Exact k-way sparse matrix partitioning

Engelina L. Jenneskens
Mathematical Institute
Utrecht University
Utrecht, The Netherlands
e.l.jenneskens@gmail.com

Rob H. Bisseling
Mathematical Institute
Utrecht University
Utrecht, The Netherlands
r.h.bisseling@uu.nl

***Abstract*—To minimize the communication in parallel sparse matrix-vector multiplication while maintaining load balance, we need to partition the sparse matrix optimally into k disjoint parts, which is an NP-complete problem. We present an exact algorithm based on the branch and bound (BB) method which partitions a matrix for any k, and we explore exact sparse matrix partitioning beyond bipartitioning. The algorithm has been implemented in a software package General Matrix Partitioner (GMP). We also present an integer linear programming (ILP) model for the same problem, based on a hypergraph formulation. We used both methods to determine optimal 2,3,4-way partitionings for a subset of small matrices from the SuiteSparse Matrix Collection. For k=2, BB outperforms ILP, whereas for larger k, ILP is superior. We used the results found by these exact methods for k=4 to analyse the performance of recursive bipartitioning (RB) with exact bipartitioning. For 46 matrices of the 89 matrices in our test set of matrices with less than 250 nonzeros, the communication volume determined by RB was optimal. For the other matrices, RB is able to find 4-way partitionings with communication volume close to the optimal volume.**

***Index Terms*—branch-and-bound, integer linear programming, exact algorithm, sparse matrix-vector multiplication, hypergraph, parallel computing**

I. Introduction

Sparse matrix–vector multiplication (SpMV) is the core operation at the heart of many computations in scientific computing and in data analytics, such as iterative linear system solutions [1] and graph computations formulated in terms of sparse matrix operations [2]. To solve larger problems, SpMVs need to be carried out in parallel, which requires a suitable data partitioning into multiple disjoint parts.

Assume that we have an $m \times n$ matrix A, with entries a_{ij}, $0 \leq i < m$ and $0 \leq j < n$, of which $nz(A)$ are nonzero. The parallel multiplication of A with a dense vector $\mathbf{v}$ of length n yields a vector

$$\mathbf{u} = A\mathbf{v} \tag{1}$$

of length m. A parallel algorithm for the computation of the values

$$u_i = \sum_{j=0}^{n-1} a_{ij} v_j, \qquad 0 \leq i < m, \tag{2}$$

for an arbitrary distribution of the matrix and the vectors across the processors, consists of four phases:

1) **Fan-out:** Communication of the values v_j to the processors that need them.

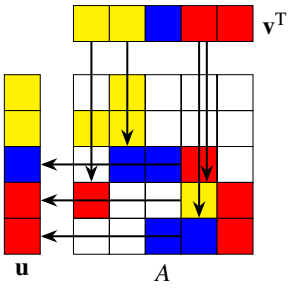

Fig. 1. Parallel sparse matrix-vector multiplication, with three processors 0 (red), 1 (blue), and 2 (yellow). The arrows represent the communication between the processors, with vertical arrows for the fan-out and horizontal arrows for the fan-in. The owners of the nonzeros and the vector components are indicated by their color. The zeros of the matrix are shown in white. The communication volume is equal to the number of arrows, $CV = 7$.

2) **Local SpMV:** Local multiplications $a_{ij} \cdot v_j$ and additions into a local partial sum.
3) **Fan-in:** Communication of partial sums to the owners of the values u_i.
4) **Summation:** Summation of the received partial sums.

Fig. 1 illustrates the parallel SpMV algorithm. In this article, we will only be concerned with the distribution of the matrix: we assume that the vector distribution can be chosen freely based on the matrix distribution. Therefore, the owner of v_j can be taken as one of the processors represented in column j of the matrix, and similarly for u_i and row i of the matrix. Thus, the vector distribution will not cause any additional communication. Without loss of generality, we assume that no matrix row or column is empty. Such rows and columns can easily be removed from the matrix without affecting the partitioning problem.

The speed of a parallel SpMV depends heavily on an equal distribution of the workload and on the number of data words that need to be communicated in the fan-out and fan-in phases. This gives rise to the problem of partitioning a sparse matrix A into k disjoint parts,

$$A = \bigcup_{i=0}^{k-1} A_i, \tag{3}$$

while minimizing the communication volume and maintaining

978-1-6654-9747-3/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPSW55747.2022.00129

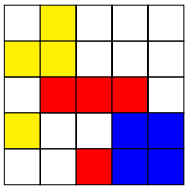

Fig. 2. An optimal 3-way partitioning of the matrix from Fig. 1, with communication volume $CV = 4$. The load imbalance parameter was set to $\epsilon = 0.03$.

the load balance, expressed by the constraint

$$nz(A_i) \leq (1+\epsilon) \left\lceil \frac{nz(A)}{k} \right\rceil, \quad 0 \leq i < k. \tag{4}$$

Here, $\epsilon \geq 0$ is the *load imbalance parameter*. It is set beforehand, often to a value $\epsilon = 0.03$. The ceiling function is used to make sure that a valid partition is possible even when $\epsilon = 0$. This problem is called the *sparse matrix partitioning problem*. If we define λ_j^c to be the number of different processors that own an element in column j of A, then the communication volume of this matrix column is $\lambda_j^c - 1$, since during the fan-out one of these processors sends v_j to the others. Similarly, the communication volume of a row i during the fan-in is equal to $\lambda_i^r - 1$, with λ_i^r the number of processors owning a nonzero in row i. The total communication volume of a partitioned $m \times n$ matrix A in the parallel SpMV then equals

$$CV(A) = \sum_{i=0}^{m-1} (\lambda_i^r - 1) + \sum_{j=0}^{n-1} (\lambda_j^c - 1). \tag{5}$$

Different partitionings of the same matrix can have a large difference in communication volume. In Fig. 2, an optimal 3-way partitioning of the matrix from Fig. 1 is shown, which has $CV = 4$, while the partitioning in Fig. 1 has $CV = 7$.

The sparse matrix partitioning problem has been shown to be NP-complete for fixed ϵ, already for $k = 2$ [3]. Therefore, only relatively small problems can be solved to optimality, while for larger problems a heuristic algorithm is necessary. Solving small problems may not seem very useful, but the solutions to these small problems can be used to analyse the performance of heuristic methods. For instance, it was shown in [3] for a set of 839 test matrices from the SuiteSparse collection [4] that combining the medium-grain (heuristic) method from the Mondriaan package [5], [6] with the (heuristic) hypergraph partitioner from the PaToH package [7] yields on average results within 10% of optimality; this justifies the practical use of such heuristic solvers for bipartitioning, presumably also for larger problems.

The communication requirements of sparse matrix partitioners can be modeled exactly by formulating the partitioning problem in terms of hypergraphs. The problem can then be solved by using one of the currenly available sequential hypergraph partitioners hMetis [8], PaToH [7], Mondriaan [5], KaHyPar [9], [10], and the parallel partitioner Zoltan [11].

