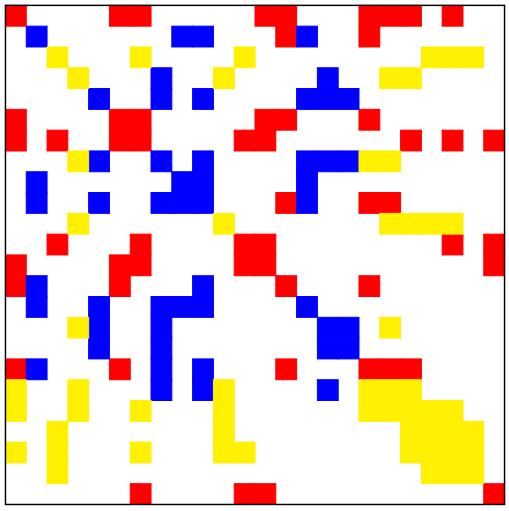
Beyond bipartitioning: Exact sparse matrix partitioning into multiple parts.

Master Thesis Mathematical Sciences



Lieneke Jenneskens

December 3, 2021

Utrecht University

First supervisor:
Prof. dr. R.H. Bisseling
Second supervisor:
Dr. Ivan Kryven

Cover image: An optimal $3-$way partitioning of the 24×24 matrix `can_24`, containing 124 nonzeros. The load imbalance parameter was set to $\epsilon = 0.03$ and the optimal communication volume is equal to 16.

Abstract

In order to minimize the communication in parallel sparse matrix-vector multiplications we need to solve the sparse matrix partitioning problem, that is the partitioning of a sparse matrix A into p disjoint parts, while minimizing the communication volume. This problem is NP-Complete.

In this thesis we present an algorithm based on the branch and bound method which optimally partitions a matrix into p disjoint parts, with $p \in \mathbb{N}_{\geq 2}$. We also present an ILP model which solves the sparse matrix partitioning problem. We used both methods in order to determine optimal $2, 3, 4-$way partitionings for a subset of matrices from the SuiteSparse Matrix Collection with 1000 nonzeros or less. The ILP model outperformed the branch and bound method for $p > 2$, as it was both able to find optimal partitionings for a greater number of matrices, and was able to do this faster than the branch and bound method.

We used the results found by these exact methods for $p = 4$ to analyse the performance of recursive bipartitioning (RB). For 57 matrices of the 100 matrices in our test set, the communication volume determined by the recursive bipartitioning method was equal to the optimal communication volume. For 97% of the matrices in our test set, the recursive bipartitioning method is able to find $4-$way partitionings with communication volume at most 1.2 times the optimal communication volume of a $4-$way partitioning.

Contents

Nomenclature

m	The number of rows in matrix A.
n	The number of columns in matrix A.
$nz(A)$	The number of nonzeros in a matrix A.
p	The number of processors.
$\mathscr{P}$	The set of the p processors $\{0, \cdots, p-1\}$.
M	The maximum allowed size of a part.
$CV()$	Communication volume of a (partial) partitioning.
n_i	Number of nonzeros owned by processor i.
s_i	Number of nonzeros that we need to assign to processor i to prevent a cut in a row or column.
$x_1 \cdots x_k$	State of a row/column. The row/column is assigned to k processors, $x_1, \cdots, x_k \in \mathscr{P}$.
$B_{x_1 \cdots x_k}$	Set of the rows and columns of a full partitioning that are assigned to the k processors $x_1, \cdots, x_k$. If $p = 3$, there are 7 such sets, B_0, B_1, B_2, B_{01}, B_{02}, B_{12}, B_{012}.
$\hat{B}_{x_1 \cdots x_k}$	Set of the rows and columns of a partial partitioning that are assigned to the k processors $x_1, \cdots, x_k$.
$\mathcal{B}$	Full partitioning of a matrix, with $\mathcal{B} = \bigcup_{x_1 \cdots x_k \subseteq \mathscr{P}} B_{x_1 \cdots x_k}$.
$\hat{\mathcal{B}}$	Partial partitioning of a matrix, with $\hat{\mathcal{B}} = \bigcup_{x_1 \cdots x_k \subseteq \mathscr{P}} \hat{B}_{x_1 \cdots x_k}$.
P_x	Set of rows and columns that are partially assigned to processor x.
P_x^r	The rows that are partially assigned to processor x.
P_x^c	The columns that are partially assigned to processor x.
P_{xy}	Set of rows and columns that are partially assigned to the two processors x and y.
P_{xy}^r	The rows that are partially assigned to the two processors x and y.
P_{xy}^c	The columns that are partially assigned to the two processors x and y.
L_i	Local lower bound i.
GL_i	Global lower bound i.

