D columnwise partitioning incurs only fold communication, because it respects column coherence by assigning entire columns to processors while disturbing row coherence. Therefore, in the 1D matrix partitioning, the number of messages sent by a processor may be as high as $K-1$, for a parallel system with K processors, giving a total of $K(K-1)$ messages. In a rowwise partitioning, the worst-case total communication volume is $(K-1)N$ for an $M \times N$ matrix, and this worst case occurs when each column has at least one nonzero in each row stripe. Similarly, for a columnwise partitioning, the worst-case total communication volume is $(K-1)M$.

In the fine-grain partitioning method, nonzeros are allowed to be assigned individually to processors. Since neither row coherence nor column coherence is enforced, this method may incur both expand and fold operations, and hence the number of messages sent by a processor may be as high as $2(K-1)$, giving a total of $2K(K-1)$ messages. The worst-case communication volume is $(K-1)(M+N)$ units in total. The proposed fine-grain hypergraph partitioning model is highly flexible, since it enables assignment of each nonzero entry individually—it can partition a given matrix among Z processors as compared to, for example, M processors in rowwise partitioning. The fine-grain model has a higher degree of freedom than the 1D models in minimizing communication volume, since it enforces neither row coherence nor column coherence.

The jagged-like partitioning is intrinsically better than the fine-grain partitioning in terms of the total number of messages. In the expand communication phase, the maximum number of messages per processor is $P \times Q - Q = K - Q$ for a $P \times Q$ mesh of processors, since the processors in the same row of the processor mesh do not require communication of $\mathbf{x}$-vector components. In the fold communication phase, the maximum number of messages per processor is $Q-1$, since row coherence is maintained at the level of rows of the processor mesh. Hence, the upper bound on the total number of messages in jagged-like partitioning is $K(K-Q)+K(Q-1) = K(K-1)$. The total communication volume may be as high as $(P-1)N+(Q-1)M$, and this worst case occurs when each row and column of each submatrix of $\mathbf{A}^{(k)}$, as shown in (2.1), has at least one nonzero.

The 2D checkerboard partitioning method can be considered as a trade-off between 1D partitioning and 2D fine-grain partitioning methods. This method respects both row and column coherences in a coarse level. It respects row coherence by assigning entire matrix rows to the processors in the same row of the processor mesh. It also respects column coherence by assigning entire matrix columns to the processors in the same column of the processor mesh. In other words, this method confines the expand and fold operations, respectively, to the columns and the rows of the processor mesh. In this way, it reduces the maximum number of messages sent by a processor to $P+Q-2$ for a $P \times Q$ mesh of processors; if $P = Q = \sqrt{K}$, this results in $2K(\sqrt{K}-1)$ messages in total. The total communication volume may be as high as $(P-1)N+(Q-1)M$, and this worst case occurs when each row and column of each submatrix has at least one nonzero.

4.2. Modeling collective communication. As discussed above, the proposed jagged-like partitioning method confines the communication regarding $\mathbf{y}$-vector entries to a row of the processor mesh; i.e., each y_i will be computed by at most Q processors.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

The checkerboard partitioning approach goes one step further and also confines the communications regarding the **x**-vector entries to a column of the processor mesh; i.e., each x_j is needed by at most P processors. Since P and Q are usually much smaller than K, all-to-all communication looks affordable in the row-column-parallel multiply algorithm given in section 2.1. More precisely, if jagged-like or checkerboard partitioning approaches are used, steps 4 and 5 of the row-column-parallel SpMxV algorithm given in section 2.1 can be replaced by an optimized all-to-all reduction operation. If the checkerboard partitioning approach is used, then steps 1 and 2 of the same algorithm can be replaced by an optimized all-to-all broadcast operation. Those collective communication operations can be implemented for any number of processors [24]; i.e., the number of processors is not restricted to the powers of two. The attractive feature of this all-to-all communication scheme is that it reduces the maximum number of messages per processor to $\lceil \log P \rceil$ or $\lceil \log Q \rceil$ in the expand or fold communication phases. When such an optimized all-to-all scheme is used, the total communication volume can be minimized by reducing the total number of **x**-vector entries expanded or **y**-vector entries folded. Clearly, the objective here is equivalent to reducing the number columns and rows whose nonzeros are shared by more than one processor. The models proposed in this work can be directly used to address this problem by using the cut-net objective function (2.3) instead of the connectivity-1 objective function (2.4).

4.3. How to apply hypergraph-based partitioning to other applications. Although we have exclusively considered the SpMxV, there are other applications that can make use of the contributions of the current work—in general, matrix partitioning methods. Parallel reduction (aggregation) operations form a significant class of such applications [16, 18]. The reduction operation consists of computing M output elements using N input elements. An output element may depend on multiple input elements, and an input element may contribute to multiple output elements. Assume that the operation on which reduction is performed is commutative and associative. Then, the inherent computational structure can be represented with an $M \times N$ dependency matrix, where each row and column of the matrix represents an output element and an input element, respectively. For an input element x_j and an output element y_i, if y_i depends on x_j, a_{ij} is set to 1 (otherwise zero). Using this representation, the problem of partitioning the workload for the reduction operation is equivalent to the problem of partitioning the dependency matrix for efficient SpMxV [13, 29].

In some other reduction problems, the input and output elements may be pre-assigned to parts. The proposed hypergraph model can be accommodated to those problems by adding K *part* vertices and connecting those vertices to the nets which correspond to the pre-assigned input and output elements. Obviously, those part vertices must be fixed to the corresponding parts during the partitioning. Since the required property is already included in the existing hypergraph partitioners [1, 6, 10, 28], this does not add extra complexity to our methods.

4.4. A recipe for matrix partitioning. The following abbreviations will be used here and hereafter for the matrix partitioning methods discussed so far:

- RW: Rowwise 1D partitioning,
- CW: Columnwise 1D partitioning,
- FG: Fine-grain 2D partitioning,
- JL: Jagged-like 2D partitioning,
- CH: Checkerboard 2D partitioning.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

As discussed so far, the FG method is most likely to give better total communication volume and computational load balance than any other method discussed. However, it is also most likely to be the slowest. The CH method, on the other hand, should be the fastest, it most likely obtains better total number of messages than any of the others, but it is likely to obtain the worst total communication volume and the worst computational load balance. The JL method should be in between these two methods in almost any metric considered. Except in extremely skewed matrices, 1D partitioning methods can never be significantly better than all of the remaining methods.