A recent exact solver for the sparse matrix partitioning problem is MondriaanOpt [12], a branch-and-bound (BB) bipartitioner that branches on the possible choices for a row or column, which is either completely assigned to processor 0, or completely assigned to processor 1, or *cut*, meaning that some nonzeros are assigned to processor 0 and others to processor 1, where the choice of nonzeros need not be specified. MondriaanOpt prunes the solution tree by lower bounds on the communication volume of a partial solution, based on cuts that are either explicitly or implicitly present. It increases these bounds by exploiting inevitable conflicts between rows and columns that share a nonzero, or by inevitable cuts of rows or columns with many nonzeros to prevent violation of the load balance constraint (4).

MondriaanOpt has been extended by Mumcuyan, Usta, Kaya, and Yenigün [13] with machine learning techniques to obtain a good branching ordering of the rows and columns for a given sparse matrix, and it has been accelerated by shared-memory parallelization. Usta also developed an exact BB-based k-way hypergraph partitioner PHaraoh [14], which can be used to find an optimal sparse matrix partitioning by applying it to the corresponding fine-grain hypergraph [15].

An improved exact solver is MatrixPartitioner (MP) [3], which exploits conflicts not only by directly intersecting rows and columns, but also along a path of nonzeros in the matrix with conflicting end points. Furthermore, MP also generalizes the use of the load balance constraint by considering whole neighbourhoods of rows and columns instead of single rows or columns. This led to a much larger set of matrices that could be solved, extending the database with results for 356 matrices from the SuiteSparse collection [4] to 839 matrices.

The aims of the present work are: (i) to develop a BB-based exact sparse matrix partitioner for the general case $k \geq 2$, which we call General Matrix Partitioner (GMP); (ii) to formulate the problem as an integer linear programming (ILP) problem that can be fed into an ILP solver; (iii) to compare the speed and the quality of BB and ILP partitioners for a set of (inevitably small) test problems and to provide their solutions in a database; (iv) study the quality of recursive bipartitioning (RB) by using exact bipartitioning within the overall partitioning framework.

II. Partitioning by the Branch-and-Bound Method

The BB method explores the set $\mathcal{S}$ of feasible solutions of an optimization problem and uses bounds on the optimal solution to prevent searching the whole feasible region. In our case, $\mathcal{S}$ is the set of partitionings of the matrix that meet the load balance constraint (4). We start with the complete set $\mathcal{S}$ and we split it into subsets $\mathcal{S}_1, \ldots, \mathcal{S}_r$, based on a well-chosen property of the solutions. This procedure is repeated by splitting the subsets each into smaller subsets based on another property of the solutions and we repeat this process until we have sets that represent a single solution. This splitting of the set of feasible solutions is the "branch" part of the BB algorithm. The whole BB process can be captured by a rooted tree with the root representing the complete set of feasible solutions, each node

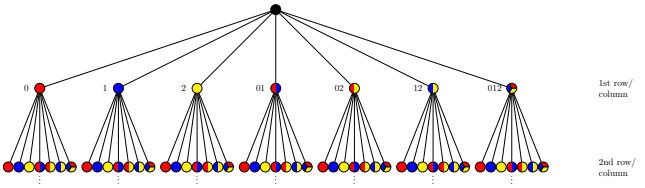

Fig. 3. The first two levels of the BB tree for $k = 3$, with three processors 0 (red), 1 (blue), and 2 (yellow). A node with multiple colors represents that the row/column is assigned to multiple processors.

representing a subset of the feasible solutions, and each leaf representing a single solution. Essentially, the BB method is just an enumeration of all possible feasible solutions.

Fig. 3 shows a BB tree for our partitioning problem that branches on the first two rows or columns, based on the property of the colors present in the row or column. For $k = 3$, this means that a row or column can be assigned either to processor 0, 1, or 2, or to a pair 01, 02, 12, or to all processors, denoted by 012.

The BB tree of the partitioning problem grows exponentially: every node has $2^k - 1$ children, so that the number of leaves is $(2^k - 1)^{m+n}$. The size of the BB tree can be reduced by at most a factor of $k!$ if we exploit symmetry: when we introduce a new processor in the partitioning, we can take the lowest numbered one not used so far. In Fig. 3, at the first level only the nodes 0, 01, and 012 need to be kept; the others can be discarded because their solutions are equivalent to one of these three. For node 0, its children to be kept are 0, 1, 01, 12, 012; the nodes 2 and 02 can be discarded. For the other nodes, this is done in a similar fashion.

We can use bounds on the communication volume to avoid searching the whole feasible set $\mathcal{S}$. Assume that we have an upper bound UB on the optimal solution, which can, for instance, be the best solution found so far, or an initial bound obtained by a heuristic method. Before we branch on a node v, we compute a lower bound LB on the solutions represented by this node; if $LB \geq UB$, single solutions coming from this subset of solutions cannot improve our current best solution, so we can prune this part of the tree and we do not have to branch from node v. This is the "bound" part of the BB method.

We use lower bounds on the communication volume of partial solutions to prune the BB tree. If we have a partial partitioning $\mathcal{B}$ of a matrix A, with part of the rows and columns assigned to processors or sets of processors, what can we say about the communication volume of a full solution that is an extension of this partial solution $\mathcal{B}$? We will answer this question by generalizing ideas on lower bounds for $k = 2$ from [3], [12] to the case $k \geq 2$.

A. Lower bounds based on explicit and implicit cuts

The most trivial bound is to use the assignments of the rows/columns that are already assigned to one or more processors. This gives the first lower bound L_1. Assume that $k = 3$, so that a partial solution of a matrix A can be

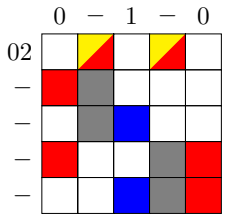

Fig. 4. A partial partitioning of a matrix. Next to each row and column are the numbers of the processors to which a row/column is assigned. For example, '0' indicates that the row/column is assigned to processor 0, while '-' indicates that the row/column is unassigned. If a nonzero has multiple colors, then it is assigned to the processors corresponding to those colors.

represented by $\mathcal{B} = \{B_0, B_1, B_2, B_{01}, B_{02}, B_{12}, B_{012}\}$. We know that all rows/columns that are assigned to 2 processors are already explicitly cut; these are the rows/columns in the sets B_{01}, B_{02}, B_{12}. The rows/columns in set B_{012} are assigned to 3 processors and have 2 explicit cuts. We can count the number of rows/columns in these sets and the number of cuts associated with them to obtain the first lower bound L_1 on the communication volume. So the explicit cuts of a partial partitioning give the lower bound L_1, and for $k = 3$ this equals

$$L_1(\mathcal{B}) = |B_{01}| + |B_{02}| + |B_{12}| + 2|B_{012}|. \tag{6}$$

Fig. 4 illustrates a partial assignment with a lower bound $L_1 = 1$ caused by row $0 \in B_{02}$.

The lower bound L_1 can easily be extended to the general case $k \geq 2$:

$$L_1(\mathcal{B}) = \sum_{x_1 \cdots x_r \subseteq \mathcal{P}} (r-1)|B_{x_1 \cdots x_r}|. \tag{7}$$

Here, $\mathcal{P} = \{0, 1, \ldots, k-1\}$ denotes the set of all processors, and $x_1 \cdots x_r$ a subset of r processors with $r \geq 2$.