1 Introduction

Matrix-vector multiplication is an operation that often appears in scientific computing algorithms. In many cases the matrix that is multiplied is sparse. A matrix is sparse if most of the entries of the matrix are zero or in other words an $m \times n$ matrix A, with entries a_{ij} with $1 \leq i \leq m$ and $1 \leq j \leq n$, that has $nz(A)$ nonzeros is sparse if $nz(A) \ll mn$. An example of a sparse matrix is the adjacency matrix that represents the graph of a social network. We can exploit the sparsity of a sparse matrix to speed up computation and to store the matrix more efficiently, because of the fact that we can basically ignore the zero entries when storing the matrix or performing computations. If we multiply a regular $m \times n$ matrix with a $1 \times n$ vector it takes $O(nm)$ time, while if this matrix is sparse with $nz(A)$ nonzeros the multiplication can be done in $O(nz(A))$ time.

These sparse matrices can be very large so in order to increase the speed of the computation even more we distribute the work of the sparse matrix-vector multiplication (SpMV) over multiple processors and let the processors work in parallel. Each processor executes a part of the matrix-vector multiplication in parallel, which results in parallel sparse matrix-vector multiplication. In order to execute the parallel sparse matrix-vector multiplication we distribute the $nz(A)$ nonzeros of the sparse matrix A over the p processors, such that every processor handles approximately $nz(A)/p$ nonzeros. The elements of both the input vector x and output vector b are distributed over the p processors. Consequentially, the parallel sparse matrix-vector multiplication $b = Ax$ is computed in four phases.

1. **Fan-out:** Communication of the values of the input vector.
2. **Multiplication:** Local multiplications.
3. **Fan-in:** Communication of the partial sums.
4. **Summation:** Summation of the partial sums.

An example of parallel sparse matrix vector multiplication for $p = 3$ can be seen in Figure 1.1, where the zeros of the matrix A are the white squares while the other squares represent the nonzeros. The red squares are nonzeros assigned to processor 0, the blue squares are the nonzeros assigned to processor 1 and the yellow squares are the nonzeros assigned to processor 2.

In the **fan-out** phase elements of the input vector x are communicated between the processors. This is necessary if there is an element in column j that is owned by a different processor than the owner of x_j. For instance in Figure 1.1 x_1 is owned by processor 2 and in column 1 there is also an element that is owned by processor 0, so the element x_1 needs to be 'communicated' by processor 2 to processor 0. The element x_4 needs to be communicated twice by processor 0, once to processor 1 and once to processor 2, because all three processors own nonzeros in column 4. This is the

communication corresponding to the columns of the matrix and this communication is represented by the vertical arrows in Figure 1.1.

In the **multiplication** step, the processors multiply their nonzeros with the corresponding element of the vector x and compute the partial sum per row. For example, in Figure 1.1, in the third row processor 0 calculates $a_{34}x_4$ and processor 1 calculates the partial sum $a_{32}x_2 + a_{33}x_3$.

In the **fan-in** step the partial sums of the previous step are communicated between the processors, as we need to add up all partial sums of row i in order to determine the element b_i. Therefore the partial sums of row i need to be communicated to the processor that owns element b_i. The example in Figure 1.1 illustrates this in row three. The partial sum $a_{34}x_4$ is communicated from processor 0 to processor 1. This step represents the communication volume of a row of the matrix; this communication is displayed in Figure 1.1 through the use of the horizontal arrows.

In the **summation** step the output vector is determined by adding the partial sums.

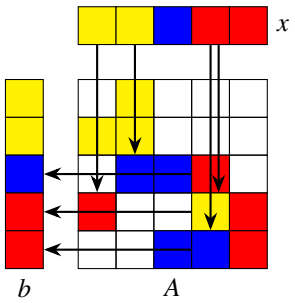


Figure 1.1: Parallel matrix-vector multiplication, with three processors $0, 1$ and 2. The arrows represent the communication between the processors. The assignment of the nonzeros and vector elements are indicated by their color: red nonzeros/ vector elements are assigned to processor 0, blue nonzeros/ vector elements are assigned to processor 1, yellow nonzeros/vector elements are assigned to processor 2 and the zeros of the matrix are white. The communication volume is equal to the number of arrows, 7.

The speed of parallel sparse matrix-vector multiplication depends heavily on an equal distribution of the workload and on the number of elements that need to be communicated in the fan-out and fan-in phase.

Therefore, we would like to minimize the communication between the processors, while maintaining a load balance. This gives rise to the problem of partitioning a sparse matrix A into p disjoint parts, while also minimizing the communication volume and maintaining the load balance; this is called the sparse matrix partitioning problem. So we partition a matrix A, such that

$$A = \bigcup_{i=0}^{p-1} A_i. \tag{1.1}$$

Furthermore we want that the nonzeros are evenly distributed over the different parts, so that every A_i satisfies the load balance constraint:

$$nz(A_i) \leq (1+\epsilon)\left\lceil \frac{nz(A)}{p} \right\rceil, \qquad \text{for } 0 \leq i < p, \tag{1.2}$$

with $\epsilon \geq 0$ being the load imbalance parameter. The load imbalance parameter is set beforehand and is often set to $\epsilon = 0.03$. The ceiling function is used in order to make sure that a valid partition is possible when $\epsilon = 0$.

