A matrix partitioning interface to PaToH in MATLAB

Bora Uçar [a,*,1], Ümit V. Çatalyürek [b,2], and Cevdet Aykanat [c,3]

[a] *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*

[b] *Department of Biomedical Informatics and Department of Electrical & Computer Engineering, The Ohio State University, Columbus, Ohio, USA*

[c] *Department of Computer Engineering, Bilkent University, Bilkent, 06800, Ankara, Turkey*

Abstract

We present the PaToH MATLAB Matrix Partitioning Interface. The interface provides support for hypergraph-based sparse matrix partitioning methods which are used for efficient parallelization of sparse matrix-vector multiplication operations. The interface also offers tools for visualizing and measuring the quality of a given matrix partition. We propose a novel, multilevel, 2D coarsening-based 2D matrix partitioning method and implement it using the interface. We have performed extensive comparison of the proposed method against our implementation of orthogonal recursive bisection and fine-grain methods on a large set of publicly available test matrices. The conclusion of the experiments is that the new method can compete with the fine-grain method while also suggesting new research directions.

Key words: Matrix partitioning, hypergraph partitioning, sparse matrix-vector multiplication.

* Corresponding author, Tel: +33 (4) 72728932; Fax: +33(4)72728080
Email addresses: bora.ucar@ens-lyon.fr (Bora Uçar), catalyurek.1@osu.edu (Ümit V. Çatalyürek), aykanat@cs.bilkent.edu.tr (Cevdet Aykanat).

[1] The work of this author is partially supported by "Agence Nationale de la Recherche" through SOLSTICE project ANR-06-CIS6-010.

[2] The work of this author was supported in part by the National Science Foundation under Grants CNS-0643969, CCF-0342615, CNS-0403342 and DoE SciDAC Grant DE-FC02-06ER2775.

[3] The work of this author was supported in part by The Scientific and Technological Research Council of Turkey (TUBITAK) under project EEEAG-109E019.

1 Introduction

During the last decade, several successful hypergraph-based models and methods were proposed for sparse matrix partitioning [1–10]. These models and methods have gained wide acceptance in the literature for efficient parallelization of sparse matrix-vector multiply operations. There are three basic hypergraph models, namely, the column-net model [2], the row-net model [2], and the column-row-net (fine-grain) model [3,5]. The column-net and row-net models are respectively used for one-dimensional (1D) rowwise and 1D columnwise partitioning. The column-row-net model is used for 2D nonzero-based fine-grain partitioning. In all these models, the partitioning objective of minimizing the cutsize, which is defined over the hyperedges, exactly corresponds to minimizing the total communication volume. The partitioning constraint of maintaining a balance on part weights, which are defined over the vertex weights, corresponds to maintaining a computational load balance. Apart from the methods that partition the basic models, there are other methods such as the orthogonal recursive bisection [6], the jagged-like and the checkerboard partitioning methods [4,5] which utilize both the column-net and the row-net models for 2D partitioning.

MATLAB provides an ideal platform for quick prototyping of new numerical algorithms and for visualization of matrix-based data. Although a mesh partitioning toolbox was available [11], a comprehensive hypergraph-based matrix partitioning toolbox was missing. In this work, we provide the *PaToH Matrix Partitioning Interface* that utilizes PaToH [12] via the MATLAB mex function utility. Our toolbox not only provides an interface for various 1D and 2D matrix partitioning methods, but also offers partitioning visualization tools and supplementary codes to measure the quality of a given partition.

Among the various hypergraph-based partitioning methods [1–6], the fine-grain partitioning method [3,5] provides the most flexible partitioning technique by allowing assignment of individual nonzero elements to the processors. This degree of freedom comes with a high execution time during partitioning. In this paper, we propose yet another novel 2D matrix partitioning method that utilizes the fine-grain method more intelligently in the multilevel partitioning framework. The proposed approach uses a 2D coarsening method, in other words 1D coarsening of rows and columns, during the coarsening phase of the multilevel partitioning framework, whereas it uses a weighted fine-grain model for the initial partitioning and uncoarsening phases. The new coarsening method generates coarser approximations of the original matrix where, when projected back to the original matrix, the cutsize of a partitioning at the coarser matrix gives an upper bound on the cutsize. While going up through the matrix hierarchy towards the original matrix, this upper bound gets tighter, and can be refined by the flexible fine-grain model. We expect

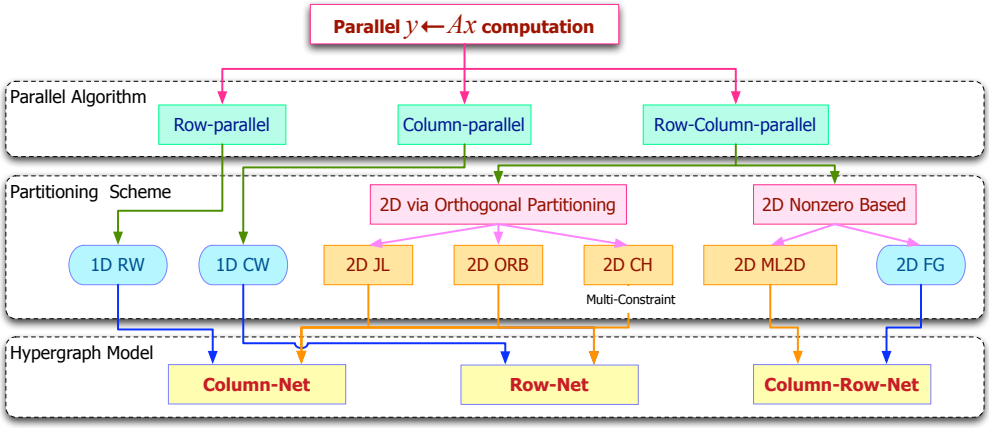


Fig. 1. A taxonomy for sparse matrix partitioning models and methods. There are three basic hypergraph models. Different partitioning methods use these models.

this approach to be faster than direct application of fine-grain model and to produce comparable results.

The rest of the paper is organized as follows: Section 2 presents the background material on existing hypergraph models and methods for 1D and 2D matrix partitioning schemes. Our PaToH MATLAB Matrix Partitioning Interface is presented in Section 3 together with a sample method, Orthogonal Recursive Bisection, that utilizes this interface to provide a different matrix partitioning method. Section 4 contains the detailed description of the proposed multilevel, 2D coarsening-based 2D matrix partitioning method. A comprehensive evaluation of the interface and the new method is presented in Section 5. Finally, we conclude in Section 6.

2 Background

Here, we present some background material on parallel sparse matrix-vector multiply algorithms and the hypergraph partitioning problem. We also give an overview of the hypergraph models and methods used for efficient parallelization of sparse matrix-vector multiply algorithms.

2.1 Parallel matrix-vector multiply algorithms

Figure 1 displays our taxonomy for the sparse matrix partitioning models and methods. As seen in the figure, partitioning models and methods can be classified as one-dimensional (1D) and two-dimensional (2D). The row-parallel and column-parallel matrix-vector multiply algorithms can be applied to the partitions resulting from the 1D rowwise and 1D columnwise schemes. The

row-column-parallel matrix-vector multiply algorithm can be applied to the partitions resulting from the 2D schemes. We will first briefly describe the most general nonzero-based row-column-parallel algorithm and then identify the row-parallel and column-parallel algorithms as special cases of the row-column-parallel algorithm.

Consider the matrix-vector multiply of the form $y \leftarrow A\,x$, where the nonzeros of the sparse matrix A as well as the entries of the input and output vectors x and y are partitioned arbitrarily among the processors. Let $map(\cdot)$ denote the nonzero-to-processor and vector-entry-to-processor assignments induced by this partitioning. The row-column-parallel algorithm executes the following steps at each processor P_k.

(1) Send the local input-vector entries x_j, for all j with $map(x_j) = P_k$, to those processors that have at least one nonzero in column j.
(2) Compute the scalar products $a_{ij}\,x_j$ for the local nonzeros, i.e., the nonzeros for which $map(a_{ij}) = P_k$ and accumulate the results y_i^k for the same row index i.
(3) Send local nonzero partial results y_i^k to the processor $map(y_i) \neq P_k$, for all nonzero y_i^k.
(4) Add the partial y_i^ℓ results received to compute the final results $y_i = \sum y_i^\ell$ for each i with $map(y_i) = P_k$.

As seen in the algorithm, it is necessary to have partitions on the matrix A and the input- and output-vectors x and y of the matrix-vector multiply operation. Finding a partition on the vectors x and y is referred to as the vector partitioning operation, and it can be performed in three different ways: by decoding the partition given on A [2]; in a post-processing step using the partition on the matrix [7,8,13]; or explicitly partitioning the vectors during partitioning the matrix [9]. In any of these cases, the vector partitioning for matrix-vector operations is called *symmetric* if x and y have the same partition, and *non-symmetric* otherwise. We say a vector partitioning is *consistent*, if each vector entry is assigned to a processor that has at least one nonzero in the respective row or column of the matrix. The consistency is easy to achieve for the nonsymmetric vector partitioning; x_j can be assigned to any of the processors that has a nonzero in the column j, and y_i can be assigned to any of the processors that has a nonzero in the row i. If a symmetric vector 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 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 the processor that holds the diagonal entry a_{ii}.

There are two communication phases in the parallel algorithm given above. The first one is at step 1 just before the local matrix-vector multiply, and it is due to the communication of the x-vector entries. We refer to this multi-cast like operation as *expand*. The second communication phase is at step 3 just after the local matrix-vector multiply, and it is due to the communication of the partial results on the y-vector entries. We refer to this operation as *fold*. As it can be seen from these two steps, row and column *coherences* are important factors in a matrix partition for efficient parallel computations. Column coherence relates to the fact that nonzeros on the same column require the same x-vector entry. Row coherence relates to the fact that nonzeros on the same row generate partial results for the same y-vector entry. Disturbing the column coherences incurs expand communication, and disturbing the row coherences incurs fold communication.

The one-dimensional rowwise partitioning incurs only expand communication, because it respects row coherence by assigning entire rows to processors while disturbing column coherence. Therefore, steps 3 and 4 will not exist in the row-parallel algorithm. 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, step 1 will not exist in the column-parallel algorithm. The communication requirements of the row-parallel and the column-parallel algorithms are discussed in [8,14], and the communication requirement of the row-column-parallel algorithm is discussed in [5]. The implementation details of all these algorithms can be found in [15].

2.2 Hypergraphs and hypergraph partitioning

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ consists of 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 in $\mathcal{V}$; these vertices are called the *pins* of n_j. The size of a net is equal to the number of its pins. The size of a hypergraph is defined as the total number of its pins, i.e., $|\mathcal{H}| = \sum_{n_j \in \mathcal{N}} |n_j|$. Weights can be associated with vertices and costs can be associated with nets. We use $w(v_i)$ and $c(n_j)$ to denote, respectively, the weight of vertex v_i and the cost of net $c(n_j)$.

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 non-empty, parts are pairwise disjoint, and the union of parts gives $\mathcal{V}$. The partitioning constraint is to maintain a balance criterion on part weights, i.e.,

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

In (1), W_k denotes the weight of part $\mathcal{V}_k$, i.e., $W_k = \sum_{v_i \in \mathcal{V}_k} w(v_i)$, W_{avg} denotes the average part weight, i.e., $W_{avg} = \sum_{v_i \in \mathcal{V}} w(v_i)/K$, and ε represents the predetermined, maximum allowable imbalance ratio.