Whereas the lower bound L_1 only counts explicit cuts, we can also consider *implicit cuts*, such as shown in the last row of Fig. 4. Although this row is unassigned, it has two nonzeros that have been assigned through their column, namely column 2 to processor 1 and column 4 to processor 0. Thus, the row must cause at least one communication. For each unassigned row, we can define a subset $x_1 \cdots x_r$ of processors that must be represented in that row because of the column assignments of the nonzeros of the row. With proper care, we can also count processors based on nonzeros with multiple colors: for a row with nonzeros assigned to 0, 12, we can either choose $x_1x_2 = 01$ or $x_1x_2 = 02$, and for a row with nonzeros assigned to 0, 12, 1, we must choose $x_1x_2 = 01$. All rows and columns for a particular choice of $x_1 \cdots x_r$ together define a set of unassigned rows/columns $B'_{x_1 \cdots x_r}$, which leads to the bound

$$L_2(\mathcal{B}') = \sum_{x_1 \cdots x_r \subseteq \mathcal{P}} (r-1)|B'_{x_1 \cdots x_r}|, \tag{8}$$

where the subsets must be of size $r \geq 2$. In Fig. 4, we have $L_2 = 1$. We can add bounds L_1 and L_2, since their sets of rows/columns are disjoint.

B. Lower bounds based on partially assigned rows/columns

A *partially assigned row* is an unassigned row that has at least one nonzero in a column that has already been assigned to a *strict subset* of the processors; a partially assigned column is defined similarly. So for $k = 3$, a partially assigned row has at least one nonzero in a column with one or two processors, but not three processors 012 since then that column does not impose a fixed assignment of a nonzero in the row. In this work, we will only take into account rows and columns that are partially assigned to one or two processors.

We say that an unassigned row is *partially assigned to processor* x if it has at least one nonzero in a column that is assigned to processor x and all other nonzeros in columns that are either unassigned or have been assigned to a set of processors that contains processor x. A similar definition holds for a partially assigned column. We define P_x as the subset of rows and columns that are partially assigned to processor x.

We say that an unassigned row is *partially assigned to processors* x *and* y, for $x \neq y$, if it has at least one nonzero in a column that is assigned to x, at least one nonzero in a column that is assigned to y, and all other nonzeros in columns that are either unassigned or have been assigned to a set of processors that contains x or y, or both. We also say that the row is partially assigned to processors x and y in case it has at least one nonzero in a column that is assigned to xy and has all other nonzeros in columns that are either unassigned or have been assigned to xy. A similar definition holds for a partially assigned column. We define P_{xy} as the subset of rows and columns that are partially assigned to processors x and y, for $x \neq y$.

If we need to make a distinction between rows and columns, we add a superscript: e.g., we write P_x^r for the subset of rows partially assigned to x and P_x^c for the subset of columns.

In Fig. 4, we have $P_0 = \{r_1, r_3\}$, $P_1 = \{r_2\}$, $P_{01} = \{r_4\}$, and $P_{02} = \{c_1, c_3\}$. Here, we denote the rows by $r_i, 0 \leq i < m$ and the columns by $c_j, 0 \leq j < n$.

We can exploit a partially assigned row in two different ways: either by checking whether we would violate the load balance constraint (4) if we would assign all its unassigned nonzeros to one of the processors already present in the row, or by checking whether it has an unassigned nonzero in a partially assigned column with a conflicting partial assignment. We can do this for a single row or a larger set of rows, and similarly also for a set of columns.

In Fig. 4, if we demand perfect load balance, so that the 12 nonzeros must be divided equally among the three processors, then processor 0 has already attained its maximum of 4 nonzeros and hence r_1 and r_3 must both be cut. This observation will be the basis for bound L_3, which is called a *packing bound* [3] because it considers the number of nonzeros that can be packed into a part.

The third lower bound, L_3, is determined by computing the maximum allowed number of nonzeros M of a part from (4) and then for each x cutting rows and removing them from P_x^r until the remaining rows can all be assigned to x within the load balance constraint. To make sure we find a lower bound,

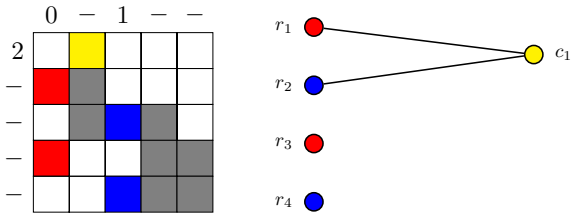

Fig. 5. A partial partitioning of a matrix for $k = 3$, with partially assigned rows $r_1, r_3 \in P_0$, $r_2, r_4 \in P_1$ and partially assigned column $c_1 \in P_2$, and the corresponding bipartite graph. The vertices have been colored according to their corresponding partial assignment. The vertices corresponding to rows $i \in P_0$ are red, those corresponding to rows $i \in P_1$ are blue, and the vertex corresponding to column $j \in P_2$ is yellow. An edge between r_i and c_j corresponds to an unassigned nonzero a_{ij}.

i.e., cut the least number of rows, we start by cutting the row with the largest number of nonzeros not assigned to x, and so on. We can do the same for the column sets P_x^c. The L_3 bound is the sum of the number of cuts for all sets $P_x^r, x \in \mathcal{P}$, and the number of cuts for the sets $P_x^c, x \in \mathcal{P}$.

The fourth lower bound, L_4, is obtained by matching rows and columns that conflict with each other because of an unassigned nonzero that they share. If row r_i is partially assigned to $x_1 \cdots x_r$ and column c_j to $y_1 \cdots y_s$, where these processor sets are disjoint, then assigning the still unassigned nonzero a_{ij} will give an extra cut in either the row or column. This is illustrated by Fig. 5, where the nonzeros a_{11} and a_{21} both cause a conflict. We only consider sets $x_1 \cdots x_r$ with $r \leq 2$ and $y_1 \cdots y_s$ with $s \leq 2$ to limit the cost of dynamically maintaining subsets $P_{x_1 \cdots x_r}$ during the BB algorithm. This is also the reason for our restrictive definitions of P_x and P_{xy}, which do not exploit all possible options.

To obtain a lower bound on the number of cuts based on direct conflicts, we cannot use the same row or column twice, meaning that we have to *match* the rows of the conflict nonzeros with the columns. Following [12], we can conveniently convert this problem to a bipartite matching problem for a graph $G = (V^r \cup V^c, E)$, where the vertices of V^r represent the rows and those of V^c the columns, and where the edges in E represent the conflicting nonzeros. The number of edges in an optimal (i.e., maximum) matching for G then becomes the lower bound L_4, which we call a *matching bound*. In Fig. 5, we choose one of the two edges in the matching, and hence $L_4 = 1$.