As there are a number of alternative partitioning methods, each with a different trait, a means to automate the decision of choosing which alternative to partition a given matrix is necessary. If any of the metrics mentioned above is significantly more important than the others, then the best method should be chosen. For example, if the total number of messages is of utmost importance, then the checkerboard partitioning method seems to be the method of choice, as it has the lowest limit for this metric. However, usually a combination of the communication metrics, including the maximum volume and message sent by a processor [47] or these two quantities both in terms of sends and receives [3], corresponds to the communication cost. Usually, a user of the partitioning methods does not have to know all the details of the partitioning methods. Therefore, we present a recipe that tries to suggest a partitioning method for a given matrix, where the suggestions are made for the total communication volume, meanwhile trying to reduce the other metrics on the average. Notice that for metrics other than the total volume of communication, different recipes can be developed. As the principal aim of hypergraph partitioning models is to minimize the total communication volume, we think that a recipe based on this metric could be the most accurate and beneficial one. Note that for 1D partitioning methods, it has been already said that it is advisable to partition along the columns (rows) if the given matrix has dense rows (columns) but no dense columns (rows) [21]. This advice, although it is sound for 1D partitioning, is not adequate for 2D partitioning, as neither row coherence nor column coherence is guaranteed to be respected.

The proposed recipe is shown in Figure 4.1. A user provides a matrix $\mathbf{A}$, number of processors K and an imbalance ratio ε (usually less than 3%) to use the recipe. A number of quantities are then computed and compared with certain threshold values to suggest a method. The recipe uses the pattern symmetry score, $Sym(\mathbf{A}) = \sum_{p_{ij}} p_{ij}p_{ji}/Z$, where $p_{ij} = 1$ if $a_{ij} \neq 0$ and zero otherwise and a number of statistical descriptions of the row degrees (shown with d_r) and the column degrees (shown with d_c). In the recipe, *max*, *avg*, and *med* represent, respectively, the maximum, average, and median; *mode* is the value that has the largest number of occurrences; e.g., mode(d_r) is the most occurring row degree. Q_3 is the third quartile; e.g., $Q_3(d_c)$ is a number which is greater than 75% of the column degrees and smaller than the rest. Note that all these numbers, including the symmetry score, can be computed in $O(Z + M + N)$ time—see [14, Chapter 10] for median and order statistics and [45, p. 720] for calculating the symmetry score.

The recipe has three sets of tests to suggest a method: one set for rectangular matrices, one set for square and almost symmetric (with a pattern symmetry score larger than 0.95) matrices, and another set for the remaining square unsymmetric matrices. For a rectangular matrix, if it is a very tall or very wide matrix, it selects the CW or the RW method, respectively; otherwise it chooses the FG method. For square matrices, it chooses the FG method for pathological cases—when the number of nonzeros is less than the number of rows ($Z \leq M$) and mode of the row or column degrees is zero; the same choice is made for those matrices where the CH or JL

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

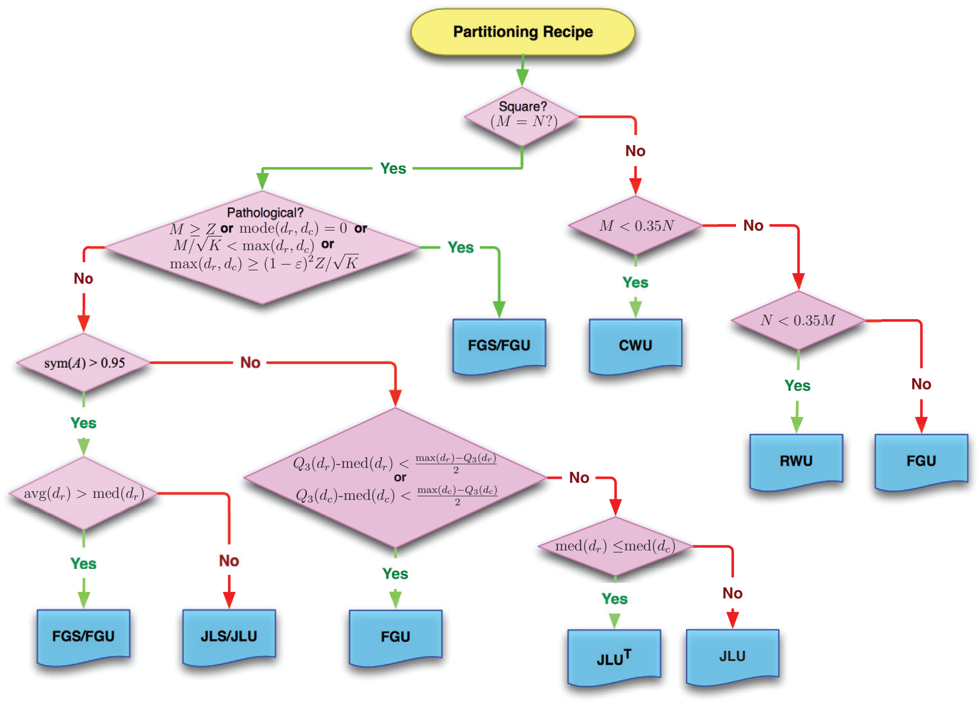

Partitioning Recipe
Square? $(M = N?)$
Yes
No
Pathological? $M \geq Z$ or mode$(d_r, d_c) = 0$ or $M/\sqrt{K} < \max(d_r, d_c)$ or $\max(d_r, d_c) \geq (1-\varepsilon)^2 Z/\sqrt{K}$
Yes
No
FGS/FGU
$M < 0.35N$
Yes
No
CWU
$N < 0.35M$
Yes
No
RWU
FGU
sym$(A) > 0.95$
Yes
No
avg$(d_r) >$ med(d_r)
Yes
No
FGS/FGU
JLS/JLU
$Q_3(d_r)$-med$(d_r) < \frac{\max(d_r)-Q_3(d_r)}{2}$ or $Q_3(d_c)$-med$(d_c) < \frac{\max(d_c)-Q_3(d_c)}{2}$
Yes
No
FGU
med$(d_r) \leq$ med(d_c)
Yes
No
JLU^T
JLU