In a partition Π of $\mathcal{H}$, a net that has at least one vertex in a part is said to connect that part. The *connectivity set* Λ_j of a net n_j is defined as the set of parts connected by n_j. The *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 partitioning objective is to minimize the cutsize defined over the cut nets. There are various cutsize definitions. The relevant cutsize definition for our purposes in this paper is as follows

$$cutsize(\Pi) = \sum_{n_j \in \mathcal{N}} c(n_j) \, (\lambda_j - 1) \; . \tag{2}$$

In (2), each cut net n_j adds the cost $c(n_j)(\lambda_j - 1)$ to the cutsize, whereas internal nets do not incur any cost. The hypergraph partitioning problem is known to be NP-hard [16].

A recent variant of the above problem is the multi-constraint hypergraph partitioning [4,17–20] 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 balance criterion associated with each weight.

The multilevel paradigm [21] has been successfully applied to hypergraph partitioning [2,17,19,22]. A multilevel hypergraph partitioning method consists of three phases: coarsening, initial partitioning and uncoarsening. In the coarsening phase, the original hypergraph is coarsened into a smaller hypergraph through a series of coarsening levels. At each coarsening level, highly coherent vertices are grouped into super-vertices of the next level, thus decreasing the sizes of the nets. In the initial partitioning phase, the coarsest hypergraph is partitioned using various heuristics. In the uncoarsening phase, the generated coarse hypergraphs are uncoarsened back to the original, flat hypergraph. At each uncoarsening level, the partition projected from the previous coarser level is improved using a refinement heuristic such as FM [23] or KL [24].

2.3 Hypergraph models for sparse matrix partitioning

Here, we describe the hypergraph-based matrix partitioning models and methods according to the taxonomy given in Fig. 1. We describe the existing ORB method in Section 3.2, because this method is used to show how our newly

developed PaToH MATLAB Matrix Partitioning Interface can be used to prototype new methods. In our discussion, we will consider the partitioning of an $M \times N$ matrix A with Z nonzeros. We note that only the sparsity pattern of matrix A is important in our discussion, i.e., the matrix A is always a $\{0,1\}$ matrix in the rest of the paper. We use the term "total communication volume" of a partition to refer to the total communication volume of the matrix-vector multiply operation resulting from the given partitions on a matrix and the input- and output-vectors.

2.3.1 One-dimensional partitioning

In the *column-net hypergraph model* [1,2] used for 1D rowwise partitioning, matrix A is represented as a unit-cost hypergraph $\mathcal{H}_{\mathcal{R}} = (\mathcal{V}_{\mathcal{R}}, \mathcal{N}_{\mathcal{C}})$ with $|\mathcal{V}_{\mathcal{R}}| = M$ vertices, $|\mathcal{N}_{\mathcal{C}}| = N$ nets, and $|\mathcal{H}_{\mathcal{R}}| = Z$ pins. In $\mathcal{H}_{\mathcal{R}}$, there exists one vertex $v_i \in \mathcal{V}_{\mathcal{R}}$ for each row i of matrix A. Weight $w(v_i)$ of a vertex v_i is equal to the number of nonzeros in row i. The name of the model comes from the fact that columns are represented as nets. That is, there exists one unit-cost net $n_j \in \mathcal{N}_{\mathcal{C}}$ for each column j of matrix A. Net n_j connects the vertices corresponding to the rows that have a nonzero in column j. That is, $v_i \in n_j$ if and only if $a_{ij} \neq 0$.

In the *row-net hypergraph model* [1,2] used for 1D columnwise partitioning, matrix A is represented as a unit-cost hypergraph $\mathcal{H}_{\mathcal{C}} = (\mathcal{V}_{\mathcal{C}}, \mathcal{N}_{\mathcal{R}})$ with $|\mathcal{V}_{\mathcal{C}}| = N$ vertices, $|\mathcal{N}_{\mathcal{R}}| = M$ nets, and $|\mathcal{H}_{\mathcal{C}}| = Z$ pins. In $\mathcal{H}_{\mathcal{C}}$, there exists one vertex $v_j \in \mathcal{V}_{\mathcal{C}}$ for each column j of matrix A. Weight $w(v_j)$ of a vertex $v_j \in \mathcal{V}_{\mathcal{R}}$ is equal to the number of nonzeros in column j. The name of the model comes from the fact that rows are represented as nets. That is, there exists one unit-cost net $n_i \in \mathcal{N}_{\mathcal{R}}$ for each row i of matrix A. Net $n_i \subseteq \mathcal{V}_{\mathcal{C}}$ connects the vertices corresponding to the columns that have a nonzero in row i. That is, $v_j \in n_i$ if and only if $a_{ij} \neq 0$.

The use of the row-net and column-net hypergraph models in 1D sparse matrix partitioning is described in [1,2]. It is shown in [1,2] that the partitioning objective (2) corresponds exactly to the total communication volume (with a consistent vector partitioning), and the partitioning constraint (1) corresponds to maintaining a computational load balance.

2.3.2 Two-dimensional partitioning: Fine-grain (column-row-net) model

In the *column-row-net hypergraph model* [3] used for 2D nonzero-based fine-grain partitioning, matrix A is represented as a unit-weight and unit-cost hypergraph $\mathcal{H}_{\mathcal{Z}} = (\mathcal{V}_{\mathcal{Z}}, \mathcal{N}_{\mathcal{RC}})$ with $|\mathcal{V}_{\mathcal{Z}}| = Z$ vertices, $|\mathcal{N}_{\mathcal{RC}}| = M + N$ nets and $|\mathcal{H}_{\mathcal{Z}}| = 2Z$ pins. In $\mathcal{V}_{\mathcal{Z}}$, there exists one unit-weight vertex v_{ij} for each nonzero a_{ij} of matrix A. The name of the model comes from the fact that both rows and columns are represented as nets. That is, in $\mathcal{N}_{\mathcal{RC}}$, there exist one unit-cost

row-net r_i for each row i of matrix A and one unit-cost column-net c_j for each column j of matrix A. The row-net r_i connects the vertices corresponding to the nonzeros in row i of matrix A, and the column-net c_j connects the vertices corresponding to the nonzeros in column j of matrix A. 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 in exactly two nets.

The use of the column-row-net model in 2D fine-grain sparse matrix partitioning is described in [3,5]. In this model, the column nets model the expand-communication phase, whereas the row nets model the fold-communication phase. Therefore, the column-row-net model, as the row-net and column-net models for 1D partitioning, encodes the total communication volume exactly [5], again with a consistent vector partitioning. The partitioning constraint defined over the unit-weight vertices corresponds to maintaining a computational load balance.

2.3.3 Two-dimensional partitioning: Jagged-like method

The jagged-like (JL) partitioning method [18] achieves 2D partitioning through two consecutive 1D partitioning steps. One of the steps models the expand phase using the column-net model, and the other step models the fold phase using the row-net model. Among the two alternative schemes, we discuss the one which models the expand phase in the first step and the fold phase in the second step. A similar discussion holds for the other alternative.

The JL method assumes that the K processors of the parallel system are organized as a $P \times Q$ virtual mesh, where $K = P \times Q$. In the first step, matrix A is partitioned rowwise into P parts using the column-net hypergraph model. This P-way rowwise partitioning step corresponds to partitioning the rows of matrix A among the P rows of the processor mesh. Therefore, this partition is decoded as inducing P submatrices each assigned to a distinct row of the processor mesh. Note that those P matrices have roughly equal number of nonzeros. In the second step, each of the P submatrices is independently partitioned columnwise into Q parts using the row-net hypergraph $\mathcal{H}_C$. The Q-way columnwise partitioning of a submatrix corresponds to partitioning the nonzeros of every row of that submatrix among the Q processors of the respective row of the processor mesh. In this way, the nonzeros in a row of matrix A are partitioned among the Q processors in a row of the processor mesh.

The JL method is described in [5,18] for 2D orthogonal partitioning of sparse matrices. The column-net model used for P-way rowwise partitioning in the first step exactly encodes the total communication volume to be incurred during the expand phase. The row-net models used for P independent Q-way columnwise partitionings in the second step exactly encode the total commu-

nication volume to be incurred during the fold phase. The balancing constraint adopted in the partitioning of the first step together with the balancing constraint adopted in each of the P partitionings of the second step ensures a computational load balance as described in [5].

We note that the MRD method [25] also obtains a jagged partition. We discuss again the method which performs a P-way partitioning in the first step and P many Q-way partitionings in the second step. For a $P \times Q$ mesh of processors, MRD proceeds by partitioning the matrix first into P row stripes and then by partitioning each stripe vertically into Q column stripes independently. In the resulting partition, the splits span the entire matrix in one dimension, while they are jagged in the other one. There are two main differences between the MRD and the JL methods. First, MRD's aim is to achieve balance on processor loads, and it does not explicitly address the minimization of the communication cost metrics (an implicit upper bound exist because of the resulting structured partitioning); whereas in JL method we use hypergraph models to explicitly minimize the total communication volume, while also aiming load balance. Second, in the MRD method, the initial orders of the rows and the columns are kept intact, and each of the Q independent partitioning steps are performed with the same ordering; whereas in the JL method these orders are not respected, and each of the Q independent partitioning steps may result in a different ordering of the columns, when the columns are mapped to the columns of the processor mesh. For these two differences, the method we proposed [5,18] is called the jagged-like method. We also note that due to this reason, the splits defining the partition of the JL method cannot be overlaid with the matrix in its entirety.

2.3.4 Two-dimensional partitioning: Checkerboard method

As the JL method, the checkerboard partitioning (CH) method [4] is also a two-step method, in which each step models either the expand phase or the fold phase. Similar to JL, it assumes that the processors are organized as a $P \times Q$ mesh, and therefore can be performed in two alternatives schemes. We discuss the one which models the expand phase in the first step and the fold phase in the second step.

The first step of the CH method is exactly the same as that of the JL method. That is, the matrix A is partitioned rowwise into P parts using the column-net hypergraph model. In the second step, the matrix A is partitioned columnwise into Q parts using the row-net model with a multi-constraint partitioning formulation. Unlike the JL method, the whole matrix A is partitioned in the second step. The multi-constraint formulation adopted in the second step is needed to achieve computational load balance. In the hypergraph $\mathcal{H}_{\mathcal{C}}$ used in the second step, each vertex v_i is associated with P weights according to the

distribution of the nonzeros of column i among the P submatrices induced by the rowwise partition obtained in the first step.

The CH method is described in [4,5] for 2D orthogonal partitioning of sparse matrices. The column-net and row-net models used in the first and second steps exactly encode the total communication volume to be incurred during the expand and fold phases, respectively. The balancing constraint adopted in the partitioning of the first step together with the multi-constraint formulation adopted in the partitioning of the second step ensures a computational load balance as described in [5].

The matrix partition resulting from the CH method is Cartesian in the sense that the rows and the columns are partitioned, respectively, into P and Q sets, and the Cartesian products of these sets are mapped to the $P \times Q$ processor mesh. In other words the orders of the columns (rows) in each row (column) stripe are the same. Compared to the JL method, the CH method thus produces partitions whose boundaries can be overlaid with the given matrix.

3 PaToH MATLAB Matrix Partitioning Interface

Here, we briefly present the design of the main components of the matrix partitioning interface; a manual is available [26]. Then, we will demonstrate how a new partitioning algorithm can be developed using the basic components of the interface.