The matching bound can be improved for $k > 2$ by also exploiting indirect conflicts. Such a conflict occurs for instance when a row $r_i \in P_x$ and a row $r_j \in P_y$ have an unassigned nonzero in the same column $k \in P_z$. This situation increases the communication volume by two, but when we look at a maximum matching we will only increment by one. In Fig. 5, either two of row r_1, r_2 and column c_1 must be cut, or column c_1 must be cut twice. In the graph formulation, this can be taken into account by splitting the vertices of the bipartite graph. We formally split each vertex $v \in P_x$ into $k-1$ vertices v^y, with $y \in \mathcal{P} - \{x\}$, and we create an edge between vertices

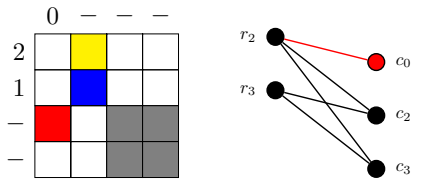

Fig. 6. A partial partitioning of a matrix for $k = 3$ and the neighborhood of row $r_2 \in P_0$, shown as black edges and vertices. The red color of vertex c_0 means that column 0 has been assigned to processor 0; the red color of edge (r_2, c_0) means that nonzero a_{20} has been assigned to processor 0. All the edges in the neighborhood need to be colored red to avoid a cut in a row or column in the partial partitioning.

r_i^x and c_j^y if there is an unassigned nonzero a_{ij} with $r_i \in P_y$ and $c_j \in P_x$. We discard degree-0 vertices. In Fig. 5, we obtain two edges, (r_1^2, c_1^0) and (r_2^2, c_1^1), which are not adjacent and hence both end up in the matching, giving a better bound $L_4 = 2$.

Packing bound L_3 and matching bound L_4 both use partially assigned rows and columns and hence they might use the same rows or columns. Therefore we cannot add these bounds, but instead must take their maximum as the combined bound $\max(L_3, L_4)$. An alternative is first to compute L_4, then remove all the matched rows and columns, and after that compute L_3 for the remaining rows and columns. This gives a combined bound L_5. The best bound is then $\max(L_3, L_4, L_5)$.

C. Global lower bounds

The packing bound L_3 is local because it only takes a single row or column into account and not the rows or columns that are connected to it through intersections in a nonzero. We can turn L_3 into a global bound GL_3 by expanding the neighborhood of a partially assigned row or column. This is illustrated in Fig. 6, which shows the neighborhood of row $r_2 \in P_0$. All four edges of the neighbourhood must be assigned to processor 0 to avoid a cut.

In the following, we generalize the definition of a neighborhood from [3] to the case $k > 2$, although for $k = 2$ our definition differs slightly. We define a *neighborhood* (V, E) adjacent to processor x as a subset V of the vertices and a subset E of the edges of the whole bipartite graph that satisfies the following requirements:

1) for $v \in V$ with $v \notin P_x$, all edges of v are unassigned;
2) if u is an end point of two distinct edges in E, then $u \in V$;
3) (V, E) is path-connected with respect to edges;
4) V contains at least one vertex $v \in P_x$;
5) all edges $e \in E$ are either unassigned or assigned to a set of at least two processors that contains x.

If we have a neighborhood (V, E) adjacent to vertex $v \in P_x$, then to avoid extra cuts we need to assign all edges in E to processor x. As was the case for the L_3 bound this can only be done if the load balance constraint (4) allows this; otherwise we need to assign at least some of the edges to a different processor. This means that at least one of the vertices in V is cut.

To determine the GL_3 bound, we try to find neighborhoods for each partially assigned row/column in P_x. These neighborhoods need to be pairwise disjoint, i.e., vertex disjoint. If we have found a set of neighborhoods $(V_1, E_1), \ldots, (V_l, E_l)$ adjacent to part x, then to prevent cuts in the future we need to assign all edges of these neighborhoods to processor x. However, if load balance does not allow this, we need to cut some of the neighborhoods, where we start by cutting the neighborhood with the largest number of edges. We repeat this until the load balance constraint is satisfied. Each cut neighborhood leads to one extra communication volume. We can do this for every $x \in \mathcal{P}$, and sum over all neighborhoods that need to be cut to get the global packing bound GL_3.

The matching bound L_4 is local because it only takes the immediate vicinity of a conflict nonzero into account. In the graph, we can view the edge corresponding to the conflict nonzero as a path of length 1. We can turn L_4 into a global bound GL_4 by also taking conflicts at longer distances into account. This is illustrated in Fig. 7, which shows two conflict paths that start in a partially assigned row $r_4 \in P_1$ and end in different sets P_0 and P_1, thus giving $GL_4 = 2$. We can find a set of such conflict paths starting in a vertex $v \in P_x$ by performing a breadth-first search (BFS) from v, where we search for end points that are each in a different set P_y.

We can repeat this process, each time starting a BFS from a new vertex v, where we have to make sure that new paths are internally vertex disjoint with all paths produced by previous searches; end points of previous paths are allowed to be used as a starting point or ending point of a new path. Here, we have to exclude paths ending in P_y if v has already been connected to P_y by a previous search where $u \in P_y$ is the start of a path and v the end point. In our implementation, we generalize the searches to include also sets of the form P_{xy}; for details see [16].

At most we will perform a BFS at cost $\mathcal{O}(|V| + |E|)$ for all vertices in V. So the computation of GL_4 costs at most $\mathcal{O}((m+n)(m+n+nz(A)))$. This is, however, a very pessimistic upper bound on the computation time. In practice, the computation time will be far less. We observed that this bound is beneficial and it enabled us to find optimal 3-way and 4-way partitionings for more matrices than if we had only used the local matching bound L_4.

The global bounds GL_3 and GL_4 might use the same rows or columns, and hence these bounds cannot be added. Still, they can be combined in a similar fashion as was done for L_3 and L_4, first determining GL_4 and then computing GL_3 on a subset of the rows and columns, avoiding those that were used by GL_4. For the sake of brevity, we omit the details; see [16].

III. Integer linear programming

In section II, we described an exact BB method to solve the matrix partitioning problem. Another way to solve this problem exactly is to formulate it as an *integer linear program* (ILP), which has the following form:

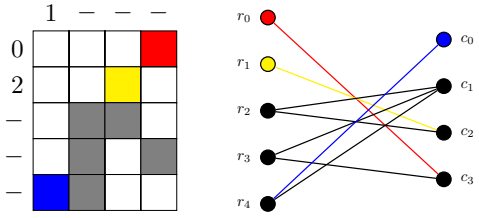

Fig. 7. A partial partitioning of a matrix and the corresponding bipartite graph. The path r_4, c_1, r_2, c_2 leads to a conflict between the partial assignment of $r_4 \in P_1$ and $c_2 \in P_2$; the path r_4, c_1, r_3, c_3 leads to a conflict between $r_4 \in P_1$ and $c_3 \in P_0$. Vertices have been colored according to the corresponding assignment of their rows or columns. Black vertices correspond to unassigned rows or columns. Similar for the edges and their corresponding nonzeros.

$$\begin{aligned} \min \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{s.t.} \quad & C\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \in \mathbb{Z}^n, \end{aligned} \tag{9}$$

with given vectors $\mathbf{c}, \mathbf{b}$ and constraint matrix C.

It is known that integer linear programming is NP-complete [17]. Still, several software packages exist that can solve ILPs of small and medium size, although we cannot expect these packages to be able to solve large instances optimally in general. One of these software packages is CPLEX [18], which we will use in this work. The ability to solve an ILP highly depends on a suitable formulation of the ILP.