FIG. 4.1. *A recipe for matrix partitioning. Matrix* $\mathbf{A}$ *is of size* $M \times N$ *with* Z *nonzeros to be partitioned among* K *processors. The arrays* d_r *and* d_c *represent the row and column degrees; i.e.,* $d_r(i)$ *is the number of nonzeros in row* i*.The statistical descriptors max, avg, and med represent, respectively, the maximum, average, and median; mode is the value that has the largest number of occurrences.* Q_3 *is the third quartile; e.g.,* $Q_3(d_c)$ *is a number which is greater than 75% of the column degrees. Sym*$(\mathbf{A})$ *measures the symmetry score, and* ε *is a user specified allowed imbalance ratio. Letters S and U after the method names (RW, CW, FG, JL, CH) represent symmetric and unsymmetric vector partitioning, respectively.*

methods could lead to a load imbalance or have a hard time obtaining balance, i.e., $\max(d_r, d_c) \geq (1-\varepsilon)^2 Z/\sqrt{K}$. For pattern symmetric or almost pattern symmetric matrices (i.e., $Sym(\mathbf{A}) > 0.95$), the recipe chooses the FG method, if the average row/column degree is greater than the median; otherwise it chooses the JL method. Note that in any case a symmetric vector partitioning is always suggested for these types of matrices. For the unsymmetric square matrices with $Sym(\mathbf{A}) \leq 0.95$, the recipe chooses the FG method, if there is a certain relation among the median, third quartile, and the maximum of the row or column degrees; otherwise a variant of JL partitioning is suggested. For these matrices, the recipe uses the JL method that performs columnwise partitioning first (JL^T) when the median of the row degrees is smaller than that of the column degrees. The tests avg$(d_r) >$ med(d_r) and the big one with the quartiles try to see if there are sufficiently large numbers of rows or columns with a high number of nonzeros. With the aid of the test med$(d_r) \leq$ med(d_c), the recipe tries to adapt the advice on 1D partitioning to the JL method; i.e., if the median degree of the rows is smaller than the median degree of the columns, then most probably there are more dense rows than columns, and hence partitioning along the columns in the first step is advisable. With this test, the recipe also tries to leave more flexibility to the second phase of the JL partitioning method in terms of load balancing.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

5. Experimental results.

We performed an extensive experimental evaluation of the proposed 2D sparse matrix partitioning methods as well as 1D partitioning methods [9] using almost all large matrices of the University of Florida (UFL) sparse matrix collection [15]. Here, we first present the results of this experimental evaluation and then investigate the effectiveness of the partitioning recipe.

5.1. Test dataset and experimental setup.

We ran our tests using the newly developed PaToH Matlab Matrix-Partitioning Interface [10, 51] (PaToH and Matlab Matrix-Partitioning Interface are available at http://bmi.osu.edu/~umit/software.html) on a 32-node cluster owned by the Department of Biomedical Informatics at The Ohio State University. Each computer is equipped with dual 2.4 GHz Opteron 250 processors, 8 GB of RAM, and 500 GB of local storage. Nodes are interconnected by a switched Infiniband network. We ran PaToH using default parameters. In order to facilitate the running of thousands of sequential partitionings via Matlab interface, we developed a simple first-in-first-out job scheduler and Matlab script executer, *dcexec*, using DataCutter [2], that runs each partitioning one-by-one on one of the available nodes. The script *dcexec* keeps an instance of Matlab running on each node and passes Matlab partitioning commands via the use of a FIFO file, hence avoiding Matlab startup overhead for each partitioning.

We excluded matrices that have less than 500 nonzeros. Since our testing environment is based on sequential partitioning of the matrices within the Matlab environment, we also excluded matrices that have more than 10,000,000 nonzeros. There were 1,413 matrices in the UFL collection satisfying these properties (there were a total of 1,877 matrices at the time of experimentation among which 57 had more than 10,000,000 nonzeros). We tested with $K \in \{4, 16, 64, 256\}$. For a specific K value, K-way partitioning of a test matrix constitutes a partitioning instance. The partitioning instances in which $\min\{M, N\} < 50 \times K$ are discarded, as the parts would become too small to be meaningful. These resulted in 4,100 partitioning instances, among which 1,932 were with a symmetric matrix, 1,456 were with a square unsymmetric matrix, and 712 were with a rectangular matrix (on 45 instances $M > N$ and on 667 instances $M < N$).

5.2. Partitioning methods.

We tested all five partitioning methods RW, CW, FG, JL and CH for every partitioning instance. We also include the results of the recipe discussed in section 4.4. As discussed before, the recipe, being a meta-partitioning method, chooses a partitioning method from the above list, using some matrix statistics, and applies the chosen partitioning method.

We considered symmetric and nonsymmetric vector partitioning for square matrices. In the symmetric case, we added missing diagonal entries to the matrices before partitioning and assigned the vector entries to the parts which contained the corresponding diagonal entry. In the nonsymmetric vector partitioning case, we used a simple approach to assign vector entries after the matrix partitioning. In this approach, each vector entry x_j or y_i was assigned to a part having at least one nonzero in the corresponding column (the jth column) or row (the ith row), respectively, of $\mathbf{A}$. If more than one part was qualified for assignment, we arbitrarily picked the one with the least number of vector entries assigned so far.

For checkerboard and jagged-like partitioning, our default approach partitions the matrix rowwise in the first step and columnwise in the second step. For completeness, for unsymmetric matrices we also considered changing the order of partitioning direction, which is achieved by taking the transpose of the input matrix prior to parti-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

tioning. Reported results for checkerboard and jagged-like partitioning include these additional methods, and they are referred to as JL^T and CH^T.

As PaToH involves randomized algorithms, we obtained 10 different partitions for each partitioning instance with every applicable method and used the average of the 10 partitionings as the representative result for that particular method on that particular partitioning instance. In all partitioning instances, maximum allowable imbalance ratio ε, see (2.2) and (2.5), is set to 3%. Although the balance constraint is met in most of the partitionings, it was not feasible in some of the problem instances. We will try to point out the balance problems while explaining the results.

The jagged-like and checkerboard methods assume a virtual 2D processor mesh (see sections 3.2 and 3.3). In our experiments, for mesh dimensions P and Q we selected $P = Q = \sqrt{K}$, for $K \in \{4, 16, 64, 256\}$. The multi-constraint partitioning techniques have been observed to perform worse when the number of constraints is increased [1, 49]. Therefore, for partitioning cases with $P \neq Q$, we suggest partitioning first for the smaller of P and Q to have a smaller number of constraints in the second phase of the checkerboard method.

5.3. Performance profiles. We use a generic tool, *performance profiles*, introduced by Dolan and Moré [17] for comparing a set of *methods* over a large set of *test cases* (in our case, the partitioning instances) with regard to a specific performance *metric*. The main idea behind performance profiles is to use a cumulative *distribution function* for a performance metric, instead of, for example, taking averages over all the test cases. We will compare the partitioning methods using the following metrics: the total communication volume, the total number of messages, the maximum volume of messages sent by a single processor, the computational load imbalance, and the partitioning time.