3.1 Main components

The two main components of the interface are `PaToHMatrixPart` and `PaToH` which have the following signatures.

```
[nnzpv, outpv, inpv] = PaToHMatrixPart(A, K, mth);
[partv, ptime] = PaToH(hyp, K, nconst, vw, nc, imbal);
```

Here, `A` is the sparse matrix to be partitioned, `K` is the number of parts and `mth` is a string specifying the partitioning method of the choice. The `mth` can be any of the strings shown in Table 1. The function `PaToH` lies at the core of `PaToHMatrixPart` and enables direct PaToH library calls from MATLAB. In this function, `hyp` is the hypergraph given as a sparse matrix (assuming the column-net model), `K` is the number of parts, `nconst` is the number of vertex weight constraints (used in multi-constraint partitioning), `vw` is the array of vertex weights, `nc` is the array of net costs, and `imbal` is the maximum allowed imbalance ratio ε of (1).

Table 1
Partitioning methods provided in the matrix partitioning interface. Each method can be asked to produce a symmetric or non-symmetric vector partitioning.

Partitioning method	vector partitioning	
	symmetric	non-symmetric
Rowwise	`RWS`	`RWU`
Columnwise	`CWS`	`CWU`
Fine-grain	`FGS`	`FGU`
Jagged-like	`JLS`	`JLU`
Checkerboard	`CHS`	`CHU`

Given the matrix `A`, the number of parts `K`, and the partitioning method `mth`, the function `PaToHMatrixPart` calls the appropriate partitioning method to partition the matrix. The output `nnzpv` is an $M \times N$ sparse matrix whose nonzero entries define the processor assignment for each nonzero of the matrix `A`, i.e., the number of nonzeros in those two matrices are the same. The output `outpv` is an $M \times 1$ array, and it defines the partition on the output-vector y of the $y \leftarrow Ax$ operation. The output `inpv` is an $N \times 1$ array, and it defines the partition on the input-vector x of the same operation. As zero is not a valid value for the nonzeros of a sparse matrix, we use numbers from 1 to `K` to denote part numbers.

We think that returning the partitioning information in the three output arguments `nnzpv`, `outpv`, and `inpv` is convenient and useful. Convenient because it defines the partitioning on all components of the sparse matrix-vector multiply operation. Useful, because the three outputs all together enable visualization of the partition. Consider the example, generated using the function `PaToHSpy` (for a detailed description of the function we refer the reader to the manual [26]), given in Fig. 2, which is obtained by a 4-way, `FGS` partitioning of a matrix using the function `PaToHMatrixPart`. As seen in the figure, the matrix is permuted into a 4×4 block form. The function `PaToHSpy` permutes the rows according to the partition `outpv` in such a way that a row i, where `outpv(i)=k`, is ordered before any other row j where `outpv(j)` is bigger than `k`. The order of rows within a block is arbitrary. A similar permutation is applied to the columns using the `inpv`. Hence, this block form indicates the vector partitioning such that the input and output-vector elements of x and y aligned with the first diagonal block belong to processor 1; those aligned with the second diagonal block belong to processor 2 and so on. The owner of other nonzeros can be inferred from the shapes or colors.

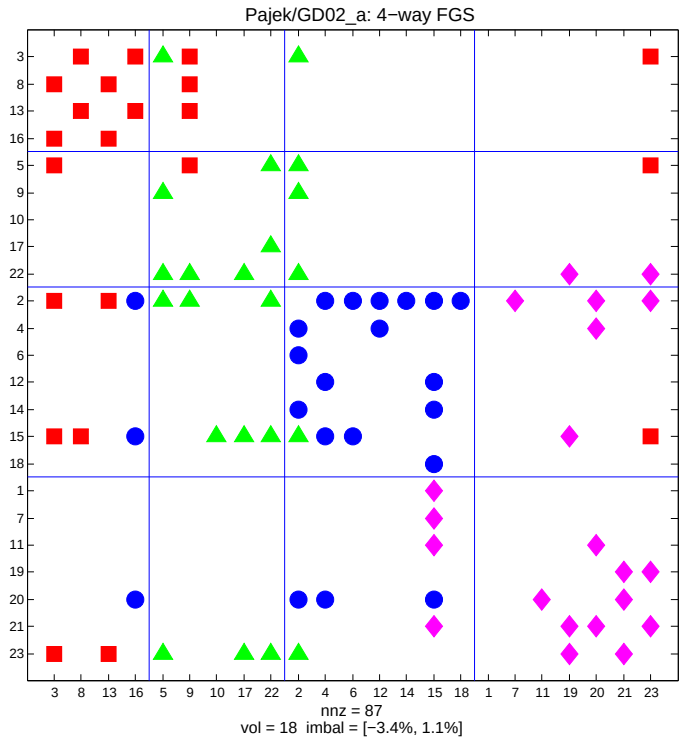


Fig. 2. A sample 4-way partitioning of the Pajek/GD02_a matrix from the UFL collection [27]. The 2D partitioning is obtained by using fine-grain method by running the `PaToHMatrixPart` function and displayed by using the `PaToHSpy` function.

3.2 Developing partitioning algorithms

Orthogonal recursive bisection (ORB) has initially been proposed for partitioning non-uniform two-dimensional computational domains into rectangular units of equal computational requirements [28,29]. In this approach, the domain is first partitioned vertically into two blocks of equal work. Then each subblock is further partitioned horizontally into two, again each having equal work. The process continues by partitioning alternately into horizontal and vertical blocks until the desired number of partitions is obtained.

The ORB method is also applied to partitioning sparse matrices for parallel sparse matrix-vector multiplication operation [6]. In this approach, the matrix is first partitioned rowwise into two submatrices using the column-net hypergraph model [2], and then each part is further partitioned columnwise into two parts using the row-net hypergraph model [2]. The process is continued recursively until the desired number of parts is obtained. It is also suggested to partition both rowwise and columnwise at each recursion step and to continue with the best of the two. In this paper, we follow a similar but more simplistic approach. For rectangular submatrices, we partition along the longer dimension. For the square submatrices, we compute both the rowwise and

columnwise partitions and choose the one with the best volume (in case of ties, we choose the one which gives a better balance among the two parts) for the current bisection. Therefore, our ORB implementation is different than Mondriaan [6] mainly due to the way we choose the partitioning direction. We note that another alternative [30] is to try the fine-grain partitioning as well at each recursive step and to choose the best among the three alternatives.

Algorithm 1 Orthogonal recursive bisection based partitioning

```
function [pvr, pvc, pids] = orbPartAux(Klow, Kup, rlist, clist)
m = length(rlist);
n = length(clist);
submat = A(rlist, clist);
if(m<n),
    [npv1, opv1, ipv1] = PaToHMatrixPart(submat, 2, 'CWU');
    whichnpv = npv1;
    rlistLeft = rlist;
    rlistRight = rlist;
    clistLeft =  clist(ipv1 == 1);
    clistRight = clist(ipv1 == 2);
elseif(m>n)
    [npv2, opv2, ipv2] = PaToHMatrixPart(submat, 2, 'RWU');
    whichnpv = npv2;
    rlistLeft  = rlist(opv2 == 1) ;
    rlistRight = rlist(opv2 == 2) ;
    clistLeft =  clist;
    clistRight = clist;
else
   %Skipped in the presentation, the above two bisections
   %are applied and the best one is choosen.
end
if(Kup-Klow == 1),%Base case
    [submrows,submcols, submnnzpv] = find(whichnpv);
    pvr  = rlist( submrows );
    pvc  = clist( submcols );
    pids  =  submnnzpv' + Klow-1;
else              %Recursive calls
    kmid = (Kup-Klow+1)/2;
    [pvrLeft, pvcLeft, pidsLeft ] =...
        orbPartAux(Klow, Klow+kmid-1, rlistLeft, clistLeft);
    [pvrRight, pvcRight, pidsRight] =...
        orbPartAux(Klow+kmid, Kup, rlistRight, clistRight);
    %Combine
    pvr = [pvrLeft, pvrRight];
    pvc = [pvcLeft, pvcRight];
    pids = [pidsLeft, pidsRight];
end
```

We show the implementation, almost complete, in Algorithm 1. The function

`orbPartAux` is implemented as a nested function within the scope of

`function [outpv, inpv, nnzpv] = PaToHorbPart(mat, K, dim).`

Here, `mat` is a given original matrix. From `mat`, we obtain the pattern matrix `A` using `A = spones(mat)` and add missing diagonal entries if a symmetric partitioning (`dim='ORS'`) is requested. The nested function `orbPartAux` is then called with inputs `orbPartAux(1, K, 1:orgm, 1:orgn)` where `orgm` and `orgn` correspond to the row and column sizes of the original matrix. The function `orbPartAux` then creates a submatrix of `A` using the `rlist` and `clist`, initially corresponding to all of the rows and the columns of `A`.

In the implementation, we did not filter out any possibly empty rows or columns from the submatrix `submat` constructed within the function `orbPartAux`. Therefore, the invocations to `PaToHMatrixPart` can have matrices with empty rows or columns.

4 A novel two-dimensional partitioning method

A noteworthy feature of the multilevel hypergraph partitioning heuristics is that the rate of decrease in the number of nets is, in general, much smaller than that of the number of vertices. Although this is desirable for the KL- and FM-based refinement heuristics, the presence of a large number of nets slows down the partitioning process.

In the algorithms that use 1D partitioning models, the coarsening amounts to folding the matrix along the partitioning dimension, along the rows or columns, while keeping the other dimension, columns or rows respectively, intact. In the fine-grain case, the nonzeros are matched; the row- and column-nets corresponding to the rows and the columns of the original matrix remain intact. The fine-grain partitioning case can be visualized on a larger $Z \times (M + N)$ matrix whose rows correspond to the nonzeros of the original matrix, and whose columns correspond to the set of rows and columns of the original matrix. Then, coarsening will again result in folding the matrix along the rows while keeping the columns intact. Note that in any of these cases nets disappear in the subsequent levels only if they become of size one during the coarsening phase. Also note that the coarsening in these three cases works pattern-wise, e.g., in the rowwise partitioning case a coarse row has a nonzero of value 1 in a column if either of its constituent rows (of the previous matrix) has a nonzero in that column, regardless of the current level of coarsening.

Here, we propose a novel multilevel coarsening approach whose main idea can be perceived as folding the matrix both along the rows and the columns. In

this way, the number of rows, columns, and nonzeros will decrease roughly at similar rates during the levels of the coarsening phase. This 2D coarsening approach will be utilized to develop a multilevel matrix partitioning algorithm which enables the use of the FG model in the initial partitioning and uncoarsening phases. As reported in [5], the fine-grain partitioning method, because of the higher degree of freedom inherent in the underlying model, is the most competent method among those discussed in Section 2.3 in reducing the total communication volume. However, it suffers from a high partitioning time, mainly because of the large number of time-consuming coarsening levels. The proposed 2D partitioning approach attempts to benefit from the refinement capability inherent in the fine-grain model, while trying to obtain an effective, and in the meantime, faster coarsening operation.