Our ILP approach is to formulate the sparse matrix partitioning problem as a hypergraph partitioning problem by the fine-grain model [15], and then convert the hypergraph problem to an ILP.

A *hypergraph* $H = (V, N)$ is a set of vertices with a set of *nets* or *hyperedges*, which are subsets of V. Whereas in a graph edges connect two vertices, in a hypergraph the hyperedges may connect an arbitrary number of vertices. In the fine-grain model, each vertex represents a nonzero a_{ij} of the sparse matrix A; the nonzeros of row i are converted into a row-net r_i and those of column j into a column-net c_j. This hypergraph has $nz(A)$ vertices and $m + n$ nets. Each vertex is contained in exactly two nets.

A k-way partitioning of the vertices of the hypergraph with a load balance constraint induced by (4) and a communication volume induced by (5) directly corresponds to a solution of the sparse matrix partitioning problem with the same load balance and communication volume [15].

For our ILP, we define the following decision variables:

$$\begin{aligned} x_{is} &= \begin{cases} 1 & \text{if vertex } i \text{ is in part } s, \\ 0 & \text{otherwise,} \end{cases} \\ y_{js} &= \begin{cases} 1 & \text{if net } j \text{ has vertices in part } s, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{10}$$

We also define M as the maximum number of nonzeros allowed in a part by the load balance criterion (4).

Using these variables, our ILP formulation becomes:

$$\min \quad \sum_{j=0}^{|N|-1} \Big(\sum_{s=0}^{k-1} y_{js} - 1\Big) \tag{11}$$

$$\text{s.t.} \quad \sum_{s=0}^{k-1} x_{is} = 1 \qquad \forall i \tag{12}$$

$$\sum_{i=0}^{|V|-1} x_{is} \leq M \qquad \forall s \tag{13}$$

$$x_{is} \leq y_{js} \qquad \forall j, s, i, \text{with } i \in \text{net } j \tag{14}$$

$$x_{00} = 1 \tag{15}$$

$$y_{js} \in \{0, 1\} \qquad \forall j, s \tag{16}$$

$$x_{is} \in \{0, 1\} \qquad \forall i, s \tag{17}$$

The minimization in (11) counts for every net j the number of parts that contain vertices of net j, i.e. $y_{j0} + \cdots + y_{j,k-1} = \lambda_j$, so that net j contributes $\lambda_j - 1$ to the communication volume of the partitioned hypergraph. Furthermore, (12) expresses that each vertex can only be in one part; (13) is the load balance constraint; (14) takes care that a vertex i in a net j and in part s forces the net variable y_{js} to be 1; (15) exploits symmetry (albeit to a very limited extent).

Our ILP formulation has $nz(A) \cdot k$ variables x_{is} and $(m + n) \cdot k$ variables y_{js}, so in total $k(nz(A) + m + n)$ decision variables. There are $nz(A)$ vertex constraints and k load balance constraints. Furthermore, there are $2 \cdot k \cdot nz(A)$ net constraints, because every nonzero appears in two net constraints: one for its row and one for its column. Therefore, the total number of constraints is $nz(A) + k(2 \cdot nz(A) + 1)$.

IV. Recursive Bipartitioning

Recursive bipartitioning (RB) is used in many heuristic solvers to obtain a k-way partitioning of a matrix. It works as follows: we start with the set of all $nz(A)$ nonzeros, which we then split into two subsets. Each subset is subsequently split into two subsets, resulting in four subsets. We repeat the splitting of every subset until there are k subsets, meaning we have obtained a k-way partitioning. Therefore, if $k = 2^l$, there are l levels at which we split the subsets. For simplicity, we study RB for the case where k is a power of 2, but the method can be adapted to other values of k.

We will study the RB method that uses an exact method to bipartition a subset, within the given load balance constraint and in such a way that it minimizes the communication volume that arises. This RB method is greedy: although it bipartitions a subset optimally, it does not take into account the subsequent bipartitionings while doing this. Therefore, we ask ourselves in how far the RB method is able to approach the minimal communication volume that is possible for a k-way sparse matrix partitioning.

Our question on the quality of RB has been posed before by Simon and Teng [19] for graph partitioning with the total *edge cut* as cost metric, which differs from the incurred communication volume that we have. They showed that RB may produce

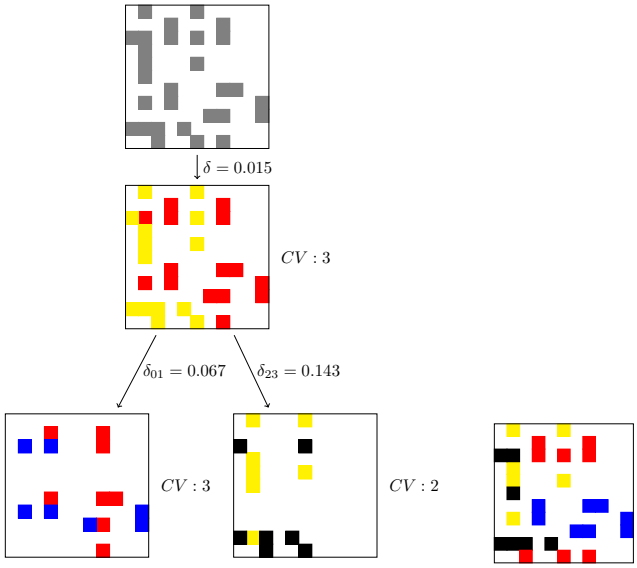

Fig. 8. Recursive bipartitioning of the 11×11 matrix `Tina_AskCal` from [4] with $nz(A) = 29$ (left), and an optimal 4-way partitioning (right), both with $\epsilon = 0.03$. The first bipartitioning step splits the nonzeros into two subsets, A_{01} (red) and A_{23} (yellow). During the second bipartitioning step, the nonzeros of subset A_{01} are split into subsets A_0 (red) and A_1 (blue), and those of A_{23} into subsets A_2 (yellow) and A_3 (black). Next to the arrows is the value of the load imbalance parameter used in each split, and next to each split subset is the communication volume arising from this split. For RB, this leads to $CV = 3 + 3 + 2 = 8$ and for the optimal partitioning to $CV = 7$.

a k-way graph partitioning that is very far from optimal, although for well-shaped meshes it is within a constant factor of optimal; note that mesh problems form a class of problems for which edge cut is a good approximation of communication volume and where graph partitioning is often used instead of hypergraph partitioning. We will try to answer the question for the sparse matrix partitioning problem with communication volume as cost metric, and we will do this by comparing the results obtained by the recursive bipartitioning of a matrix A for the case $k = 4$ with the optimal value that was determined by using direct 4-way partitioning. Fig. 8 illustrates this.