If we define λ_j^c to be the number of different processors that owns an element in column j, and let the element x_j be owned by one of these processors, then the communication volume of this matrix column is $\lambda_j^c - 1$. Similarly, the communication volume of a row i during the fan-in phase is equal to $\lambda_i^r - 1$, with λ_i^r being 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 sparse matrix-vector multiplication is then equal to

$$CV(A) = \sum_{i=1}^{m}(\lambda_i^r - 1) + \sum_{j=1}^{n}(\lambda_j^c - 1). \tag{1.3}$$

Here we assume that no row or column is empty. We can see that different partitionings of the same matrix can have a huge difference in communication volume. In Figure 1.2, an optimal 3−way partitioning for the same matrix as in Figure 1.1 is shown. This optimal partitioning has communication volume equal to four, while the partitioning in Figure 1.1 has a communication volume equal to seven.

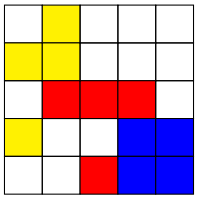


Figure 1.2: An optimal 3−way partitioning of this matrix, with corresponding optimal communication volume of 4. The load imbalance parameter was set to $\epsilon = 0.03$. This figure showcases a different partitioning when compared to the partitioning shown in Figure 1.1, but they are both partitionings of the same matrix. The assignment of the nonzeros is indicated by their color: red nonzeros are assigned to processor 0, blue nonzeros are assigned to processor 1 and yellow nonzeros are assigned to processor 2.

The sparse matrix partitioning problem is NP-Complete [12]. Therefore, only 'small' problems can be solved optimally, 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 other non-optimal methods. In this thesis we will use the results on optimal partitionings to examine the performance of recursive bipartitioning.

We will start by giving a short overview of prior research done on the sparse matrix partitioning problem, in section 1.1. In section 2, we build upon the work of both [19] and [12], who have developed an algorithm to optimally bipartition a matrix based on the branch and bound method. We will describe a similar branch and bound method which solves the sparse matrix partitioning problem for any $p \in \mathbb{N}_{\geq 2}$ optimally.

In section 3, we will describe another exact algorithm which also enables us to determine an optimal partitioning of a matrix into p parts. This algorithm is also based on the branch and bound method, but uses a different branching rule: it branches on the assignment of a nonzero to a specific processor, as opposed to branching on the assignment of a row/column to a subset of the processors. In section 4, we will describe

an ILP model which can be used to find optimal $p-$way partitioning of a sparse matrix. In section 5, we will explain the recursive bipartitioning (RB) method, which is not an exact method.

After we have described these exact methods and the RB method, we will compare their performance in section 6. In this section we will also examine the performance of the recursive bipartitioning method with regards to determining 4−way partitionings. Finally, in section 7, we will present our conclusions and give some ideas for future research.

In Appendix A the optimal communication volume of 3− and 4−way partitionings is given. Appendix B contains a table with the optimal communication volumes of 2−, 3− and 4−way partitionings, and the communication volume obtained by recursively bipartitioning a matrix into 4 parts.

1.1 Prior research

In this section we will discuss prior research done on the sparse matrix partitioning problem. The sparse matrix partitioning problem can be reformulated in several ways. One way is to formulate the matrix as a graph and partition the vertices into p sets. The standard graph model represents an $n \times n$ symmetric matrix A as the graph $G = (V, E)$ with n vertices, where a vertex i represents both row i and column i, and with an edge between vertex i and j iff a_{ij} and a_{ji} are nonzero. Thus, the matrix A is the adjacency matrix of the graph $G = (V, E)$. In order to create a partitioning, the vertices of the graph are partitioned into p subsets which are approximately equal in size, while minimizing the edge cuts. We speak of an edge cut when the pair of vertices of the edge both belong to a different subset. If vertex i belongs to subset k, with $k \in \{0, \cdots, p-1\}$ in the graph partitioning, then the nonzeros of row and column i are owned by processor k in the corresponding matrix partitioning. The edge cuts in the graph partitioning represent the communication volume of the corresponding partitioning of the matrix A in parallel SpMV.

This model has some obvious disadvantages. Firstly, the vertex i represents both row i and column i, meaning that both row i and column i will be in the same subset in a final graph partitioning. Therefore, we do not have the freedom to assign row i and column i to different processors. Secondly, this approach only works for symmetric (square) matrices.

Hendrickson and Kolda [10] proposed the bipartite graph model as an improvement over the standard graph model. In this model, an $m \times n$ matrix A is represented by a bipartite graph $G = (V_r \cup V_c,\ E)$. The graph has m vertices, that represent the rows of matrix A, and n vertices, that represent the columns of matrix A. These vertices form vertex sets V_r and V_c respectively, with an edge between a row vertex $i \in V_r$ and a column vertex $j \in V_c$ iff $a_{ij} \neq 0$. In contrast to the standard graph model, nonsymmetric and nonsquare matrices can also be modeled by this model. In order to determine an optimal partitioning for the original matrix A, the vertices of the graph G are again partitioned into p subsets, while minimizing the sum of the edge cuts. Notice that in this case row i and column i can belong to different subsets, which was not the case for the standard graph model. If vertex $v \in V_r \cup V_c$ belongs to subset k in the graph partitioning, then the nonzeros of the row or column, which is represented by vertex v, are owned by processor k in the corresponding matrix partitioning. Identical to the standard graph model, the edge cuts represent the communication volume of the corresponding matrix partitioning.