Our folding approach considers the initial matrix as a $\{0,1\}$-matrix, but performs addition operation while folding the matrix. In other words, the gist of the proposed method is to build a hierarchy of integer matrices resulting from a sequence of both rowwise and columnwise coarsenings, each time adding up the values of the nonzeros. Figure 3 illustrates the concept of the 2D coarsening approach. On the top, we have an 11×11 matrix, referred to as $A^{(0)}$. The matrix is coarsened for two levels yielding the 3×4 matrix at the bottom. A matching among the rows of $A^{(0)}$ is found and represented as a 6×11 matrix $P^{(1)}$. In $P^{(1)}$ each column has a single nonzero, mapping a row of $A^{(0)}$ to a super-row in the coarser matrix. Therefore, two columns having their nonzeros in a common row represent the match of the associated rows of $A^{(0)}$. Similarly the matrix $Q^{(1)}$ represents the matching among the columns of $A^{(0)}$. Any row i in $P^{(1)}$ having only one nonzero signifies that the row of $A^{(0)}$ corresponding to the nonzero column in row i of $P^{(1)}$ is not matched and left as singleton. A column of $Q^{(1)}$ with only one nonzero signifies the same result for the associated column of $A^{(0)}$. Thus, the coarser matrix $A^{(1)}$ can be computed by multiplying $A^{(0)}$ with $P^{(1)}$ from the left and with $Q^{(1)}$ from the right, i.e., $A^{(1)} = P^{(1)}A^{(0)}Q^{(1)}$. Hence, a nonzero entry in $A^{(1)}$ represents a set of entries in $A^{(0)}$ which are amassed by the rowwise and columnwise folding operations. For example, consider the nonzero $a^{(1)}_{2,3}$ of $A^{(1)}$ in Fig. 3. As seen in the figure, the super-row 2 of $A^{(1)}$ contains rows 2 and 7 of $A^{(0)}$, and the super-column 3 of $A^{(1)}$ contains columns 4 and 8 of $A^{(0)}$. Therefore, $a^{(1)}_{2,3}$ becomes 4 as the rows 2 and 7 both have nonzeros in the columns 4 and 8.

The 2D coarsening approach is not easily realizable in the hypergraph partitioning tools like PaToH [12] and Mondriaan [6]. The closest approach available in these libraries is the coarsening operation on a fine-grain hypergraph model. As there is no distinction between row nets and column nets in these libraries, it would not be possible to match, more precisely, to cluster nonzeros in two rows or in two columns. We note, however, that in a symmetric matrix, the 2D coarsening approach works like the coarsening operation in the multilevel graph partitioning methods (see [31] for coarsening in graph partitioning), if

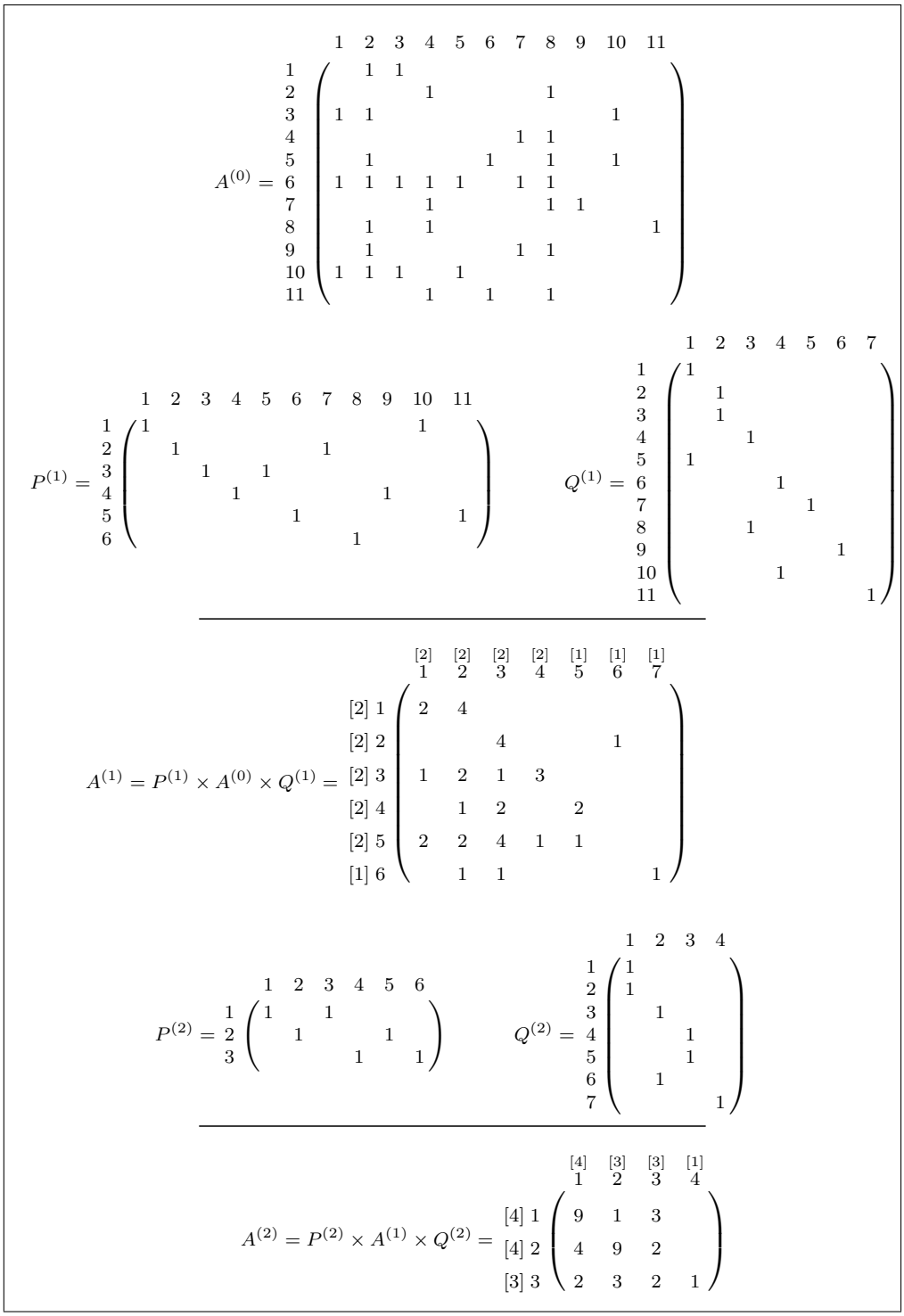


Fig. 3. Two-dimensional coarsening operation on an 11×11 matrix, `Tina_AskCog`, from the UFL collection [27]. The row and column indices are shown. The values in brackets denote the number of columns/rows of the flat matrix $A^{(0)}$ represented by the respective row/column of the coarse matrix.

the matching of the rows and the matching of the columns are found using the same deterministic algorithm.

As the conceived 2D coarsening approach is not realizable within the PaToH library, we have implemented a recursive bisection based, multilevel method. For the recursion, we use the skeleton of Algorithm 1 of Section 3.2. This time, however, instead of providing the matrix and the list of rows and columns for partitioning, we pass the row and column indices of the nonzeros that will be partitioned. At the beginning of each invocation, a sparse matrix is built using the given nonzeros. Then, a multilevel algorithm with all three phases—coarsening, initial partitioning, and uncoarsening—is called for bisecting the so-built matrix. We found it very convenient to formulate the coarsening and uncoarsening operations in terms of matrix-matrix multiply operations. Below, we describe the three phases of the multilevel bisection algorithm, highlighting the matrix-matrix multiplication based formulations.

4.1 Coarsening

A significant component of the coarsening phase is a function that finds a matching for the rows and another matching for the columns of a given matrix. In this paper, we propose the use of the cosine similarity as the matching criterion. The cosine similarity of two real vectors x and y is defined as the angle θ, where

$$\cos\theta = \frac{x^T y}{\|x\|\|y\|} \ . \tag{3}$$

Here, $x^T y$ is the inner product of the two vectors, i.e., $\sum x_i y_i$, and $\|\cdot\|$ is the magnitude of a vector, i.e., $\|x\| = \sqrt{\sum x_i^2}$ and $\|y\| = \sqrt{\sum y_i^2}$. The smaller the θ between two vectors, the more similar they are. We note that the heavy connectivity matching [12] or the inner product matching [6] use the inner product $x^T y$ as the similarity metric. There is, however, a subtle difference: the vectors x and y are always $\{0,1\}$-vectors in these two latter alternatives regardless of the coarsening level. That is, the inner product metric has been used in the existing hypergraph partitioning libraries only with $\{0,1\}$-vectors.

For a given row i, let $\mathcal{C}(i)$ be the set of columns in which row i has nonzeros. Similarly for a given column j, let $\mathcal{R}(j)$ be the set of rows in which column j has nonzeros. For a given unmatched row i, all unmatched rows in $\{r : r \in \mathcal{R}(j) \text{ for some } j \in \mathcal{C}(i)\}$ form the set of possible mates for the row i. The cosine similarity between row i and each such row is computed using (3) and the row r with the smallest θ is chosen as the mate of row i, i.e., `P(i)=r` and `P(r)=i`. As we are working with nonnegative vectors, this amounts to choosing an available row with the maximum value of $\cos\theta$. We have implemented this matching algorithm for the rows of a matrix as a MATLAB mex

function. The matching Q for the columns of a matrix can easily be obtained by invoking that function on the transpose of the matrix. For now, we process the rows and columns in descending order of number of nonzeros, but we intend to investigate alternatives. Furthermore, we apply a threshold test to the similarity: if $\cos\theta < \vartheta$, for a given ϑ we do not apply the matching.

We have implemented a function PaToHMatch(A) that carries out a matching according to the cosine similarity metric. The function PaToHMatch(A) gets a matrix A and computes two arrays P and Q, where P defines a matching among the rows, and Q defines a matching among the columns of A. The matching P is defined in such a way that if two rows i and j are matched, then P(i)=j and P(j)=i. If a row is not matched, then P(i)=i. The matching Q of the columns is defined similarly. From the matching arrays P and Q, we obtain the coarsening operators PHier{level} and QHier{level} at the level level. These two operators are sparse matrices and are constructed as follows. For a given $M \times N$ matrix A, the matrix PHier{level} has a column dimension of M. Suppose we have obtained m_1 matchings in rows, i.e., the total number of matched pairs i and j (that is, P(i)=j and P(j)=i) plus the number of singletons (that is, P(i)=i) is m_1. Then, the row dimension of PHier{level} is m_1, each row representing a matching or a singleton. If, for example, i and j are matched, then there is a row in PHier{level} with 1's in columns i and j. Similarly, the matrix QHier{level} has a row dimension of N. Suppose we have obtained n_1 matchings in the columns, i.e., the total number of matched pairs j and k (that is, Q(j)=k and Q(k)=j) plus the number of singletons (that is Q(j)=j) is n_1. Then, the column dimension of QHier{level} is n_1, each column representing a matching or a singleton. If, for example, the column k of A is left as singleton, then there is a column in QHier{level} with one nonzero entry at row k. The coarsened matrix Alevel is then of size $m_1 \times n_1$ and can be computed by multiplying A from the left by PHier{level} and from the right by QHier{level}.

The multilevel algorithm that carries out the two-dimensional coarsening on a matrix A is shown in Algorithm 2. We initialize levelMax=10 to allow at most 10 levels of coarsening. During coarsening if either the number of rows or the number of columns of the current matrix falls below 1000, we stop the coarsening phase. Since we apply a threshold test for matching two rows or columns, we also stop the coarsening if the contraction in a level does not reduce the number of rows and the number of columns significantly (we use a ratio of 7/8 as seen in the algorithm). As described before, the matchings are computed as arrays with the function PaToHMatch and then converted to sparse matrices in the function PaToHMLCoarseOperator. The matrices corresponding to the coarsening operators are stored in the structures PHier and QHier.