Each performance profile plot helps compare different methods with respect to a specific metric. For a given metric, a profile plot shows the probability that a specific partitioning method gives results which are within some value τ of the best result reached by all methods. Therefore the higher the probability, the more preferable the method is. For example, for the total communication volume metric, a τ value shows the probability for a partitioning method that the total communication volume obtained by that method is within τ of the best result reached by all methods shown in the same plot.

On 63 partitioning instances, the minimum total volume of communication found by at least one partitioning method was zero. We exclude those instances while plotting the performance profiles. In the performance profiles for the rectangular matrices, the plots of RW, CW, FG, JL, and CH always refer to the results of the associated partitioning method, post-processed with a nonsymmetric vector partitioning. In the profiles for the symmetric matrices, these labels always refer to the results of the associated partitioning method with a symmetric vector partitioning (and hence the missing diagonal entries are always added to the matrices before partitioning). The unsymmetric square case is a little more complicated. For these matrices, RW, CW, and FG always refer to the best result (with respect to the total communication volume) of the associated method with a symmetric and nonsymmetric vector partitioning (in the symmetric vector partitioning case the missing diagonal entries are added). JL and CH, in addition to the two different vector partitioning approaches, also include the best of the transposed partitioning approaches, i.e., JL^T and CH^T. In the performance profile figures labeled with "all instances," each method refers to the

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

best of symmetric (whenever applicable), nonsymmetric, and transposed approaches (whenever applicable). We did not use transposed approaches on the rectangular matrices, as this will change the size of the matrices. The symmetric matrices usually require symmetric vector partitioning (for example, in linear system solvers for symmetric matrices). In unsymmetric matrices, all methods with all variations are acceptable. With the "all instances" figures, we mean to show what can be expected when a certain partitioning approach is used with all variations; e.g., a user of these partitioning methods tries all possible combinations at hand and uses the best (in terms of the total communication volume).

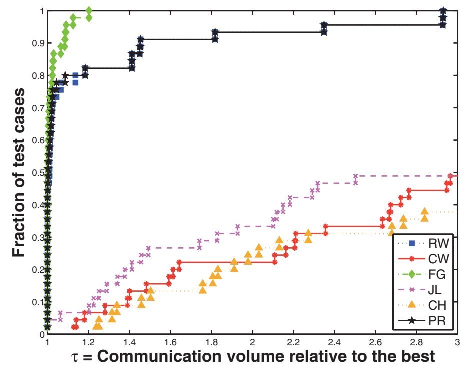
5.4. Results. Figure 5.1 displays performance profiles of five partitioning methods as well as the proposed recipe, using the total communication volume as the comparison metric. As seen in Figure 5.1(f), in almost 90% of the partitioning instances the FG method obtains results within 1.2 of the best. In rectangular instances (Figures 5.1(a)–5.1(c)), when the number of rows is greater than the number of columns (i.e., $M > N$, Figure 5.1(a)), as one might expect, the RW method produces the second-best results, and when the number of columns is greater than the number of rows (Figure 5.1(b)), the CW method produces the second-best results. When we consider all rectangular instances (Figure 5.1(c)), since there are a lot more instances with more columns than rows (662 vs 45—here 5 matrices were discarded because they had zero total communication volume), the CW method still produces the second-best results. In all different types of instances, CH produces the worst results because it is the most restricted partitioning method. However, in almost 75% of partitioning instances even CH results are within 2 of the best.

Figure 5.2 displays the comparison of the methods using the total number of messages as the comparison metric. As seen in the figure, this is the metric for which CH shines, as it inherently limits the maximum number of messages to a much smaller number than the others. In almost 75% of the partitioning instances, it produces the least number of messages, and in 95% of the partitioning instances, it produces the results within 1.5 of the best. Although the upper bound on the total number of messages for the JL method is the same as those for the RW and CW methods ($K(K-1)$), JL seems to be preferable for square matrices (for rectangular matrices, if $M > N$, RW is preferable, else CW is preferable). As expected, FG achieves the worst performance in this metric.

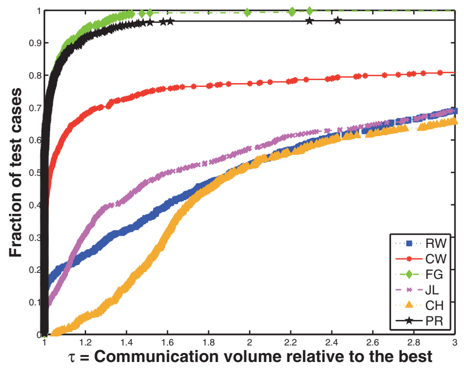
Figure 5.3 displays the comparisons on the metrics of the maximum volume and maximum number of messages of a processor. Since the trends in these two metrics are similar to those of the total volume and the total number of messages, we present a single performance profile plot for each of those metrics (containing all instances). In terms of the maximum volume per processor metric, FG and JL generally produce better results. The CH method produces more pronounced best results for the maximum number of messages per processor metric, since its upper bound is the smallest among all. In almost 85% of the partitioning instances, CH produces the best results.

Figure 5.4(a) displays the comparison of average imbalance ratios of the computational loads of the processors. In 1,411 partitioning instances (about 35% of the whole), one of the methods found a perfect balance; that is, the best ε was 0, see (2.2) and (2.5). Hence, the plots do not reach up to 1 in the y-axis of the performance profile plots. As expected, FG achieves better partitionings in terms of imbalance ratios because it does its assignments in the finest granularity possible. Actually, FG found partitions with perfect balance in 1,266 partitioning instances. Recall that the CH and JL methods are two-step methods, and unfortunately solution of the first step

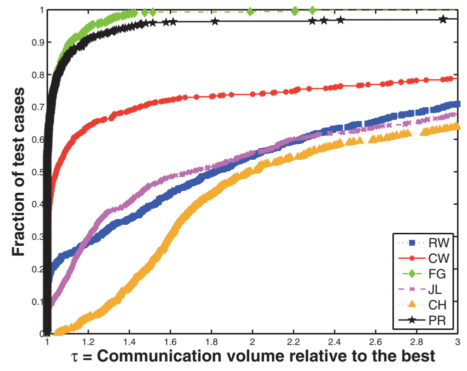
Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.