However, the most important shortcoming of both graph models is that the sum of the edge cuts in the graph partitioning is not proportional to the communication

volume. The number of edge cuts is only an approximation of the real communication volume of the corresponding matrix A [9, 4]. For instance, imagine a situation in which x_j is owned by processor 0, row/vertex 2 and row/vertex 3 are both owned by processor 1, and a_{2j}, a_{3j} are both nonzero in matrix A. Processor 0 will then only communicate the value of x_j to processor 1 once. This situation results in a single communication volume, but in the graph models a total of two edges are cut.

Çatalyürek and Aykanat [4] resolved this shortcoming by formulating the sparse matrix partitioning problem as a hypergraph partitioning problem. In [4] they presented the row-net and column-net models. In the row-net model, every column i of the $m \times n$ matrix A is represented by a vertex i and every row j is represented by a net n_j (hyperedge), which is a subset of V. This results in the hypergraph $H = (V, N)$. The weight w_i of a vertex i is equal to the number of nonzeros in column i. The net n_j will only contain vertex i iff $a_{ji} \neq 0$. Conversely, in the column-net model there is a vertex for every row and a net for every column. The vertices of the hypergraph H are partitioned into p subsets with approximately equal weight. This partitioning has communication volume $\sum_{n_j \in N}(\lambda_j - 1)$, where λ_j is the number of subsets that net n_j connects. These two hypergraph models correctly determine the communication volume, meaning that the partitioning of the matrix A that corresponds to the partitioning of the hypergraph has the same communication volume in parallel SpMV as it does in the hypergraph model.

The row-net and column-net models are 1-dimensional models. The authors Çatalyürek and Aykanat have also proposed a 2-dimensional hypergraph model: the fine-grain hypergraph model [5]. Every nonzero of an $m \times n$ matrix A is a vertex of the hypergraph $H = (V, N)$ and every row and every column will be represented by a net, so $|V| = nz(A)$ and $|N| = m + n$. The vertices of the hypergraph H are partitioned into p subsets of approximately equal size. The communication volume of the partitioning is again $\sum_{n_j \in N}(\lambda_j - 1)$. The fine-grain model also correctly determines the communication volume of a matrix A in parallel SpMV. Furthermore, the fine-grain model is more flexible, because not all nonzeros of one row (column) need to be assigned to the same processor, as is the case in the column-net (row-net) model. Therefore, this model also gets partitionings with lower communication volume than the 1-dimensional models, although it will cost more computation time and usage of memory. Furthermore, since every nonzero is represented by a vertex, meaning that there is no restriction on assigning a nonzero to a subset except for a load balance constraint, an optimal $p-$way partitioning of the vertices of the hypergraph $H = (V, N)$ corresponds to an optimal $p-$way partitioning of the matrix A. However, hypergraph partitioning is NP-Hard [14].

Little work has been done on optimally solving the sparse matrix partitioning problem or on optimal hypergraph partitioning. An optimal sparse matrix bipartitioner, MondriaanOpt, was developed by Pelt and Bisseling [19]. MondriaanOpt determines an optimal bipartitioning ($p = 2$) of a matrix A and was able to solve 355 matrices from the SuiteSparse Matrix Collection [7] within 24 hours. An improvement on MondriaanOpt is the program MatrixPartitioner (MP) by Knigge and Bisseling [12]. The program MatrixPartitioner was able to optimally bipartition 839 matrices of the SuiteSparse Matrix Collection within 24 hours. The solution of these 839 optimally solved matrices can be found at [1].

An optimal hypergraph partitioner, PHaraoh, was developed by Usta [21]. PHaraoh uses a branch and bound approach and can be used to find the optimal partitioning of a hypergraph for various hypergraph metrics, among which the connectivity-1 ($\lambda - 1$) metric, which can be used to exactly model the communication volume of SpMV.

In [6], Gray-code based enumeration and branch and bound are used to optimally solve small instances of hypergraph partitioning that arise in standard-cell placement.

[1] `https://webspace.science.uu.nl/~bisse101/Mondriaan/Opt/`

In [13], several test problems of hypergraph partitioning arising from VLSI design are solved with different methods, among which the exact method ILP and the heuristic hypergraph partitioner hMetis. Although the ILP method is optimal, hMetis finds the same optimal solution in a fraction of the time for the instances of the test problems that could be solved with ILP, but without guaranteeing its optimality. Wrighton and DeHon [24] formulate the hypergraph partitioning problem as a boolean satisfiability problem and optimally solve problems up to 150 vertices.