Consider the example shown in Fig. 3 (for simplicity, we do not apply the

Algorithm 2 Multilevel two-dimensional coarsening algorithm

```
function [PHier,QHier,nlevels,ACoarsest] = PaToHMLCoarse(A, levelMax)
[P,Q] = PaToHMatch(A);
[PHier{1}, QHier{1}] = PaToHMLCoarseOperator(A, P, Q);
Alevel = A;
for level = 2:levelMax,
    PMult = PHier{level-1};
    QMult = QHier{level-1};
    Alevel = PMult * Alevel * QMult;
    [P, Q] = PaToHMatch(Alevel);
    [PHier{level}, QHier{level}] = PaToHMLCoarseOperator(Alevel, P, Q);
    if(size(PHier{level},1) < 1000 || size(QHier{level},2) < 1000),
        break;
    end
    if (size(PHier{level},1)/size(PHier{level-1},1) >=7/8 &&
        size(QHier{level},2)/size(QHier{level-1},2) >=7/8 ),
        break;
    end
end
nlevels = level;
ACoarsest = PHier{level}*Alevel*QHier{level};
```

threshold test on the similarity of two rows or columns in this example). The given initial matrix $A^{(0)}$ has 36 nonzeros, each having a value 1. The coarsening operators `PHier{1}` and `QHier{1}` are shown, respectively, as $P^{(1)}$ and $Q^{(1)}$, and they are found by converting the matching arrays

```
P=[10 7 5 9 3 11 2 8 4 1 6 ]
Q=[ 5 3 2 8 1 10 7 4 9 6 11]
```

into matrices. The coarsened matrix $A^{(1)}$ is shown in the same figure. As seen in the figure, the sum of the nonzero values of $A^{(1)}$ is again 36. The coarsening matrices `PHier{2}` and `QHier{2}` are shown, respectively, as $P^{(2)}$ and $Q^{(2)}$, and they are found by converting the matching arrays

```
P=[3 5 1 6 2 4  ]
Q=[2 1 6 5 4 3 7]
```

of the rows and columns of $A^{(1)}$. The coarsest matrix $A^{(2)}$ is shown at the bottom of the figure. The sum of the values of the nonzeros in $A^{(2)}$ is 36. In the figure, each row and column of the matrices $A^{(1)}$ and $A^{(2)}$ is tagged with the number of constituting rows and columns.

The proposed coarsening approach makes use of the 1D row-net and column-net hypergraph models augmented with weights representing the nonzero values of the coarser matrices. We note that those weights seem to be associated

with the pins of the 1D hypergraphs—a counter-intuitive relationship since a pin of a hypergraph only represents a membership. We store the weights twice, in two different arrays. The first one is aligned with the vertex lists of nets, and the second one is aligned with the net lists of vertices. In a typical coarsening scheme used within the context of the 1D hypergraph models, the nets of a vertex v are visited in turn, and the vertices of each such net are considered as a potential mate for v. In our approach, while doing the same operation for a vertex v, we accumulate the inner product by using the two copies of the weights with a constant additional time per each pin accessed. As a preprocess, we compute the norms of the vectors corresponding to the vertices of a hypergraph in $O(Z+M)$-time, where the hypergraph corresponds to the column-net hypergraph of a matrix with M rows and Z nonzeros. As the weights are used while accessing the pins of a hypergraph, they do not affect the run-time complexity. Therefore, the proposed coarsening operation is akin to the coarsening operation in 1D hypergraph models.

4.2 Initial partitioning

The aim in this phase is to find a partition of the coarsest matrix `ACoarsest`. If this matrix were not available, it could be constructed using the hierarchy of `nlevels` of coarsening operators `PHier` and `QHier`. This can be accomplished by multiplying the original matrix `A` from the left by the product of the row coarsening matrices `PMult` and from the right by the product of the column coarsening matrices `QMult`. Note that the number of rows and columns of `PMult` are equal to, respectively, the number of rows of `ACoarsest` and the number of rows of the initial matrix `A`. Similarly, the number of rows and columns of `QMult` are equal to, respectively, the number of columns of the initial matrix `A` and the number of columns of `ACoarsest`.

To partition `ACoarsest`, we construct its fine-grain hypergraph model $\mathcal{H}_{\mathcal{Z}}$, and assign weights to vertices such that each vertex gets a weight equal to the value of the corresponding nonzero of `ACoarsest`. As the rows and the columns of `ACoarsest` are themselves amalgamations of the rows and columns of the original matrix `A`, it is necessary to assign a cost to each net. The cost of the row nets, `rnc`, and the column nets, `cnc`, of $\mathcal{H}_{\mathcal{Z}}$ can be computed by applying the coarsening operators to the vectors of 1 of appropriate size.

The fine-grain hypergraph $\mathcal{H}_{\mathcal{Z}}$ is represented as a matrix `hyp` whose rows correspond to the vertices of $\mathcal{H}_{\mathcal{Z}}$, and whose columns correspond to the row nets and the column nets of $\mathcal{H}_{\mathcal{Z}}$. With a call to the mex interface of PaToH

```
pv = PaToH(hyp, 2, 1, vw, [rnc, cnc]);
```

we compute the partitioning of the nonzeros of the coarsest matrix. We note

that at this point the cutsize of the partition will not be equal to the cutsize of partition of the original matrix, as the coarsening operation does not necessarily match two rows or two columns of the same sparsity pattern.

Consider the coarsest matrix $A^{(2)}$ of Fig. 3. A partition of the nonzeros of this matrix is shown on the top of Fig. 4 along with the costs of the row nets and the column nets in the brackets. The nonzeros in the columns 1 and 4 are assigned to processor 1, and the rest are assigned to processor 2. The loads of the processors are 16 and 20, respectively. Each row is split between the two processors. Therefore, the total communication volume is 11=4+4+3. Note that this would be an upper bound on the total communication volume, if the partition is projected to $A^{(0)}$ without any refinement.

4.3 Uncoarsening and refinement

Given an initial partitioning `pv` on the nonzeros of the coarsest matrix, the function `PaToHMLUnCoarse` shown in Algorithm 3 projects the partitioning on the coarsest matrix to the finer one, and refines the projected partition. The function initializes `nnzpv` to be equivalent to the initial partition on the coarsest matrix. Then, the partitioning information `nnzpv`, represented as a sparse matrix conforming to the pattern of the coarsest matrix, is projected to the next matrix on the first line of the for-loop by multiplying it with the transposes of the coarsening operators. At this point, the sparsity of `nnzpv` becomes a superset of the sparsity pattern of the matrix associated with that level in the matrix hierarchy. In order to get the accurate partition information, it is necessary to nullify the entries that fall outside the pattern of the matrix at the current level `Alevel`. After this adjustment, the partition on the current matrix is refined by invoking the function `Refine`. The function `Refine` is basically a wrapper to a mex function `PaToHRefineBsctn` which we have implemented to call any of the refinement functions of PaToH [12] on a fine-grain hypergraph model. We use the refinement algorithm `PATOH_REFALG_BFMKL` implemented in PaToH which performs one pass of boundary FM (BFM) followed by one pass of boundary KL (BKL). The function `PaToHRefineBsctn` has the signature

```
nnzpv = PaToHRefineBsctn(ia, ja, vw, pv, rnc, cnc, imbal);
```

where `ia(i)` and `ja(i)` define the nets of the vertex `i` which has a weight of `vw(i)`; the array `pv` of the same size as `ia` defines the current part of each vertex; the arrays `rnc` and `cnc` are the costs of the row nets and column nets computed in the function `PaToHMLUnCoarse`; finally `imbal` is the allowed imbalance ratio.

Algorithm 3 Multilevel refinement algorithm

```
function nnzpv = PaToHMLUnCoarse(A,PHier,QHier,nlevels,pv,imbal)
nnzpv = pv;
[m,n] = size(A);
for level = nlevels:-1:1,
    nnzpv = ((PHier{level})' * nnzpv) * (QHier{level})';
    [PMult, QMult]=PaToHMLGetCoarseOperator(PHier, QHier, level-1);
    Alevel = PMult * A * QMult;
    nnzpvlevel = nnzpv .* spones(Alevel);
    rnc = PMult * ones(size(PMult,2),1);
    cnc = QMult' * ones(size(QMult,1),1);
    nnzpv = Refine(nnzpvlevel, Alevel, rnc, cnc, imbal);
end
```

As noted before, the cutsize of a partition on a coarser matrix is an upper bound on the cutsize of the same partition on the original matrix. However, as the partition is projected towards the finer matrices, the cutsize of a partition will become tighter and tighter as an upper bound, and finally it will be exact on the original matrix. We also note that in order to reduce memory overhead, we do not store the coarsened matrices. This entails a large overhead during the uncoarsening phase, because the projection of the partition on a coarser level requires computing the pattern of the associated matrix (the statement `Alevel = PMult * A * QMult` in Algorithm 3). If we were to implement this routine as a standalone program, we would store the coarsened matrices for better performance.

Figure 4 shows the progress of the uncoarsening operation on the sample matrix shown in Fig. 3. At the top, we have the partition $\Pi^{(2)}$ on the coarsest matrix $A^{(2)}$. As noted before, this partition has a cutsize of 11. The partition $\Pi^{(2)}$ is then enlarged by multiplying it with the transposes of the coarsening operators of the same level, i.e., by $(P^{(2)})^T$ form the left and by $(Q^{(2)})^T$ from the right. The resulting partition $\Pi^{(1')}$ has a sparsity pattern which is a superset of the pattern of the associated matrix $A^{(1)}$. Filtering out the entries at positions that fall outside the sparsity pattern of $A^{(1)}$, i.e., by entry-wise multiplying it with `spones`(A), yields the partition $\Pi^{(1)}$ on $A^{(1)}$. The partition $\Pi^{(1)}$ has a cutsize of 7=2+2+2+1, since the rows 3, 4, 5, and 6 are split among parts 1 and 2. Note that the loads of the processors are 16 and 20. This partition is then refined (using the `Refine` subroutine) to yield an improved partition $\Pi^{(1)}$ with the same cutsize 7 and a better load distribution, 18 versus 18. The partition $\Pi^{(1)}$ is again enlarged through a multiplication operation using the transposes of the associated coarsening operators, $(P^{(1)})^T$ and $(Q^{(1)})^T$. Again, a partition $\Pi^{(0')}$ on a superset of the nonzeros of $A^{(0)} = A$ is found. This partition $\Pi^{(0')}$ is filtered using the pattern of A yielding $\Pi^{(0)}$ with a cutsize of 6 and a load distribution 18 versus 18. Later $\Pi^{(0)}$ is refined to yield the final partition on A given in Fig. 5. This partition has a total

$$\Pi^{(2)} = \begin{array}{c} \\ [4]\,1 \\ [4]\,2 \\ [3]\,3 \end{array} \begin{array}{c} \begin{array}{cccc} {\scriptstyle [4]} & {\scriptstyle [3]} & {\scriptstyle [3]} & {\scriptstyle [1]} \\ 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 2 & 2 & \\ 1 & 2 & 2 & \\ 1 & 2 & 2 & 1 \end{pmatrix} \end{array}$$

$$\Pi^{(1')} = (P^{(2)})^T \times \Pi^{(2)} \times (Q^{(2)})^T = \begin{array}{c} \\ [2]\,1 \\ [2]\,2 \\ [2]\,3 \\ [2]\,4 \\ [2]\,5 \\ [1]\,6 \end{array} \begin{array}{c} \begin{array}{ccccccc} {\scriptstyle [2]} & {\scriptstyle [2]} & {\scriptstyle [2]} & {\scriptstyle [2]} & {\scriptstyle [1]} & {\scriptstyle [1]} & {\scriptstyle [1]} \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} 1 & 1 & 2 & 2 & 2 & 2 & \\ 1 & 1 & 2 & 2 & 2 & 2 & \\ 1 & 1 & 2 & 2 & 2 & 2 & \\ 1 & 1 & 2 & 2 & 2 & 2 & 1 \\ 1 & 1 & 2 & 2 & 2 & 2 & \\ 1 & 1 & 2 & 2 & 2 & 2 & 1 \end{pmatrix} \end{array}$$