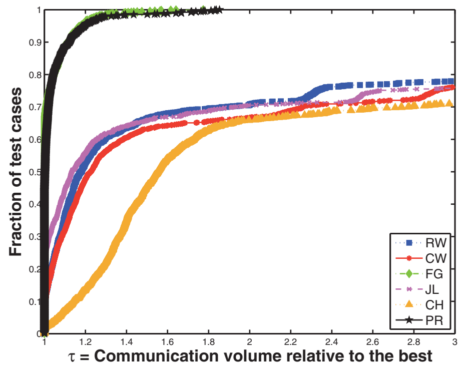
(a) rectangular instances $M > N$



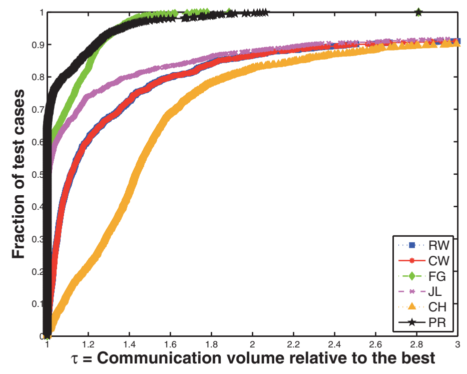
(b) rectangular instances $N > M$



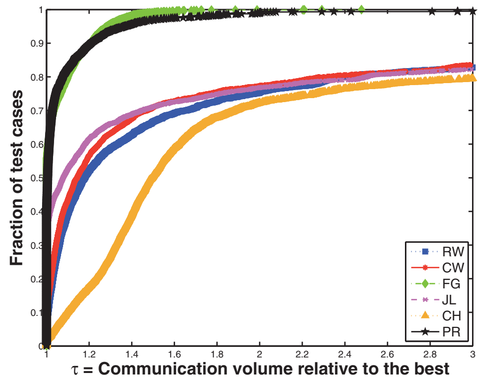
(c) all rectangular instances



(d) square nonsymmetric instances



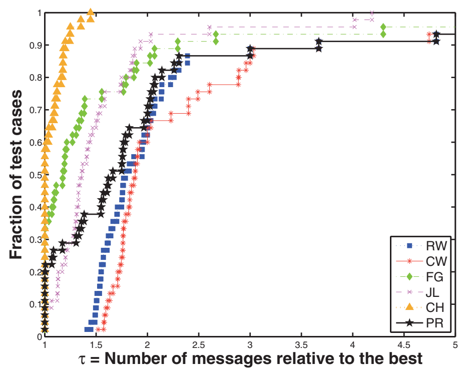
(e) square symmetric instances



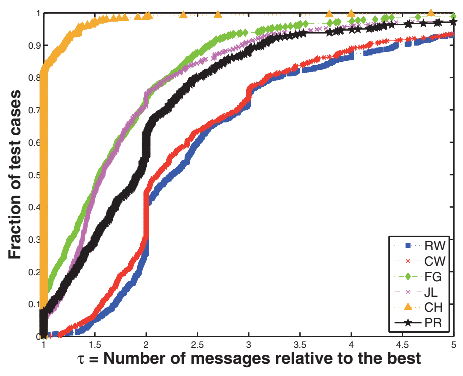
(f) all instances

FIG. 5.1. *Performance profile plots comparing the six partitioning methods (five base methods and partitioning recipe (PR)) using the total communication volume as the comparison metric:* (a) *rectangular partitioning instances where the number of rows is greater than the number of columns;* (b) *rectangular partitioning instances where the number of columns is greater than the number of rows;* (c) *all rectangular partitioning instances;* (d) *nonsymmetric square partitioning instances;* (e) *symmetric partitioning instances;* (f) *all partitioning instances.*

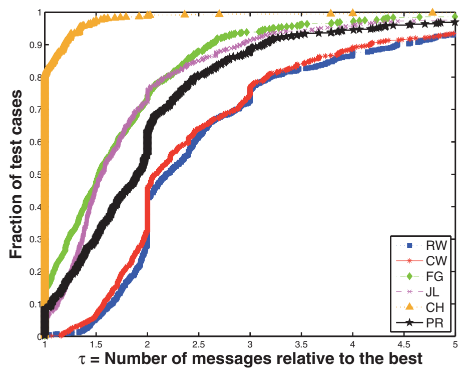
Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.