There are several heuristic solvers for hypergraph partitioning which can also be used to heuristically solve the sparse matrix partitioning problem. Among these are the sequential packages, hMetis [11], Mondriaan [23], PaToH [4, 5] and KaHypar [2] and the parallel packages Par$k$way [20], Zoltan [8] and the deterministic parallel partitioner BiPart [15].

The SuiteSparse Matrix Collection mentioned above is a large set of sparse matrices that arise in real applications, and new matrices are continuously added to this collection [7]. This collection is used to test algorithms on sparse matrices which are based on realistic applications, in order to determine the performance of these algorithms. We will use the matrices of this collection as problem instances in this thesis.

2 | Optimal partitioning

In this section we will describe an algorithm that optimally partitions a matrix A in p parts using a branch and bound method. As was mentioned in the introduction the problem of partitioning a matrix A in p parts in a balanced way, while minimizing the communication volume, is NP-Complete. So we can not expect to find a polynomial-time algorithm that optimally solves this problem. However, we can try to develop an algorithm that optimally solves small instances of the matrix partitioning problem, i.e. optimally partitions small matrices.

2.1 Branch and Bound method

The branch and bound method is often used to solve optimization problems. In short, the branch and bound method explores the set of feasible solutions of an optimization problem, and uses bounds on the optimal solution to prevent it from searching the whole feasible region in order to find a solution with optimal value. In our case, the feasible solutions are the set of partitionings of the matrix that meet the load balance constraint (1.2).

We start with the complete set of feasible solutions $\mathcal{S}$ and we split this set into subsets $\mathcal{S}_1, \cdots, \mathcal{S}_k$, based on a property of the solutions. This procedure is repeated by splitting the subsets $\mathcal{S}_1, \cdots, \mathcal{S}_k$ 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 branch and bound algorithm. This leads to the branch and bound tree; a rooted tree with the root representing the complete set of feasible solutions, a node in the tree representing a subset of the feasible solutions and a leaf in the tree representing a single solution to the optimization problem. Essentially, this procedure is just an enumeration of all possible feasible solutions.

However, we can use bounds to avoid searching the whole feasible region. Assume we have a minimization problem and that we have an upper bound UB on the optimal solution of this minimization problem. The upper bound UB can, for instance, be the best solution found so far. 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 skip this part of the tree. Therefore, we do not have to branch from node v. This is the "bounding" part of the branch and bound method.

2.2 Branching rule

We have several options on how to split, i.e. branch on, the set of feasible solutions of the sparse matrix partitioning problem. A first thought might be to use the branch and bound method to partition the nonzeros of the matrix directly. Thus branch on in which of the p parts a nonzero is put in.

If we use this branching rule and want to partition a matrix A with $nz(A)$ nonzeros into p parts, then for every nonzero we have p options resulting in $p^{nz(A)}$ different possible partitions for matrix A. The optimal partitioning is the partitioning of the matrix with the lowest communication volume. The problem with this way of branching is that it is difficult to define lower bounds on the partial solutions.

However, we can use the fact that every nonzero is located in exactly one column and exactly one row, and that the communication volume is only increased when for a row/column j, respectively λ_j^r / λ_j^c is increased. We can use this observation to partition the rows and columns of the matrix instead of directly partitioning the set of nonzeros. So branching on to which processor or processors a row/column is assigned. For example, if there are $p = 3$ processors $\{0, 1, 2\}$ a row/column can be either assigned to one of the 3 processors, or can be assigned to 2 of the 3 processors, or the row/column can be assigned to all 3 processors. So if $p = 3$, there are seven different states $\{0, 1, 2, 01, 02, 12, 012\}$ to assign to a row/column. State "0" means that a row/column is assigned to processor 0, meaning all nonzeros in this row/column are assigned to processor 0. State "01" means that a row/column is assigned to processor 0 and 1, so some of the nonzeros in that row are assigned to processor 0 and the rest of the nonzeros are assigned to processor 1. The same reasoning applies to the other states (i.e. "02", "012") and for states consisting of more integers if $p > 3$. In case there are p processors given by the set $\{0, \cdots, p-1\}$, there are $2^p - 1$ possible different states for every row and column. Since there are $p = \binom{p}{1}$ different ways of assigning a row/column to 1 processor and $\binom{p}{2}$ different ways of assigning a row/column to 2 processors, continuing this reasoning gives:

$$\binom{p}{1} + \binom{p}{2} + \cdots + \binom{p}{p-1} + \binom{p}{p} = 2^p - 1. \tag{2.1}$$

Therefore, every node in the branch and bound tree has $2^p - 1$ children. There are $m + n$ levels in the tree, resulting in a total number of leaves equal to $(2^p - 1)^{m+n}$. In Figure 2.1, the first two levels of the branch and bound tree for $p = 3$ are shown. The total number of possible partitionings is equal to $(2^p - 1)^{m+n}$. This number of possible partitionings is smaller than $p^{nz(A)}$, if:

$$\frac{m+n}{nz(A)} < \log_{2^p-1}(p) \approx \frac{\log_2 p}{p}. \tag{2.2}$$

In Table 2.1 the value of $\log_{2^p-1}(p)$ is given for several different values of p. Furthermore, many of these partitions are infeasible, for instance if $a_{ij} \neq 0$ and the state of row i is equal to 0 then the state of column j cannot be 1, because element a_{ij} cannot be assigned to both processor 0 and 1. Additionally, it is easier to define lower bounds if we branch on the state of a row/column.