$$\Pi^{(1)} = \Pi^{(1')} \cdot \times \mathrm{spones}(A^{(1)})$$

$$\Downarrow$$

$$\begin{array}{c} \\ [2]\,1 \\ [2]\,2 \\ [2]\,3 \\ [2]\,4 \\ [2]\,5 \\ [1]\,6 \end{array} \begin{array}{c} \begin{array}{ccccccc} {\scriptstyle [2]} & {\scriptstyle [2]} & {\scriptstyle [2]} & {\scriptstyle [2]} & {\scriptstyle [1]} & {\scriptstyle [1]} & {\scriptstyle [1]} \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} 1 & 1 & & & & & \\ & & 2 & & & 2 & \\ 1 & 1 & 2 & 2 & & & \\ & 1 & 2 & & 2 & & \\ 1 & 1 & 2 & 2 & 2 & & \\ & 1 & 2 & & & & 1 \end{pmatrix} \end{array} \overset{\text{refine}}{\Longrightarrow} \Pi^{(1)} = \begin{array}{c} \\ [2]\,1 \\ [2]\,2 \\ [2]\,3 \\ [2]\,4 \\ [2]\,5 \\ [1]\,6 \end{array} \begin{array}{c} \begin{array}{ccccccc} {\scriptstyle [2]} & {\scriptstyle [2]} & {\scriptstyle [2]} & {\scriptstyle [2]} & {\scriptstyle [1]} & {\scriptstyle [1]} & {\scriptstyle [1]} \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} 1 & 1 & & & & & \\ & & 2 & & & 2 & \\ 1 & 1 & 2 & 2 & & & \\ & 1 & 2 & & 1 & & \\ 1 & 1 & 2 & 2 & 1 & & \\ & 1 & 2 & & & & 2 \end{pmatrix} \end{array}$$

$$\Pi^{(0')} = (P^{(1)})^T \times \Pi^{(1)} \times (Q^{(1)})^T = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & & 1 & & & & & & \\ & & & 2 & & & & 2 & 2 & & \\ 1 & 1 & 1 & 2 & 1 & 2 & & 2 & & 2 & \\ & 1 & 1 & 2 & & & 1 & 2 & & & \\ 1 & 1 & 1 & 2 & 1 & 2 & & 2 & & 2 & \\ 1 & 1 & 1 & 2 & 1 & 2 & 1 & 2 & & 2 & \\ & & & 2 & & & & 2 & 2 & & \\ & 1 & 1 & 2 & & & & 2 & & & 2 \\ & 1 & 1 & 2 & & & 1 & 2 & & & \\ 1 & 1 & 1 & & 1 & & & & & & \\ 1 & 1 & 1 & 2 & 1 & 2 & 1 & 2 & & 2 & \end{pmatrix} \end{array}$$

$$\Pi^{(0)} = \Pi^{(0')} \cdot \times \mathrm{spones}(A^{(0)}) = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{array} \\ \begin{pmatrix} & 1 & 1 & & & & & & & & \\ & & & 2 & & & & 2 & & & \\ 1 & 1 & & & & & & & & 2 & \\ & & & & & & 1 & 2 & & & \\ & 1 & & & & 2 & & 2 & & 2 & \\ 1 & 1 & 1 & 2 & 1 & & 1 & 2 & & & \\ & & & 2 & & & & 2 & 2 & & \\ & 1 & & 2 & & & & & & & 2 \\ & 1 & & & & & 1 & 2 & & & \\ 1 & 1 & 1 & & 1 & & & & & & \\ & & & 2 & & 2 & & 2 & & & \end{pmatrix} \end{array}$$

Fig. 4. Uncoarsening operation succeeding the result of the coarsening operation shown in the previous figure.

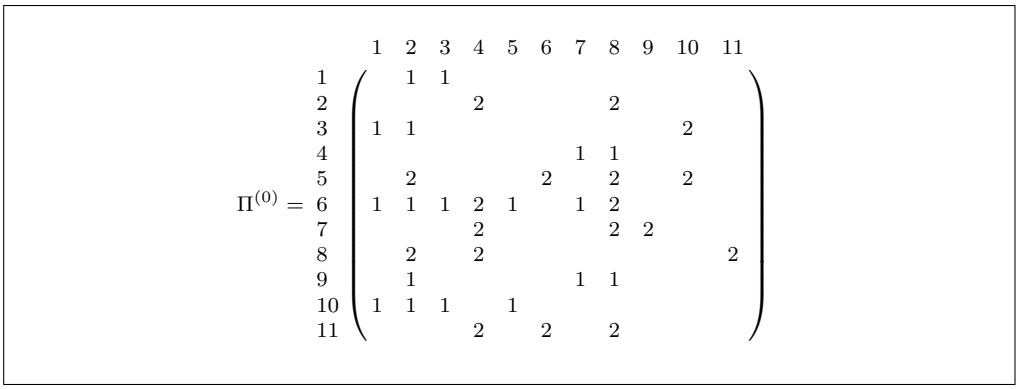


Fig. 5. Continuing from the previous figure, $\Pi^{(0)}$ is refined on the matrix $A^{(0)}$ to yield a total communication volume of 4 (rows 3 and 6 and columns 2 and 8 are split among the two processors).

cutsize of 4 due to the splits in rows 3 and 6 and in columns 2 and 8. The load distribution is again 18 versus 18.

5 Experiments

We have performed an extensive experimental evaluation of the proposed novel 2D partitioning method using the PaToH MATLAB Matrix-Partitioning Interface, on a large set of matrices from the University of Florida (UFL) sparse matrix collection [27]. The experiments were conducted on computers 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.

We have performed two different sets of experiments. In the first set of experiments, we aimed to investigate the merits of the cosine similarity metric used in the proposed multilevel 2D coarsening-based partitioning method (ML2D). In the second set, we investigated the performance of the proposed ML2D method. In the following, we use the term "partitioning instance" to refer to the partitioning of a matrix into a given number of parts. That is, for a given K, a K-way partitioning of a matrix constitutes a partitioning instance.

5.1 Results

In the first set of experiments, we have compared the total communication volume found by ML2D with the cosine similarity to that found by ML2D with the inner product metric. For this comparison, we did not apply the threshold test on the similarity of two rows or columns under the cosine similarity metric.

The inner product metric, otherwise known as the heavy connectivity metric, has been used in the coarsening phase of the hypergraph partitioning tools Mondriaan [6] and PaToH [12]. In these tools, the items whose inner products are computed are {0,1}-vectors. In our case, we compute the similarity of the rows or columns of a non-{0,1} matrix. Therefore, a sort of length-scaling is likely to be required, justifying the use of the cosine similarity instead of the inner product as the matching metric. We performed the first set of experiments in order to experimentally support the observation above. These experiments are performed using $K \in \{8, 16, 64\}$, and on real, square matrices whose number of nonzeros are between 1,000 and 5,000,000 from the UFL collection, skipping the partitioning instances in which $\min\{M, N\} < 50 \times K$. Out of 1,576 partitioning instances (at the time of writing), in only 272 of them the inner product metric performed better than the cosine similarity metric; in one instance they produced the same result. In the remaining 1,303 instances, the cosine similarity metric produced better results than the inner product metric; about 10% better on the overall average. Hence, we made the cosine similarity as the default matching criterion for the proposed ML2D method.

In the second set of experiments, we compared the performance of the proposed ML2D with the orthogonal recursive bisection, ORB (see Section 3.2). We have implemented both of them using the described matrix partitioning interface. We also wanted to compare the performance of these two methods against the existing methods presented in Section 2.3. Since our primary interest was the total communication volume, we decided to compare the two methods against the fine-grain (FG) method—in a recent work of ours [5], FG was found, by a thorough experimental evaluation, to be the best method for minimizing the total communication volume. In that work, we presented a partitioning recipe which suggests a partitioning method among those methods summarized in Section 2.3. The recipe was experimentally shown to be close to FG in the total volume of communication metric while being much faster. We did not incorporate the ML2D or ORB methods yet into the recipe, therefore we compare them against the FG method. This also makes sense, as the proposed ML2D method uses the fine-grain model in the refinement phase and produces an arbitrary 2D partitioning as the FG method does.

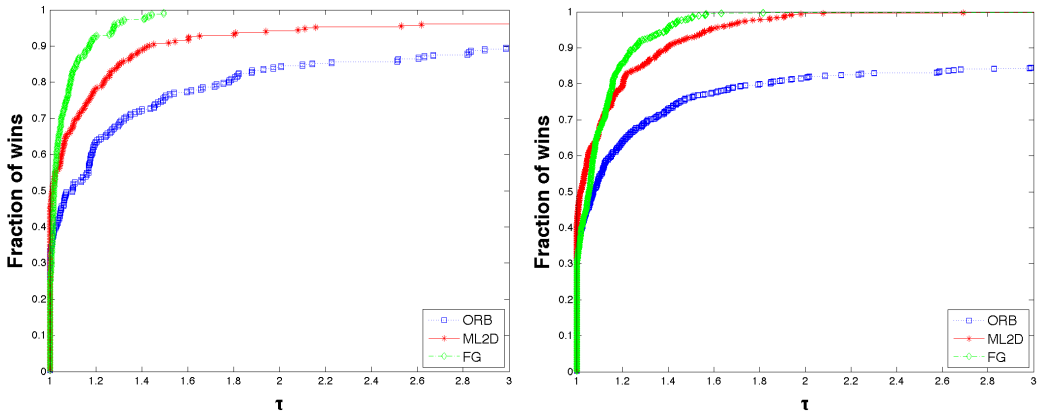
During the second set of experiments, in order to avoid possible skew or bias due to an overpopulated matrix group (there were 149 of them at the UFL collection at the time of writing), we have selected up to 5 largest matrices from each group. In this selection, we excluded matrices with number of nonzeros larger than 10,000,000 in order to complete the experiments in a reasonable time frame. We also excluded small matrices (i.e., matrices that have less than 1,000 nonzeros or columns/rows), as the parts would become too small to be meaningful. In total, we ran our experiments on 412 matrices with $K \in \{4, 16, 64, 256, 512\}$. We further discarded partitioning instances in which

$\min\{M, N\} < 100 \times K$. This resulted in 1,504 partitioning instances where 670 of them were with a symmetric matrix, 576 of them were with a non-symmetric square matrix, and 258 of them were with a rectangular matrix. For each partitioning instance, the presented results are the averages of 15 runs. We used a threshold test for the cosine similarity of two rows or columns. Two rows (or columns) are deemed similar if $\cos\theta \geq 0.70$ which corresponds roughly to 45°. Again the set of candidate mates for a row (or a column) are visited, the cosine similarities are computed, and the larger one is declared as a mate if it passes the threshold test; if not, the starting row (or the starting column) left as a singleton.

For the second set of experiments, we use performance profiles [32] to present our results. Performance profiles are a generic tool for comparing a set of methods over a large set of test cases 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. In our experiments, partitioning instances constitute the test cases. We use the ratio of a performance indicator to the best among all partitioning methods as our performance metric and call it τ. As performance indicators, we use the total communication volume, the maximum volume of messages sent by a single processor, the total number of messages, and the partitioning time.