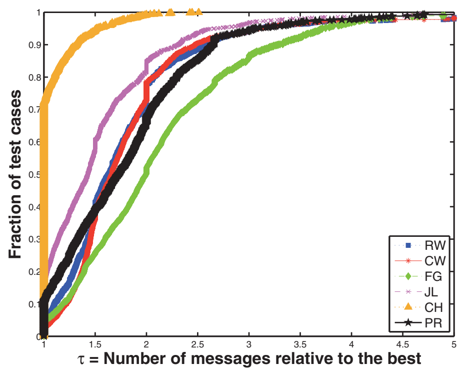
(a) rectangular instances $M > N$



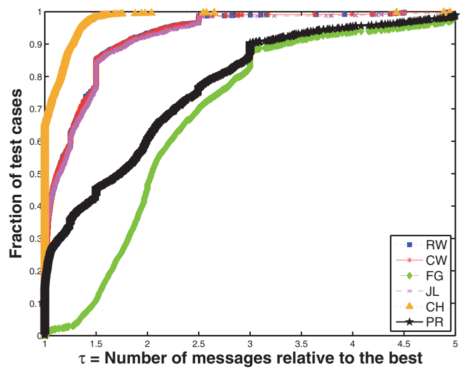
(b) rectangular instances $N > M$



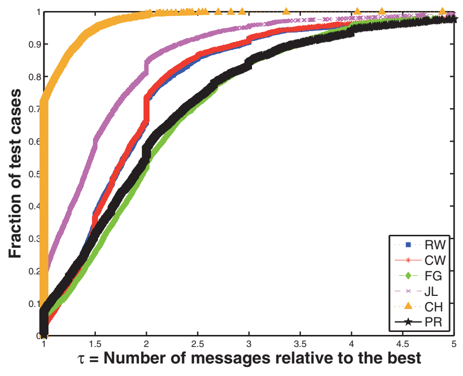
(c) all rectangular instances



(d) square nonsymmetric instances



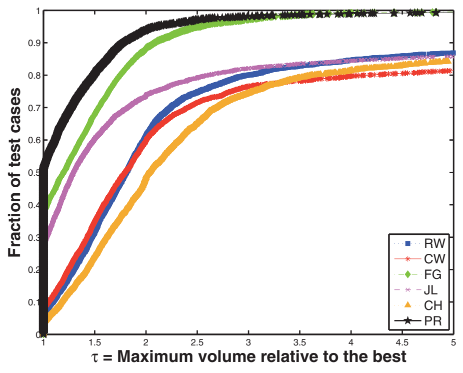
(e) square symmetric instances



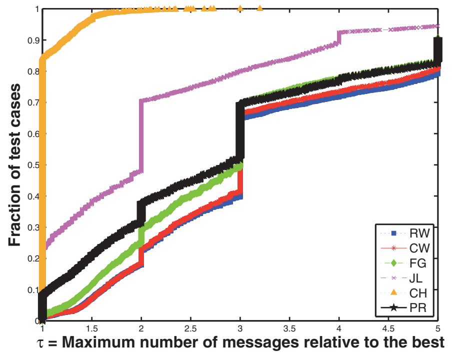
(f) all instances

FIG. 5.2. *Performance profile plots comparing the six partitioning methods using the total number of messages as the comparison metric:* (a) *rectangular partitioning instances where the number of rows is greater than the number of columns;* (b) *rectangular partitioning instances where the number of columns is greater than the number of rows;* (c) *all rectangular partitioning instances;* (d) *nonsymmetric square partitioning instances;* (e) *symmetric partitioning instances;* (f) *all partitioning instances.*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

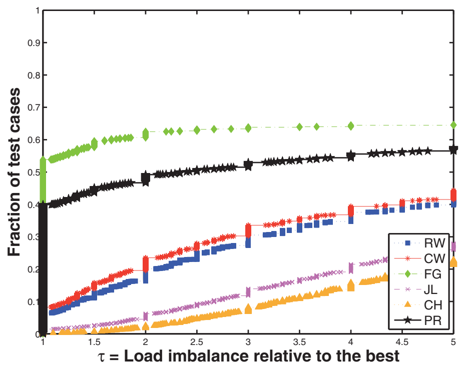


(a) Maximum volume

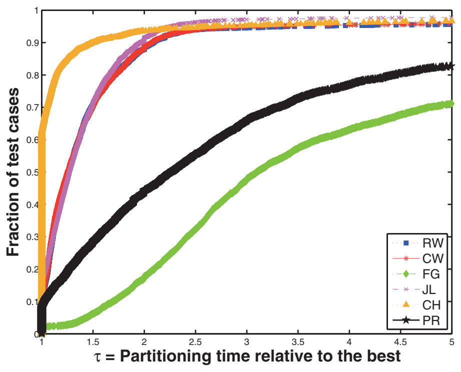


(b) Maximum number of messages

FIG. 5.3. *Performance profiles plot comparing the six partitioning methods using the maximum volume and number of messages per processor as the comparison metric.*



(a) Load balance



(b) Partitioning time

FIG. 5.4. *Performance profiles plot comparing the six partitioning methods using the load balance and partitioning time as the comparison metric on all partitioning instances.*

drastically limits the flexibility of the second step. Furthermore, assignments are done at a much coarser level; therefore those two methods produce the worst partitions in terms of the computational balance. The CH method performs worse than the JL method, because it maintains row and column coherences. The RW and CW methods are somewhere in between the FG method and the other two 2D partitioning methods CH and JL, but they are closer to the CH and JL methods than to the FG method. This happens because the granularity of the assignments limits the best balance that can be achieved. The individual plots for the symmetric, unsymmetric, and rectangular matrices are not shown because the trends are similar. The FG method is always the best, next comes the 1D partitioning methods, followed by the JL and then CH methods.

Comparison of the partitioning times is displayed in Figure 5.4(b). As expected, the FG method is the slowest of all. The RW, CW, and JL methods take more or less the same amount of time. An interesting result is that CH seems to be faster than the others in almost 90% of the instances. One might expect the execution time of the CH method to be comparable or even greater than that of the JL method—they have

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

TABLE 5.1
The number of times a method produced the best result in terms of the total communication volume. If two or more methods give the same best value, each one's score is incremented by one.

Method	$K = 4$	$K = 16$	$K = 64$	$K = 256$
RW sym	76	7	2	1
CW sym	62	8	3	2
FG sym	276	278	168	113
JL sym	158	159	111	69
CH sym	21	0	0	0
RW nonsym	139	52	26	9
CW nonsym	167	113	44	15
FG nonsym	508	481	310	179
JL nonsym	231	190	121	76
CH nonsym	38	0	0	0
JL^T nonsym	78	53	39	29
JL^T sym	31	13	26	10
CH^T nonsym	19	0	0	0
CH^T sym	5	0	0	0
PR	524	491	312	197

TABLE 5.2
The number of times a specific method was chosen by the partitioning recipe for 4,100 *partitioning instances.*

Method	$K = 4$	$K = 16$	$K = 64$	$K = 256$
FG sym	350	331	252	148
JL sym	296	272	176	107
RW nonsym	9	8	5	4
CW nonsym	94	82	52	26
FG nonsym	594	567	327	198
JL nonsym	31	23	16	12
JL^T nonsym	43	40	24	13

the same first step, but the second step of CH involves multi-constraint partitioning. The CH method partitions a larger hypergraph in the second step, whereas the JL method partitions $\sqrt{K}$ smaller hypergraphs. We think that the more restricted search space of the multi-constraint partitioning may result in such an outcome. Note that the individual plots for the symmetric, unsymmetric, and rectangular matrices are not shown, as the trends are again similar to those that are shown. Up until $\tau = 2$, the CH method is the fastest, followed by the almost equally fast RW, CW, and JL methods, with the slowest always being the FG method. After $\tau = 2$, all methods except FG are nearly equally fast, and FG is again the slowest.

5.5. Evaluation of the recipe. In this section, we try to see how successful the recipe is in reducing the total communication volume and addressing the other communication cost metrics. Tables 5.1 and 5.2 display the number of times a method produced the best result in terms of total communication volume, and the number of times a specific method was chosen by the partitioning recipe, respectively. As seen in Table 5.1, FG produces the best results in terms of the total communication volume metric in a considerably larger number of instances than any other method. Note that in this table, the best is defined in absolute terms; e.g., even if a method performs slightly better than the others, its score is incremented by one. If any two methods obtain the same best value, then each one's score is incremented by one. In other words, this table can be used to draw the performance profile of the methods at $\tau = 1$. We use Table 5.1 in order to highlight the choices made by the partitioning

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

recipe. Although selecting the FG method will most likely guarantee a better success (in terms of the total volume of communications metric) than any of the others, the recipe does not always choose the FG method; see Table 5.2. For example, for $K = 4$, it chooses FG in more than 900 instances, but it also chooses JL in more than 350 instances, and 1D partitionings (RW or CW) in more than 100 instances. We note that the recipe obtains the highest number of best results for any K shown in the tables.