Nevertheless, the number of possible partitionings grows exponentially in p and $m + n$. Therefore it is necessary to examine different ways to prune the decision tree, for example by eliminating symmetric assignments and by using strong lower bounds on the communication volume of the partial partitionings. Before we discuss ways to prune the branch and bound tree, we will first discuss some notation and concepts that will be used throughout this thesis.

p	$\log_{2^p-1}(p)$
3	≈ 0.56
4	≈ 0.51
8	≈ 0.38

Table 2.1: Values of $\log_{2^p-1}(p)$ for different values of p.

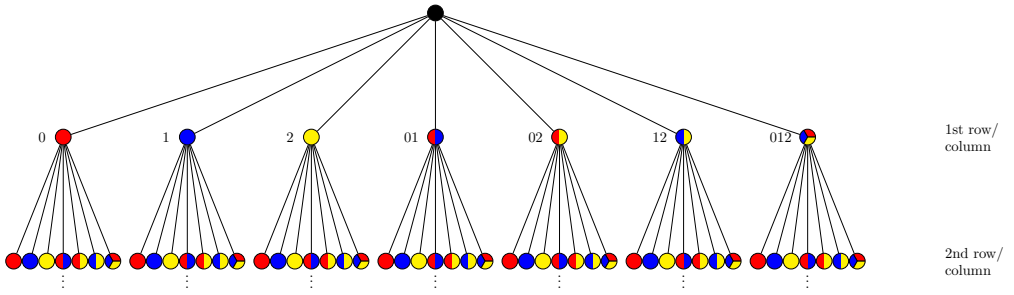


Figure 2.1: The first two levels of the branch and bound tree for $p = 3$. Red represents processor 0, blue represents processor 1 and yellow represents processor 2. A node with multiple colors represents that the row/column is assigned to multiple processors.

2.3 Notation and concepts

Before continuing with the rest of this thesis, we will discuss some notation and terms used in upcoming sections. As stated in the previous section we will assign rows to one or more processors. Assigning a row to processors $x_1 \cdots x_k$ with $x_i \neq x_j$ if $i \neq j$ and $1 \leq k \leq p$ is equivalent to assigning state $x_1 \cdots x_k$ to a row. The same holds true for columns. There are $2^p - 1$ possible different assignments for every row/column. So we are actually partitioning the rows and columns into $2^p - 1$ sets. We denote these sets by $B_{x_1 \cdots x_k}$. So if $p = 3$, we partition the rows and columns into the following 7 sets; B_0, B_1, B_2, B_{01}, B_{02}, B_{12} and B_{012} and we let $\mathcal{B} = \{B_0, B_1, B_2, B_{01}, B_{02}, B_{12}, B_{012}\}$. A full partitioning of a matrix can then be represented by

$$\mathcal{B} = \bigcup_{x_1 \cdots x_k \subseteq \mathscr{P}} B_{x_1 \cdots x_k},$$

where $\mathscr{P}$ is the set of processors $\{0, \cdots, p-1\}$. From $\mathcal{B}$ we can determine which processor a nonzero a_{ij} is assigned to. For instance, if row $r_i \in B_0$ and column $c_j \in B_{01}$, then nonzero a_{ij} is assigned to processor 0. Likewise, if $r_i \in B_{01}$ and column $c_j \in B_{012}$, then the nonzero a_{ij} can be assigned to processor 0 or to processor 1. It is possible that we are only allowed to assign a_{ij} to processor 0, and that we are not allowed to assign any more nonzeros to processor 1, because of the load balance equation. However, it can also be the case that we are free to assign nonzero a_{ij} to either processor 0 or 1.

We want to determine the communication volume of a full partitioning $\mathcal{B}$ of a matrix A. From the introduction we know that, if a row is assigned to more than one processor $\lambda_i^r > 1$, then communication is needed and the communication cost of this row is equal to $\lambda_i^r - 1$. So if a row is assigned to two processors, the communication volume of this row is equal to one. Another way of saying a row is assigned to two processors is to say that this row is “cut". In a similar sense we say that a row is “twice cut" if it is

assigned to three processors, etc. Therefore, the number of cuts in a row represents the communication volume of that row. The same is true for columns.