It has been shown that the communication volumes of subsequent splits in a general partitioning method for parallel SpMV are additive; a proof can be found in [5], [20]. For RB, this means that

$$CV(A_0, A_1, A_2, A_3) = CV(A_{01}, A_{23}) + CV(A_0, A_1) + CV(A_2, A_3). \quad (18)$$

Even though each bipartitioning is performed optimally and without looking ahead to further splits, the partitioning strategy has not been completely determined yet: we still have some freedom in choosing the imbalance parameter of every separate split. Here, we will use the adaptive strategy of the Mondriaan package [5], which is to assume at a given level the same imbalance parameter δ for all remaining splits, so that

$$1 + \epsilon = (1 + \delta)^l \approx 1 + l\delta, \quad (19)$$

giving the choice $\delta \approx \epsilon/l$. Note that for a lowest-level split, with $l = 1$, the approximation is exact. As an alternative, we could use the exact value for δ that can be obtained from (19), which is done in KaHyPar [10]. We also adjust ϵ to the current number of nonzeros in the part to be split. In Fig. 8, $nz(A_{01}) = 15$ and $nz(A_{23}) = 14$, so that the next split of A_{01} has to be tighter than that of A_{23}, with a smaller value of δ.

V. Experimental results

We have implemented the BB method from section II in the C++ programming language, in a program called General Matrix Partitioner (GMP). The results of our computations are available through the MondriaanOpt page.[1] The source code of the GMP method and the ILP method are available on GitHub.[2]

We partitioned a test set of relatively small sparse matrices from the SuiteSparse collection [4] for $k = 2, 3, 4$ and $\epsilon = 0.03$ using GMP as well as our ILP partitioner implemented using IBM ILOG CPLEX 20.1 [18]. We set the value of the parameter controlling the thread count of CPLEX to one, to let it use only one processor. This seemed fair since the other methods in our comparisons also use one processor. For all the other parameters we used the default settings of CPLEX. The CPLEX software uses a branch-and-cut method. For $k = 2$, we also used the bipartitioners MondriaanOpt [12] and MP [3]. We performed our experiments on a computer system with an AMD 3800XT processor with 8 Cores and 16 threads running at 4.3GHz, and 16 GB RAM.

The initial upper bound UB was obtained for MondriaanOpt by using the heuristic partitioner Mondriaan with the default medium-grain method. For the other methods, we used iterative deepening by starting at $UB = 1$ and then if no feasible solution was found in a run of the partitioner perform a next iteration with upper bound $\lceil 1.25 \cdot UB \rceil$.

The lower bounds that we used in our implementation were tried in order of increasing computing cost, $LB = L_1 + L_2, L_1 + L_2 + L_3, L_1 + L_2 + L_5, L_1 + L_2 + GL_5$; once $LB \geq UB$, we can prune the subtree and need not compute further bounds. In the cheaper local bounds we only consider sets P_x, whereas in the more expensive global bounds we also consider sets P_{xy}.

To execute the BB method, we need to decide on an order of the rows and columns to branch on; we determine this order beforehand. There are many possible orders and the choice of a specific order can have a dramatic influence on the performance. It seems natural to start with a row or column with the largest number of nonzeros, since its assignment has the largest influence on the load balance and on partial assignments of other rows and columns. After choosing a row or column in this ordering, we remove its nonzeros from the matrix before choosing the next row or column. This was found to be a good strategy for $k = 2$ in [13], where

[1] https://webspace.science.uu.nl/~bisse101/Mondriaan/Opt/
[2] https://github.com/lienjenns/Thesis_Matrix_Part

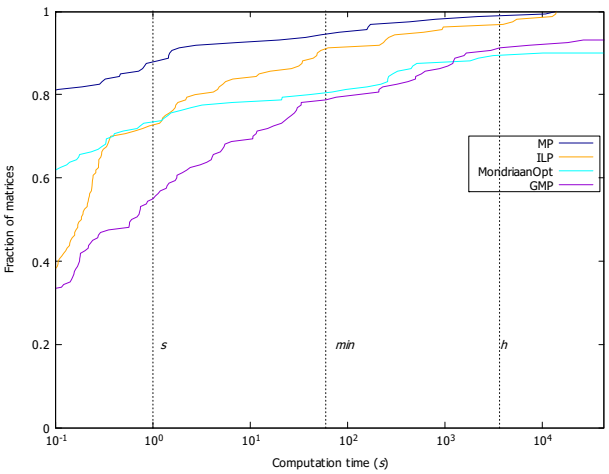

Fig. 9. Performance profile showing the fraction of all the matrices with $nz \leq 500$ from [4] that could be partitioned for $k = 2$ within the given computation time, for four different methods. The test set consists of 160 matrices. Note the logarithmic scale for the computation time. The vertical lines denote seconds, minutes, and hours.

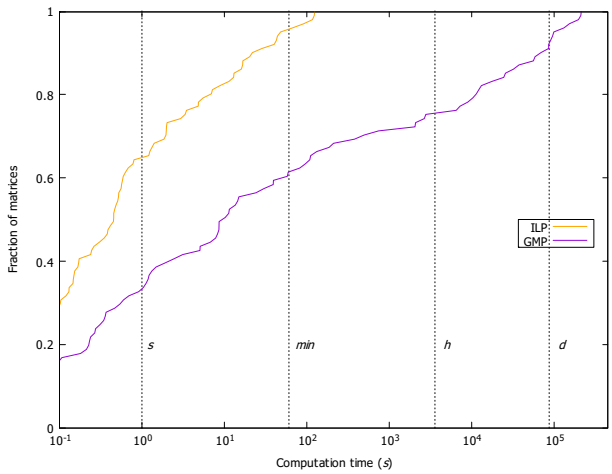

Fig. 10. Performance profile showing the fraction of matrices from our test set that could be partitioned for $k = 3$ within the given computation time, for the two general partitioning methods. The test set consists of 101 matrices with $nz \leq 1000$. The vertical lines denote seconds, minutes, hours, and days.

MondriaanOpt was used with different orders. If this order failed, we used a static branching strategy with alternating rows and columns in decreasing order of nonzero count.

Furthermore, if we branch on a chosen row or column, and multiple assignments are possible, each with its own associated subtree, we need to decide which subtree we will traverse first. We do this in increasing order of the number of processors in the assignment. We break ties by preferring processors with the lowest assigned number of nonzeros.

For $k = 2$, we compared the three BB methods MondriaanOpt, MP, GMP, and the ILP method. Fig. 9 shows a performance profile for $k = 2$ for the 160 matrices from the SuiteSparse collection with at most 500 nonzeros. The matrix partitioner MP is the fastest of these four methods, and it was able to find the optimal bipartitioning of all matrices within the set time limit of 12 hours. The ILP method also achieved this within the time limit, but it was slower than MP. Thus we see that for $k = 2$ a specialized BB partitioner can outperform a general commercial ILP solver. The ILP solver seems to have some startup overhead as for easy problems that can be solved within a second it does not perform that well. The GMP method is outperformed by the other methods, but it is still able to solve 149 out of 160 problems; MondriaanOpt solves 144 problems. GMP has some overhead because it allows partitioning for any k.

For $k = 3, 4$ we tried to solve as many matrices as possible with at most 1000 nonzeros using GMP, and we also compared GMP with ILP. For $k = 3$, we first tried to find the optimal communication volume for the 60 matrices of the SuiteSparse Matrix Collection with the fewest nonzeros. We gave the GMP partitioner at least 48 hours of computation time to try to find any feasible solution within 48 hours. If this succeeded, we then let GMP run for at least 5 more days to see if it could find the optimal communication volume. For other matrices, we focused on those with small CV that were partitionable for $k = 2$ in at most a few seconds. We noticed that both the CV of the optimal bipartitioning and the computation time needed to determine it gave us a good indication whether we would be able to find an optimal 3-way partitioning. In general, the higher the optimal CV and the computation time for $k = 2$, the longer it takes to determine an optimal 3-way partitioning. With this approach, GMP succeeded for 101 matrices.