As shown in Figure 5.1(f), the recipe achieves a performance similar to that of the FG method (in almost 90% of the instances, the results obtained by these two methods are within 1.2 of the best). Later we see that in almost 95% of the instances both results are within 1.4 of the best. Looking at Figures 5.2(d) and 5.2(e), we can see that the recipe achieves better total number of messages than the FG method. In Figure 5.2(f), the difference between the recipe and the FG method is not significant. This is because of two reasons: first, on rectangular matrices, the FG method obtains better results than the recipe; second, the recipe always chooses symmetric vector partitioning for the symmetric matrices, whereas FG is represented by the best of the symmetric and nonsymmetric vector partitioning variants. As seen in Figure 5.3(a), the recipe and the FG methods obtain the best performance in the maximum volume of a processor metric, with the recipe being more preferable up until $\tau = 2.5$. As seen in Figure 5.3(b), in terms of the metric of the maximum number of messages of a processor, the recipe demonstrates an average behavior, being inferior to the CH and JL methods while being superior to the others. Now consider, in return, Figures 5.4(a) and 5.4(b). As seen from these figures, the recipe is faster than the FG method, while the difference between the computational load balance performances are acceptable. In summary, the proposed recipe achieves performance similar to the FG method in terms of the total communication volume, while being more favorable or not too bad in terms of the other metrics, on average. Therefore, for a user of these methods, we suggest using the proposed recipe to choose a partitioning method for obtaining a good partitioning on average.

6. Conclusion.

We presented three hypergraph-based, 2D matrix partitioning methods, each having a unique advantage. The fine-grain partitioning method treats individual nonzeros as the smallest assignable elements and hence achieves a very fine-grain partitioning of matrices. This gives maximum flexibility in reducing the total communication volume and in balancing the computational loads at the expense of longer partitioning time and possibly higher total number of messages. The checkerboard partitioning method produces coarser partitionings and maintains row and column coherences at a coarse level by assigning rows and columns of the matrices to only a subset of processors. Hence, it imposes a smaller upper limit on the total number of messages and the maximum number of messages of a processor. Although more restricted partitioning may yield larger total communication volume, this method could be a good alternative for parallel architectures with high message latency. The jagged-like method trades the number of messages limit for a better communication volume.

We also presented a thorough experimental evaluation of the hypergraph-partitioning-based methods using 4,100 partitioning instances—a partitioning instance is defined as a pair of a matrix and an integer representing the number of parts. The experimentation clearly demonstrated the strengths and weaknesses of the proposed methods. The fine-grain model has the best performance in reducing the total communication volume and the maximum volume of communication (measured in terms of sends) of a processor; the checkerboard partitioning method has the best

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

performance in reducing the total number of messages and the maximum number of messages (measured in terms of sends) of a processor; the jagged-like method always strikes a balance between the communication cost metrics best reduced by the fine-grain and checkerboard methods. We further proposed an easy-to-use partitioning recipe that chooses one of the appropriate methods according to some matrix characteristics. The average behavior of the proposed recipe on the partitioning instances is such that, with 90% probability, it produces results within 1.2 of the best solution for the total volume of communication metric. Furthermore, the recipe achieves this performance while being faster than the partitioning method with the best performance on the total communication volume metric on average, and while respecting a symmetric vector partitioning for symmetric matrices. Although we are happy with the performance of the recipe, we expect more accurate ones to be developed; those that prioritize different communication cost metrics (the maximum number of messages of a processor, for example) according to different target parallel machine architectures and matrix sizes would be most useful.

REFERENCES

[1] C. Aykanat, B. B. Cambazoglu, and B. Uçar, *Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices*, J. Parallel Distrib. Comput., 68 (2008), pp. 609–625.

[2] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz, *Distributed processing of very large datasets with DataCutter*, Parallel Comput., 27 (2001), pp. 1457–1478.

[3] R. H. Bisseling and W. Meesen, *Communication balancing in parallel sparse matrix-vector multiplication*, Electron. Trans. Numer. Anal., 21 (2005), pp. 47–65.

[4] T. Bultan and C. Aykanat, *A new mapping heuristic based on mean field annealing*, J. Parallel Distrib. Comput., 16 (1992), pp. 292–305.

[5] W. Camp, S. J. Plimpton, B. Hendrickson, and R. W. Leland, *Massively parallel methods for engineering and science problems*, Commun. ACM, 37 (1994), pp. 31–41.

[6] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. Heaphy, and L. A. Fisk, *Hypergraph-based dynamic load balancing for adaptive scientific computations*, in Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS), 2007, IEEE.

[7] U. V. Çatalyürek, *Hypergraph Models for Sparse Matrix Partitioning and Reordering*, Ph.D. thesis, Bilkent University, Computer Engineering and Information Science, Ankara, Turkey, 1999, http://www.cs.bilkent.edu.tr/tech-reports/1999/ABSTRACTS.1999.html.

[8] U. V. Çatalyürek and C. Aykanat, *Decomposing irregularly sparse matrices for parallel matrix-vector multiplications*, in Proceedings of the 3rd International Symposium on Solving Irregularly Structured Problems in Parallel, Irregular'96, Lecture Notes in Comput. Sci. 1117, Springer-Verlag, New York, 1996, pp. 75–86.

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

[10] U. V. Çatalyürek and C. Aykanat, *PaToH: A Multilevel Hypergraph Partitioning Tool, Version* 3.0, Bilkent University, Department of Computer Engineering, Ankara, Turkey, 1999. PaToH is available at http://bmi.osu.edu/~umit/software.htm.

[11] U. V. Çatalyürek and C. Aykanat, *A fine-grain hypergraph model for* 2*D decomposition of sparse matrices*, in Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, 2001.

[12] U. V. Çatalyürek and C. Aykanat, *A hypergraph-partitioning approach for coarse-grain decomposition*, in ACM/IEEE SC2001, Denver, CO, 2001.

[13] C. Chang, T. Kurc, A. Sussman, U. V. Çatalyürek, and J. Saltz, *A hypergraph-based workload partitioning strategy for parallel data aggregation*, in Proceedings of the Eleventh SIAM Conference on Parallel Processing for Scientific Computing, SIAM, 2001.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, McGraw-Hill, New York, 1990.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

[15] T. Davis, *The University of Florida Sparse Matrix Collection*, Technical report REP-2007-298, CISE Department, University of Florida, Gainesville, FL, 2007.