The communication volume of a full partitioning $\mathcal{B}$ of a matrix A if $p = 3$ can be determined by adding one communication to the total communication volume for every row/column that is "once cut" and adding two communications to the total communication volume for every row/column that is "twice cut" . The communication volume of the full partitioning $\mathcal{B}$ is equal to the total number of cuts. Therefore, the communication volume of a full partitioning $\mathcal{B}$ of a matrix A in p parts is:

$$CV(\mathcal{B}) = \sum_{x_1 \cdots x_k \subseteq \mathscr{P}} (k-1)|B_{x_1 \cdots x_k}|. \tag{2.3}$$

Note that k is equal to the number of processors that own nonzero(s) in a row/column, so the communication volume of a row/column that is assigned to k processors is equal to $k-1$.

An important part of the branch and bound algorithm is to determine lower bounds on partial partitionings of a matrix. We will denote a partial partitioning of a matrix with:

$$\hat{\mathcal{B}} = \bigcup_{x_1 \cdots x_k \subseteq \mathscr{P}} \hat{B}_{x_1 \cdots x_k},$$

where $\hat{B}_{x_1 \cdots x_k}$ is the set of rows/columns which are assigned state "$x_1 \cdots x_k$" in the partial partitioning. The maximum size of a part, i.e. the maximum number of nonzeros a processor can own will be denoted by M, where $M = \lfloor (1+\epsilon)\lceil \frac{nz(A)}{p} \rceil \rfloor$

When looking at unassigned rows in order to determine a lower bound on a partial solution, we can divide the nonzeros in this row into two groups: "assigned" and "free" (or unassigned) nonzeros. The assigned nonzeros are the nonzeros that are in a column which has been assigned a state. The "free" or unassigned nonzeros are the nonzeros that are in a column which has yet to be assigned. Likewise, we can divide the nonzeros of an unassigned column into "assigned" and "free" nonzeros.

2.4 Symmetry

The article by Mumcuyan, Usta, Kaya and Yenigün [16] describes how the branch and bound tree for $p = 2$ can be pruned by using symmetry. This concept can also be applied to the instance where the number of processors is equal to $p = 3$ or $p = 4$ or more arbitrarily when $p \in \mathbb{N}_{\geq 2}$. If there are p processors, then there are $2^p - 1$ options for every row/column, so in total there are $(2^p - 1)^{m+n}$ possible matrix partitionings, but a significant number of these possible partitionings are essentially the same.

For example, see Figure 2.2, which showcases a full partitioning of a matrix in 3 parts ($p = 3$); the nonzeros of this matrix are divided into three groups. If instead we assign the nonzeros assigned to processor 0 to processor 1 and assign the nonzeros assigned to processor 1 to processor 0, then we have essentially the same partitioning, so that the communication volume does not change. The groups of the nonzeros and the communication volume stay the same. Only the processor to which the groups are assigned changes. We see that this full partitioning has $p!$ symmetric full partitionings (including itself). For the first group of nonzeros we have 3 options, i.e. one of the three processors, for the second group we have 2 options and for the last group we have 1 option. Since the nonzeros are divided into the same groups for all these symmetric partitionings (only the processors to which the groups are assigned differ) and the communication volume is the same for each of these symmetric partitionings, we do not have

to visit all these full partitionings during the branch and bound algorithm, but only one of the $p!$ symmetric partitionings.

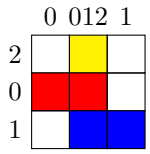


Figure 2.2: Full partitioning of a matrix for $p = 3$. The communication volume of this partitioning is equal to 2. Next to each row and column are the numbers of the processors to which a row/column is assigned. For example, '0' indicates that the row/column is assigned to processor 0. The assignment of the nonzeros is indicated by their color: red nonzeros are assigned to processor 0, blue nonzeros are assigned to processor 1 and yellow nonzeros are assigned to processor 2.

In the following we will discuss how we can prune the branch and bound tree based on symmetric partitionings. As an example, the situation that the first row/column is assigned to processor 0 and the second row/column is assigned to processor 2 is equal to the situation where the first row/column is assigned to processor 1 and the second row/column is assigned to processor 0. The main observation is that both the first and the second row/column are assigned to only one processor and each row/column is assigned to a different processor.

Similarly, there is no 'real' difference between the following two cases. The first case concerns the situation that the first row/column is assigned to two processors 0 and 1, and the second row/column is assigned to processor 2. The second case concerns the situation that the first row/column is assigned to processor 1 and 2, and the second row/column is assigned to processor 0. In both instances the first row/column is assigned to two processors and the second row/column is assigned to one processor which is different from the two previous processors.

This means that we only have to consider one of the two cases in the two examples mentioned above.

Symmetry 1st row/column:
Consequently, if we assign the first row/column to one processor, we only have to look at the case where it is assigned to processor 0, because for instance assignment to processor 1 leads to symmetric assignments. Therefore, we only have to consider p options for the first row/column that we assign. We can assign it to a single processor, we can assign it to two processors, $\cdots$, we can assign it to $p-1$ processors, or we can assign it to all p processors. This reduces the number of leaves in the branch and bound tree from $(2^p-1)^{m+n}$ to $(2^p-1)^{m+n-1} \times p$.