For $k = 4$, we used the same approach but instead of taking the results for $k = 2$ as an indicator for possible success we used the results for $k = 3$ and focused on the matrices that have a small CV and were 3-way partitionable in at most a few minutes. Here, GMP succeeded for 62 matrices.

The optimal communication volumes and computation times of the partitioned matrices for $k = 3, 4$ can be found in [16, Appendix A] and they will also become available on the MondriaanOpt page.

Figs. 10 and 11 show performance profiles for $k = 3, 4$. ILP solves all the problems that GMP can solve, and does this much faster. Using the geometric mean of the speed ratios for the matrices that both can solve, we found that ILP is $4.3\times$ faster for $k = 2$, $51.3\times$ for $k = 3$, and $182.9\times$ for $k = 4$. The far superior performance of ILP for $k = 3, 4$ was surprising to us, given that for $k = 2$ the BB solvers MondriaanOpt and MP are faster than ILP, by factors $3.1\times$ and $39.5\times$, respectively, which confirms our experience with other ILP implementations in the past.

We have two possible explanations for the better performance of the ILP approach. First, the commercial software package CPLEX has made progress over the years, incorporating many optimizations and new algorithms. This makes it harder for a specialized problem-specific BB implementation to beat it. Second, for $k > 2$, exploiting specific bounds

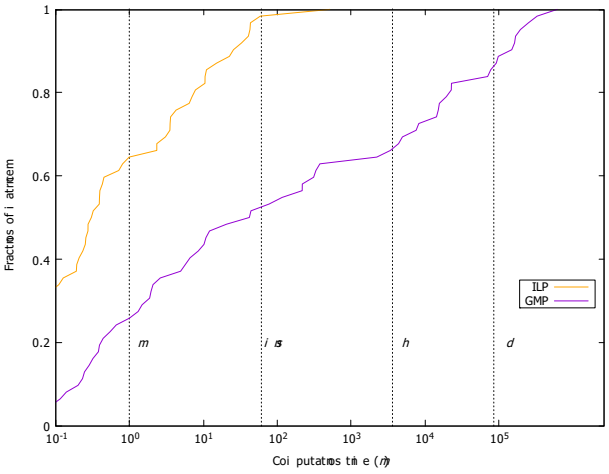

Fig. 11. Performance profile showing the fraction of matrices from our test set that could be partitioned for $k = 4$ within the given computation time, for the two general partitioning methods. The test set consists of 62 matrices with $nz \leq 1000$. The vertical lines denote seconds, minutes, hours, and days.

becomes more complicated and the right balance needs to be found between a sophisticated bound that is costly to compute and a simpler bound that is cheap. GMP tries to find this balance, e.g. by limiting partially assigned rows and columns to sets with one or two processors, but it might miss out on certain useful larger sets of processors. An advantage of GMP is that it uses much less memory than ILP; we observed differences of several orders of magnitude.

Having an exact 4-way partitioner allows us to study the quality of the recursive bipartitoning method used in many heuristic sparse matrix partitioners. We did this by the method described in section IV: we set $\epsilon = 0.03$, perform the first split with $\delta = 0.015$, and then perform the next two splits each with their own δ, according to the number of nonzeros of the two parts resulting from the first split. Here, we used MP as the exact bipartitioner. We also optimally partitioned the matrices of our test set into 4 parts by either GMP or ILP.

Tables I and II show the optimal communication volumes for $k = 2, 3, 4$, and the volume obtained by the RB method for $k = 4$ with exact bipartitioning. The test set of 89 matrices consists of all the sparse matrices with less than 250 nonzeros from the SuiteSparse collection [4], except the matrices `Trec3`, `mycielskian2`, `Trec4`, which are too small with $nz \leq 3$, and the difficult matrices `ibm32`, `mycielskian5`, `ch4-4-b1`, `Trefethen20b`, `Trefethen20`, `wheel_5_1`, `cage5`, `Maragal_1`, and `n3c5-b7`, which could not be solved by our strategy.

The results in Tables I and II show that the communication volume of RB is always close to the optimal value obtained for $k = 4$, and often it equals that value, namely in 46 out of 89 cases. In 34 cases the difference in CV is only 1; in 8 cases it is 2; and in one case it is 3, namely for the matrix `wheel_3_1`. This means that RB performs well as a heuristic strategy.

TABLE I

COMMUNICATION VOLUMES FOR MATRICES WITH AT MOST 150 NONZEROS FROM THE SUITESPARSE COLLECTION [4]. SHOWN ARE THE NUMBER OF ROWS, COLUMNS, AND NONZEROS, THE OPTIMAL VOLUMES FOR $k = 2, 3, 4$, AND THE VOLUME OBTAINED BY RECURSIVE BIPARTITIONING FOR $k = 4$ WITH EXACT BIPARTITIONING. THE LOAD IMBALANCE WAS $\epsilon = 0.03$ AND THE FIRST SPLIT WAS WITH $\delta = 0.015$.

				Communication volume			
Matrix	**m**	**n**	**nz**	**k=2**	**k=3**	**k=4**	**RB**
GL7d10	1	60	8	1	2	3	3
mycielskian3	5	5	10	2	3	4	4
Trec5	3	7	12	2	4	7	7
b1_ss	7	7	15	3	4	5	5
ch3-3-b2	6	18	18	0	0	2	2
rel3	12	5	18	3	6	10	11
cage3	5	5	19	4	7	9	9
lpi_galenet	8	14	22	2	3	4	4
relat3	12	5	24	3	8	9	9
lpi_itest2	9	13	26	3	4	6	6
lpi_itest6	11	17	29	2	3	5	5
Tina_AskCal	11	11	29	3	6	7	8
n3c4-b1	15	6	30	5	6	9	10
n3c4-b4	6	15	30	5	6	9	9
ch3-3-b1	18	9	36	5	6	9	9
Tina_AskCog	11	11	36	4	6	9	9
GD01_b	18	18	37	1	2	3	4
mycielskian4	11	11	40	6	10	12	12
Trec6	6	15	40	5	8	10	11
farm	7	17	41	4	7	10	11
Tina_DisCal	11	11	41	5	9	11	12
kleemin	8	16	44	6	8	11	12
LFAT5	14	14	46	4	4	10	10
bcsstm01	48	48	48	0	0	0	0
Tina_DisCog	11	11	48	6	9	13	14
cage4	9	9	49	9	12	16	17
GD98_a	38	38	50	0	3	4	4
jgl009	9	9	50	5	10	14	15
GD95_a	36	36	57	1	1	2	2
klein-b1	30	10	60	5	8	12	12
klein-b2	20	30	60	6	9	11	11
n3c4-b2	20	15	60	9	15	18	19
n3c4-b3	15	20	60	9	15	18	19
Ragusa18	23	23	64	5	9	12	13
bcsstm02	66	66	66	0	0	0	0
lpi_bgprtr	20	40	70	4	6	8	9
wheel_3_1	21	25	74	8	13	16	19
jgl011	11	11	76	7	11	16	17
rgg010	10	10	76	8	12	18	18
Ragusa16	24	24	81	7	12	15	16
LF10	18	18	82	4	8	12	12
problem	12	46	86	2	5	6	7
GD02_a	23	23	87	7	12	15	16
Stranke94	10	10	90	10	18	20	20
n3c5-b1	45	10	90	8	10	15	17
ch4-4-b3	24	96	96	0	0	0	0
GD95_b	73	73	96	2	2	3	5
Hamrle1	32	32	98	5	10	13	14
lp_afiro	27	51	102	5	7	11	11
rel4	66	12	104	5	8	13	14
bcsstm03	112	112	112	0	0	0	0
p0033	15	48	113	5	9	12	13
football	35	35	118	8	13	19	20
n4c5-b11	10	120	120	0	2	2	2
GlossGT	72	72	122	5	8	10	12
wheel_4_1	36	41	122	12	18	21	22
bcspwr01	39	39	131	6	8	10	12
bcsstm04	132	132	132	0	0	0	0
p0040	23	63	133	3	8	13	13
GD01_c	33	33	135	7	11	17	18
bcsstm22	138	138	138	0	0	0	0
lpi_woodinfe	35	89	140	0	0	6	6
Trec7	11	36	147	8	13	20	22
lp_sc50b	50	78	148	5	9	11	12
GD99_c	105	105	149	0	1	2	2
d_ss	53	53	149	4	9	12	12