[16] J. Dean and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters*, Commun. ACM, 51 (2008), pp. 107–113.

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

[18] R. Ferreira, T. Kurc, M. Beynon, C. Chang, A. Sussman, and J. Saltz, *Object-relational queries into multi-dimensional databases with the active data repository*, Parallel Process. Lett., 9 (1999), pp. 173–195.

[19] B. Hendrickson, *Graph Partitioning and Parallel Solvers: Has the Emperor No Clothes?*, Lecture Notes in Comput. Sci. 1457, Springer-Verlag, New York, 1998, pp. 218–225.

[20] B. Hendrickson and T. G. Kolda, *Graph partitioning models for parallel computing*, Parallel Comput., 26 (2000), pp. 1519–1534.

[21] B. Hendrickson and T. G. Kolda, *Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing*, SIAM J. Sci. Comput., 21 (2000), pp. 2048–2072.

[22] B. Hendrickson, R. Leland, and S. Plimpton, *An efficient parallel algorithm for matrix-vector multiplication*, Internat. J. High Speed Comput., 7 (1995), pp. 73–88.

[23] B. Hendrickson, R. Leland, and R. Van Driessche, *Skewed graph partitioning*, in Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computation, 1997.

[24] M. Jacunski, P. Sadayappan, and D. K. Panda, *All-to-all broadcast on switch-based clusters of workstations*, in Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, IPPS'99/SPDP'99, Washington, DC, 1999, IEEE Computer Society, pp. 325–329.

[25] M. Kaddoura, C. W. Ou, and S. Ranka, *Partitioning unstructured computational graphs for nonuniform and adaptive environments*, IEEE Parallel Distrib. Technol., 3 (1995), pp. 63–69.

[26] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

[27] G. Karypis and V. Kumar, *Multilevel Algorithms for Multi-constraint Graph Partitioning*, Technical report 98-019, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN, 1998.

[28] G. Karypis, V. Kumar, R. Aggarwal, and S. Shekhar, *hMeTiS: A Hypergraph Partitioning Package Version* 1.5.3, University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[29] S. Krishnamoorthy, U. Catalyurek, J. Nieplocha, A. Rountev, and P. Sadayappan, *Hypergraph partitioning for automatic memory hierarchy management*, in SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, 206, Tampa, FL, 98 (in cdrom).

[30] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin/Cummings, Redwood City, CA, 1994.

[31] H. Kutluca, T. M. Kurc, and C. Aykanat, *Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids*, J. Supercomputing, 15 (2001), pp. 51–93.

[32] V. Lakamsani, L. N. Bhuyan, and D. S. Linthicum, *Mapping molecular dynamics computations on to hypercubes*, Parallel Comput., 21 (1995), pp. 993–1013.

[33] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Willey–Teubner, Chichester, U.K., 1990.

[34] J. G. Lewis, D. G. Payne, and R. A. van de Geijn, *Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers*, in Proceedings of the Scalable High Performance Computing Conference, 1994, IEEE, Knoxville, TN, pp. 542–550.

[35] J. G. Lewis and R. A. van de Geijn, *Distributed memory matrix-vector multiplication and conjugate gradient algorithms*, in Proceedings of Supercomputing'93, Portland, OR, 1993, pp. 484–492.

[36] F. Manne and T. Sørevik, *Partitioning an array onto a mesh of processors*, in PARA '96: Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization, London, UK, 1996, also in Applied Parallel Computing Industrial Computation and Optimization, Lecture Notes in Comput. Sci, 1184, Springer-Verlag, New York, 1996, pp. 467–477.

[37] O. C. Martin and S. W. Otto, *Partitioning of unstructured meshes for load balancing*, Concurrency: Practice and Experience, 7 (1995), pp. 303–314.

[38] D. M. Nicol, *Rectilinear partitioning of irregular data parallel computations*, J. Parallel Distrib. Comput., 23 (1994), pp. 119–134.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

[39] A. T. Ogielski and W. Aiello, *Sparse matrix computations on parallel processor arrays*, SIAM J. Sci. Comput., 14 (1993), pp. 519–530.

[40] A. Pinar and C. Aykanat, *Fast optimal load balancing algorithms for 1D partitioning*, J. Parallel Distrib. Comput., 64 (2004), pp. 974–996.

[41] L. F. Romero and E. L. Zapata, *Data distributions for sparse matrix vector multiplication*, Parallel Comput., 21 (1995), pp. 583–605.

[42] J. H. Saltz, S. G. Petition, H. Berryman, and A. Rifkin, *Performance effects of irregular communication patterns on massively parallel multiprocessors*, J. Parallel Distrib. Comput., 13 (1991), pp. 202–212.

[43] K. Schloegel, G. Karypis, and V. Kumar, *A New Algorithm for Multi-objective Graph Partitioning*, Technical report 99-003, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN, 1999.

[44] K. Schloegel, G. Karypis, and V. Kumar, *Parallel multilevel algorithms for multi-constraint graph partitioning*, in Proceedings of the Euro-Par 2000 Parallel Processing, Munich, Germany, Lecture Notes in Comput. Sci. 1900, Springer-Verlag, New York, 2000, pp. 296–300.

[45] B. Uçar, *Heuristics for a matrix symmetrization problem*, in PPAM 2007, Lecture Notes in Comput. Sci. 4967, Springer-Verlag, New York, 2008, pp. 717–727.

[46] B. Uçar and C. Aykanat, *Minimizing communication cost in fine-grain partitioning of sparse matrices*, in ISCIS 2003, Lecture Notes in Comput. Sci. 2869, Springer-Verlag, New York, 2003, pp. 926–933.

[47] B. Uçar and C. Aykanat, *Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies*, SIAM J. Sci. Comput., 25 (2004), pp. 1837–1859.

[48] B. Uçar and C. Aykanat, *A Library for Parallel Sparse Matrix-vector Multiplies*, Technical report BU-CE-0506, Department of Computer Engineering, Bilkent University, Ankara, Turkey, 2005.

[49] B. Uçar and C. Aykanat, *Partitioning sparse matrices for parallel preconditioned iterative methods*, SIAM J. Sci. Comput., 29 (2007), pp. 1683–1709.

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

[51] B. Uçar, U. V. Çatalyürek, and C. Aykanat, *A matrix partitioning interface to PaToH in MATLAB*, Parallel Comput., to appear.

[52] 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.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.