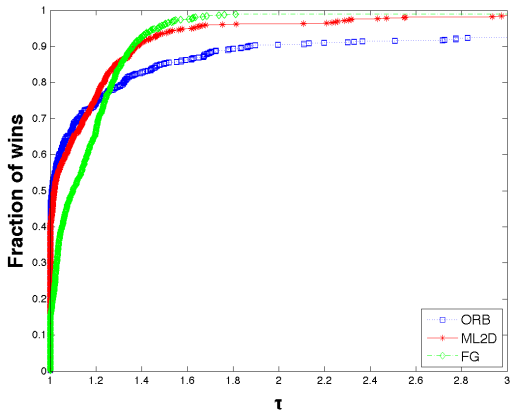
The first set of performance profile charts displayed in Fig. 6 uses the total communication volume as the performance indicator. As seen in the charts, FG and ML2D, in general, are better than ORB, though ORB obtains results within 2 times of the best result in about 85% of the cases. In the rectangular test cases, there is a gap between FG and ML2D, otherwise, they behave similarly. In the symmetric test cases, ORB and ML2D perform similarly and outperform FG until about $\tau = 1.2$. After this point, FG's performance improves and passes that of ORB and matches to that of ML2D. After $\tau = 1.4$, the performances of the three methods stabilize where the methods rank as FG, ML2D, and ORB. In other cases, the same ranking is obtained after a smaller τ.

Table 2 displays some extreme cases where one of the methods performs significantly better than the others in terms of the total communication volume (we restricted this analysis to the instances for which at least one of the methods obtained a total communication volume of at least 3,000). As seen in the table, although there are cases that ORB performs significantly better than ML2D, ORB's performance on those cases are almost the same as those of FG (the extreme cases for which FG outperformed ML2D are the same as the extreme cases for which ORB outperformed ML2D). There are some cases where ORB performs nearly two times better than FG, but in general FG performs significantly better, and in some cases FG produces total commu-

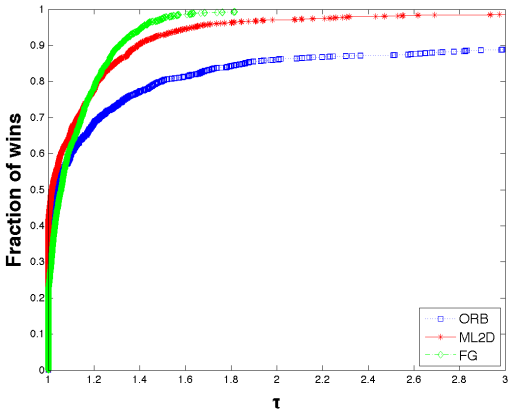


(a) rectangular instances

(b) non-symmetric instances



(c) symmetric instances



(d) all instances

Fig. 6. Performance profile plots comparing the three partitioning methods using the total communication volume as the performance indicator. In all sub-figures, at $\tau = 1.8$ the plots are FG, ML2D, and ORB from top to bottom.

nication volume that are two orders of magnitude better than ORB. In the instances where ML2D performs better than ORB, FG is usually even better than ML2D. There are instances where ML2D performs better than FG, in a significant portion of those cases, interestingly, ORB is also better than FG.

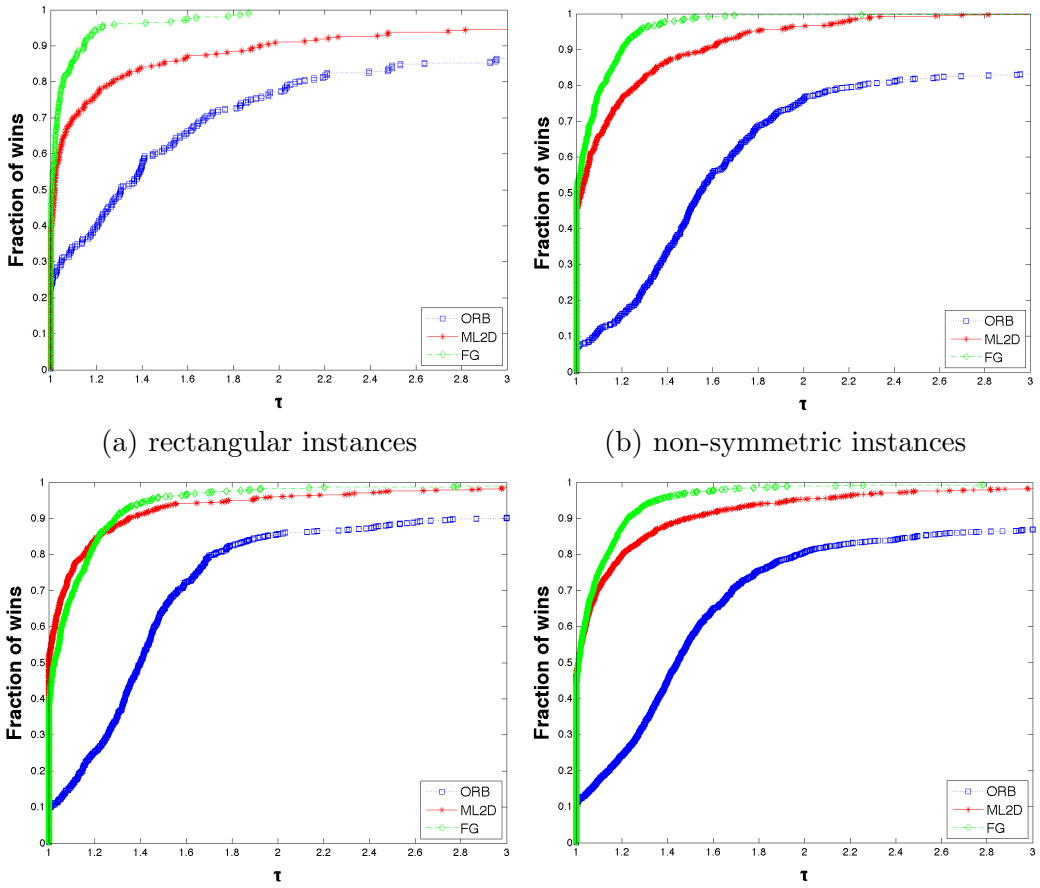
Figure 7 displays the comparison of the three partitioning methods using the maximum communication volume per processor as the performance indicator. Apart from the symmetric partitioning instances, FG clearly outperforms the other two. In about a little less than 90% of the cases, its results are within 1.2 times of the best result. The proposed ML2D is almost always in between the other two methods, while being more closer to FG than to ORB. Again ML2D's performance is visibly worse than that of FG in the rectangular cases. We had first experimented with the ML2D method in such a way that the two-dimensional coarsening was always enforced. In that case, there were again an observable difference, more amplified than the shown results, between the performances of ML2D and FG. Later on, we have updated the code such that on rectangular cases if the number of rows is less than the two thirds of the

Table 2
Comparison of total communication volumes found by ORB, ML2D and FG on some extreme cases.

Group/Name	**Prop**	K	**ORB**	**ML2D**	**FG**
ORB and FG perform better than ML2D					
LPnetlib/lp_osa_60	RECT	4	239.0	6516.6	239.0
LPnetlib/lp_osa_60	RECT	16	1085.3	15440.5	1088.7
LPnetlib/lp_osa_30	RECT	4	236.5	3321.4	235.9
ORB performs better than FG					
Li/pli	SYM	4	6066.2	9878.8	10695.2
Norris/torso1	NON-SYM	4	5510.7	6643.9	10009.8
ND/nd6k	SYM	4	13474.5	30436.9	24422.0
ML2D performs better than ORB					
IBM_EDA/trans4	NON-SYM	4	109211.7	649.1	539.5
IBM_EDA/trans5	NON-SYM	4	107876.9	659.5	544.7
Muite/Chebyshev3	NON-SYM	4	3592.9	20.0	20.0
ML2D performs better than FG					
Kamvar/Stanford_Berkeley	NON-SYM	4	5481.3	891.4	1395.2
Norris/torso1	NON-SYM	4	5510.7	6643.9	10009.8
Mancktelow/viscorocks	NON-SYM	16	4678.1	3907.3	5768.1
FG performs better than ORB					
IBM_EDA/dc2	NON-SYM	4	109775.5	1049.1	550.1
IBM_EDA/trans4	NON-SYM	4	109211.7	649.1	539.5
IBM_EDA/dc3	NON-SYM	4	109321.9	758.1	542.4

number of columns, only columnwise coarsening is performed (similar policy is applied for the other way around). Combined with the effect of the similarity threshold value, this had lifted the performance of the ML2D method towards that of the FG method, but there seems to be room for improvement.

Figure 8 presents the comparison of the three partitioning methods using the total number of messages as the performance indicator. As one might expect, FG and ML2D show a very similar trend in general. ORB manages to produce the best result in terms of the total number of messages in about 95% of the cases, at which point the results of FG and ML2D are within 2.2 times of the best.



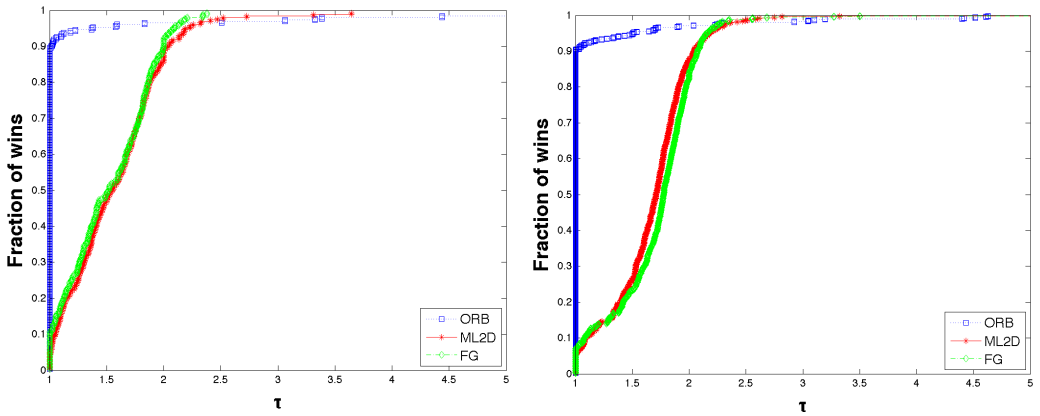
(a) rectangular instances (b) non-symmetric instances

(c) symmetric instances (d) all instances

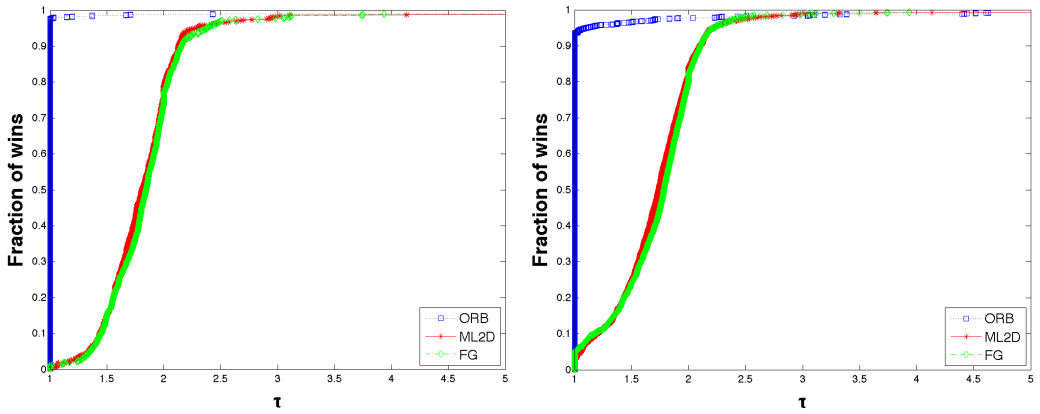
Fig. 7. Performance profile plots comparing the three partitioning methods using the maximum communication volume per processor as the performance indicator. In all sub-figures, at $\tau = 1.8$ the plots are FG, ML2D, and ORB from top to bottom.