TABLE II
COMMUNICATION VOLUMES FOR MATRICES WITH A NUMBER OF NONZEROS BETWEEN 150 AND 250 FROM THE SUITESPARSE COLLECTION [4], CONTINUED FROM TABLE I.

				Communication volume			
Matrix	**m**	**n**	**nz**	**k=2**	**k=3**	**k=4**	**RB**
bcsstm05	153	153	153	0	0	0	0
refine	29	62	153	3	6	10	10
karate	34	34	156	8	14	18	19
can_24	24	24	160	8	16	20	20
lp_sc50a	50	78	160	5	7	11	13
bcspwr02	49	49	167	4	10	14	14
lap_25	25	25	169	10	18	22	22
relat4	66	12	172	4	9	12	13
pores_1	30	30	180	9	17	22	23
GD96_b	111	111	193	3	4	7	7
GD98_b	121	121	207	0	0	0	0
n2c6-b1	105	15	210	11	15	21	22
n3c6-b1	105	105	210	11	15	21	22
n4c5-b1	105	15	210	11	15	21	22
can_62	62	62	218	6	10	14	16
dwt_72	72	72	222	4	8	12	12
divorce	50	9	225	8	16	23	24
GD96_d	180	180	229	0	0	0	0
GD02_b	80	80	232	5	10	13	13
d_dyn	87	87	238	5	10	15	15
d_dyn1	87	87	238	5	10	15	15
lpi_forest6	66	131	246	5	10	12	13
Sandi_authors	86	86	248	4	8	10	12

VI. CONCLUSION AND OUTLOOK

In this work, we have developed a branch-and-bound (BB) method for k-way partitioning of sparse matrices that can handle any value of k. We have implemented this in a C++ program called General Matrix Partitioner (GMP), and tested it on a set of matrices with up to 1000 nonzeros, for $k = 2, 3, 4$. As a result, we can now provide a database of optimal partitionings that can be used as a benchmark for heuristic solvers.

We have also formulated the sparse matrix partitioning problem as an integer linear programming (ILP) problem and used the CPLEX package to solve it. This approach proved superior for $k > 2$.

For future work, the performance of the ILP approach indicates that it is a viable alternative to BB for exact partitioning and that it deserves to be explored further, for instance by finding better ILP formulations based on the fine-grain hypergraph model. This may also be an approach for heuristic partitioning, perhaps in combination with other methods such as the medium-grain method; this would lead to a different hypergraph that can be fed into an ILP solver.

We tried to answer the question "How good is recursive bisection?" [19] by studying recursive bipartitioning (RB) with an optimal split in every bisection, comparing it with an optimal 4-way partitioning. Our answer is "Quite good in practice", even though we could study this only for relatively small problems. For future work, it would be worthwhile to investigate the freedom that RB still has when using exact bipartitioning, namely the strategy for choosing the load imbalance parameter for the next split.

REFERENCES

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed. SIAM, Philadelphia, PA, 1994.

[2] J. Kepner and J. Gilbert, Eds., *Graph Algorithms in the Language of Linear Algebra*. SIAM, Philadelphia, PA, 2011.

[3] T. E. Knigge and R. H. Bisseling, "An improved exact algorithm and an NP-completeness proof for sparse matrix bipartitioning," *Parallel Computing*, vol. 96, p. 102640, 2020.

[4] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, pp. 1:1–1:25, 2011.

[5] B. Vastenhouw and R. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Review*, vol. 47, no. 1, pp. 67–95, 2005.

[6] D. M. Pelt and R. H. Bisseling, "A medium-grain method for fast 2D bipartitioning of sparse matrices," in *Proceedings 28th International Parallel and Distributed Processing Symposium (IPDPS 2014)*. IEEE Press, 2014, pp. 529–539.

[7] U. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[8] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *Proceedings 36th ACM/IEEE Conference on Design Automation*. ACM Press, 1999, pp. 343–348.

[9] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag, "Engineering a direct k-way hypergraph partitioning algorithm," in *19th Workshop on Algorithm Engineering and Experiments, (ALENEX 2017)*, 2017, pp. 28–42.

[10] S. Schlag, T. Heuer, L. Gottesbüren, Y. Akhremtsev, C. Schulz, and P. Sanders, "High-quality hypergraph partitioning," *CoRR*, vol. abs/2106.08696, 2021.

[11] K. D. Devine, E. G. Boman, R. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *Proceedings 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*. IEEE Press, 2006, p. 102.

[12] D. M. Pelt and R. H. Bisseling, "An exact algorithm for sparse matrix bipartitioning," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 79–90, 2015.

[13] A. Mumcuyan, B. Usta, K. Kaya, and H. Yenigün, "Optimally bipartitioning sparse matrices with reordering and parallelization," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 21, p. e4687, 2018.

[14] B. Usta, "Optimal hypergraph partitioning," Master's thesis, Sabancı University, Aug. 2018.

[15] U. V. Çatalyürek and C. Aykanat, "A fine-grain hypergraph model for 2D decomposition of sparse matrices," in *Proceedings 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, 2001, pp. 1199–1204.

[16] L. Jenneskens, "Beyond bipartitioning: Exact sparse matrix partitioning into multiple parts," Master's thesis, Utrecht University, Dec. 2021.

[17] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice Hall, 1982.

[18] "IBM ILOG CPLEX Optimization Studio 20.1.0." [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio

[19] H. D. Simon and S.-H. Teng, "How good is recursive bisection?" *SIAM Journal on Scientific Computing*, vol. 18, no. 5, p. 1436–1445, 1997.

[20] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach Using BSP*, 2nd ed. Oxford University Press, Oxford, UK, 2020.