In all of our experiments, we had used computational imbalance ratio $\varepsilon = 3\%$. The three methods almost always produce well-balanced partitions. The FG and ML2D methods, being able to improve partitions on a nonzero basis, always obtained results within the allowable imbalance ratio. The ORB method failed to satisfy the required balance only in 2 instances where the obtained balance were 3.1 for 512-way partitioning of `BM_EDA/trans5` and 3.2 for 512-way partitioning of `IBM_EDA/dc3`.

Finally, even though our implementations of ORB and ML2D are not optimized in terms of execution time, we present the execution time comparison of the three methods just to give a rough idea about their characteristics. Figure 9 presents this comparison. In terms of execution time, we measured the time spent during actual hypergraph partitioning. For ML2D, we also included matching time and refinement time during coarsening and uncoarsening phases. This gives an advantage to ORB as the recursion overhead, including the construction of the submatrices, is not included in the timings, and to ML2D as the construction of coarsened matrix/hypergraph and projection



(a) rectangular instances (b) non-symmetric instances



(c) symmetric instances (d) all

Fig. 8. Performance profile plots comparing the three partitioning methods using the total number of messages as the performance indicator. In all sub-figures, the plots of FG and ML2D are very close to each other, and the plot of ORB is above those two.

times are not included in the timings. For a more fair comparison one should implement all methods completely in the same programming environment and in the same programming language. With these added advantages ORB is the fastest of the three methods in about 85% of the test cases. Second fastest is ML2D and FG is the slowest. With the current testing environment, the average run time of the ORB, ML2D, and the FG methods are, respectively, 15, 50, and 81 seconds. With a more fair comparison, we expect the differences among the average run time of the methods to decrease.

5.2 Further results and discussion

We report some more observations on the ML2D method. First, we note that we did not test the effect of the contraction parameter (7/8 in Algorithm 2); rather we used a value close to a related parameter in PaToH. The reason to

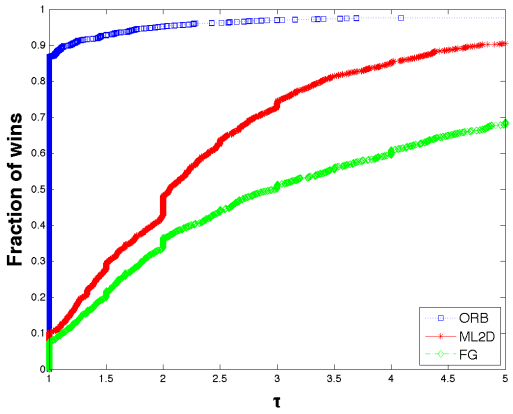


Fig. 9. Performance profile plots comparing the three partitioning methods using the partitioning time as the performance indicator. At any τ, the plots correspond to ORB, ML2D, and FG from top to bottom.

use this parameter is to avoid unnecessary calculations; if at a step the contraction is low, it is likely that in the next step the same will hold. Therefore, this parameter would not have a significant effect on the partitioning quality. On the other hand, the value of the threshold parameter ϑ has a profound impact. First, it affects the quality of the coarsening—most often a high quality coarsening leads to high quality partitions. We have observed 20% difference between using $\vartheta = 0.70$ and not using a threshold at all. We have also tested $\vartheta = 0.75$ and $\vartheta = 0.60$ which correspond, respectively to, about 41° and 53°. With the higher one, there were too little contraction, with the smaller one, there were too much contraction. Second, the threshold also affects the run time (in combination with the test on the contraction amount). If a loose one is used, then a higher number of coarsening operations are performed, resulting in a faster execution. With no threshold test, the average run time were 27 seconds, whereas with $\vartheta = 0.70$ it is 50 seconds. With the threshold test, the average number of coarsening levels were $3.97, 4.27, 4.63, 5.17$ and 5.30 for $K = 4, 16, 64, 256$, and 512, respectively. These numbers show that 2D coarsening with threshold test continues for a nontrivial number of levels. Although we are content with the choice of $\vartheta = 0.70$, more investigation can be undertaken, for perhaps adapting the threshold test in the lower levels of coarsening.

As we had already stated, ML2D performs only one side coarsening if the number of rows or the number of columns are less than the two thirds of the other dimension. We made this choice according to our previous experience in developing a recipe for the matrix partitioning problem [5]. However, as seen in the performance profile figures, ML2D lacks behind FG for the rectangular matrices, both without (not reported in the paper) and with the threshold parameter. We think that developing successful coarsening policies in combination with the threshold parameter rests as a future work.

6 Conclusion

We presented the PaToH MATLAB Matrix Partitioning Interface that provides support for various hypergraph-based 1D and 2D matrix partitioning methods including rowwise, columnwise, jagged-like, checkerboard and fine-grain partitionings [2–5]. In addition to providing an easy access to existing partitioning methods, this interface enables fast prototyping of new matrix partitioning methods.

We also proposed a novel multilevel 2D coarsening-based, 2D partitioning method, ML2D, based on the cosine similarity of the rows and columns of a suitably defined matrix. We implemented the proposed method as well as an orthogonal recursive bisection method in MATLAB using the newly developed interface. We carried out an extensive evaluation of these implementations using a large set of test matrices (more than 400) from University of Florida Sparse Matrix collection. We concluded that ML2D can compete with the fine-grain method in reducing the total communication volume while being faster on average. We also suggested further research directions for improving the ML2D method's performance. The relative performance of ML2D with respect to the fine-grain method drops for rectangular matrices which calls for the evaluation of the coarsening policies we used in this paper. We use a threshold test for the similarity test, which seems to have a profound effect on the performance of the ML2D method. We believe that improvements on the threshold test would most probably lead to improved performance, both in terms of the partition quality and the partitioning time. Apart from these most immediate improvements, one can try to develop a matching variant which incorporates the sizes of the nets into the similarity metric. Furthermore, one can also try to use agglomerative clustering schemes in which coarser vertices represent a set of finer vertices instead of just two.

References

[1] Ü. V. Çatalyürek, C. Aykanat, Decomposing irregularly sparse matrices for parallel matrix-vector multiplications, Lecture Notes in Computer Science 1117 (1996) 75–86.

[2] Ü. V. Çatalyürek, C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, IEEE Transactions Parallel and Distributed Systems 10 (7) (1999) 673–693.

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

[4] Ü. V. Çatalyürek, C. Aykanat, A hypergraph-partitioning approach for coarse-grain decomposition, in: ACM/IEEE SC2001, Denver, CO, 2001.

[5] Ü. V. Çatalyürek, C. Aykanat, B. Uçar, On two-dimensional sparse matrix partitioning: Models, methods, and a recipe, Tech. Rep. TR_2008_n04, The Ohio State University, Department of Biomedical Informatics, accepted for publication in SIAM J. Sci. Comput. (2009).

[6] B. Vastenhouw, R. H. Bisseling, A two-dimensional data distribution method for parallel sparse matrix-vector multiplication, SIAM Review 47 (1) (2005) 67–95.

[7] B. Uçar, C. Aykanat, Minimizing communication cost in fine-grain partitioning of sparse matrices, Lecture Notes in Computer Science 2869 (2003) 926–933.

[8] B. Uçar, C. Aykanat, Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies, SIAM Journal on Scientific Computing 25 (6) (2004) 1827–1859.

[9] B. Uçar, C. Aykanat, Revisiting hypergraph models for sparse matrix partitioning, SIAM Review 49 (4) (2007) 595–603.

[10] B. Uçar, C. Aykanat, Partitioning sparse matrices for parallel preconditioned iterative methods, SIAM Journal on Scientific Computing 29 (4) (2007) 1683–1709.

[11] J. R. Gilbert, G. L. Miller, S.-H. Teng, Geometric mesh partitioning: Implementation and experiments, SIAM J. Sci. Comput. 19 (6) (1998) 2091–2110.

[12] Ü. V. Çatalyürek, C. Aykanat, PaToH: A multilevel hypergraph partitioning tool, version 3.0, Tech. Rep. BU-CE-9915, Computer Engineering Department, Bilkent University (1999).

[13] R. H. Bisseling, W. Meesen, Communication balancing in parallel sparse matrix-vector multiplication, Electronic Transactions on Numerical Analysis 21 (2005) 47–65.

[14] B. Hendrickson, T. G. Kolda, Graph partitioning models for parallel computing, Parallel Computing 26 (2000) 1519–1534.

[15] B. Uçar, C. Aykanat, A library for parallel sparse matrix-vector multiplies, Tech. Rep. BU-CE-0506, Department of Computer Engineering, Bilkent University, Ankara, Turkey (2005).

[16] T. Lengauer, Combinatorial Algorithms for Integrated Circuit Layout, Wiley–Teubner, Chichester, U.K., 1990.

[17] C. Aykanat, B. B. Cambazoglu, B. Uçar, Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices, Journal of Parallel and Distributed Computing 68 (5) (2008) 609–625.

[18] Ü. V. Çatalyürek, Hypergraph models for sparse matrix partitioning and reordering, Ph.D. thesis, Bilkent University, Computer Engineering and Information Science (Nov 1999).

[19] G. Karypis, V. Kumar, R. Aggarwal, S. Shekhar, hMeTiS: A Hypergraph Partitioning Package Version 1.0.1, University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis (1998).

[20] K. Schloegel, G. Karypis, V. Kumar, Parallel multilevel algorithms for multi-constraint graph partitioning, in: Euro-Par, 2000, pp. 296–310.

[21] T. N. Bui, C. Jones, A heuristic for reducing fill in sparse matrix factorization, in: Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing, 1993, pp. 445–452.

[22] G. Karypis, V. Kumar, Multilevel k-way hypergraph partitioning, VLSI Design 11 (3) (2000) 285–300.

[23] C. M. Fiduccia, R. M. Mattheyses, A linear-time heuristic for improving network partitions, in: Proceedings of the 19th ACM/IEEE Design Automation Conference, 1982, pp. 175–181.

[24] B. W. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, The Bell System Technical Journal 49 (2) (1970) 291–307.

[25] L. F. Romero, E. L. Zapata, Data distributions for sparse matrix vector multiplication, Parallel Computing 21 (4) (1995) 583–605.

[26] B. Uçar, Ü. V. Çatalyürek, C. Aykanat, PaToH MATLAB interface, `http://bmi.osu.edu/~umit/software.html` (July 2009).

[27] T. Davis, The University of Florida sparse matrix collection, Tech. Rep. REP-2007-298, CISE Department, University of Florida, Gainesville, FL, USA (2007).

[28] M. J. Berger, S. H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, IEEE Trans. Computers C-36 (5) (1987) 570–580.

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

[30] R. H. Bisseling, T. van Leeuwen, Ü. V. Çatalyürek, A hybrid 2D method for sparse matrix partitioning, Presentation at SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, Feb. 22–24 (2006).

[31] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1) (1998) 359–392.

[32] E. D. Dolan, J. J. Moré, Benchmarking optimization software with performance profiles, Math. Prog. 91 (2) (2002) 201–213.