Symmetry 2nd row/column:
We have seen that we can reduce the number of leaves by looking only at a subset of all the possible states for the assignment of the first row/column, based on symmetry. We can take this further and look at the states we can eliminate for the rows/columns that are assigned at a lower layer in the branch and bound tree. We will now look at the states that can be eliminated from the set of possible states and which state we still have to consider when assigning the second row/column a state. When eliminating

states for the second row/column we need to take into account the assignment of the first row/column.

For instance, assume $p = 3$ and assume we have assigned the first row/column to one processor, w.l.o.g. we can assume that it is assigned to processor 0. We then only need to consider the following states for the second row/column; “0", “1", “01", “12" and “012". There is a real difference between assigning the second row/column to one processor, two processors, or three processors, but if we assign the second row/column to one processor there is no difference between assigning the second row/column state “1" or state “2". In both cases the second row/column is assigned to one processor which is different from the processor the first row/column was assigned to. However, assigning the second row/column to processor 0 is different from assigning it to processor 1 or 2, because the first row/column is also assigned to processor 0. Therefore, we have to consider both state “0" and state “1” for the second row/column. We will define a “rule" to determine the states we have to consider for the second row/column, based on the assignment of the first row/column.

Assume that the first row/column is assigned state σ and that state σ contains k processors. Without loss of generality we assume that the first k processors are used, which means that the other $p - k$ processors are not yet used. The states that need to be considered for the second row/column can be determined with the following rule:

Rule 1 *Let α be the total number of processors in the new state and let β be the number of processors of the new state that were already used in the first row/column, then $\forall\ \alpha$ with $1 \leq \alpha \leq p$ and subsequently $\forall \beta,\ 0 \leq \beta \leq \alpha,\quad if\ \ \beta \leq k\ \wedge\ \alpha - \beta \leq p - k$, create the state for the second row/column by picking the β lowest numbered processors from the k used processors and the $\alpha - \beta$ lowest numbered processors from the unused processors.*

Therefore, the number of different states we have to consider for the second row/-column is:

$$\sum_{\alpha=1}^{p} \sum_{\beta=0}^{\alpha} \mathbb{1}[\beta \leq k\ \wedge\ \alpha - \beta \leq p - k]. \tag{2.4}$$

The summation in equation (2.4) can be simplified. If we look at the inner summation we see that the lower bound of the summation together with the second part of the if statement give:

$$\left.\begin{array}{l} \beta \geq 0 \\ \beta \geq \alpha + k - p \end{array}\right\} \implies \beta \geq \max(0, \alpha + k - p). \tag{2.5}$$

The upper bound on the summation and the left part of the if statement give:

$$\left.\begin{array}{l} \beta \leq \alpha \\ \beta \leq k \end{array}\right\} \implies \beta \leq \min(\alpha, k). \tag{2.6}$$

Therefore, we can rewrite the inner summation, given a fixed α, as:

$$\begin{aligned} &\sum_{\beta=0}^{\alpha} \mathbb{1}[\beta \leq k\ \wedge\ \alpha - \beta \leq p - k] \\ &= \min(\alpha, k) - \max(0, \alpha + k - p) + 1. \end{aligned} \tag{2.7}$$

For the outer summation we can make a distinction between four cases. The four cases and the value of the inner sum for these cases, i.e. the value of equation (2.7), are given

by:

$$\left.\begin{array}{l}\alpha \leq k \\ \alpha \leq p-k\end{array}\right\} \quad I: \ \alpha+1, \qquad \left.\begin{array}{l}\alpha \leq k \\ \alpha > p-k\end{array}\right\} II: \ p-k+1,$$
$$\left.\begin{array}{l}\alpha > k \\ \alpha \leq p-k\end{array}\right\} \quad III: \ k+1, \qquad \left.\begin{array}{l}\alpha > k \\ \alpha > p-k\end{array}\right\} IV: \ p-\alpha+1. \tag{2.8}$$

Now we can determine the value of the summation in (2.4) by splitting it into 4 terms. We assume that $k \leq p-k$ (The case $k \geq p-k$ gives the same end result):

$$I: \sum_{\alpha=1}^{k}(\alpha+1) = \frac{k(k+1)}{2} + k, \tag{2.9}$$

$$II: 0, \tag{2.10}$$

$$III: \sum_{\alpha=k+1}^{p-k}(k+1) = (p-k-k)(k+1) = (p-2k)(k+1), \tag{2.11}$$

$$IV: \sum_{\alpha=p-k+1}^{p}(p-\alpha+1) = pk - \frac{k(2p-k+1)}{2} + k = \frac{k(k+1)}{2}. \tag{2.12}$$

The result of case II follows from the assumption $k \leq p-k$ and the fact that $\alpha \leq k$ and $\alpha > p-k$ in case II. This gives $\alpha \leq k \leq p-k$, and at the same time $\alpha > p-k$; this is impossible. The sum of the four equations in (2.9) - (2.12) gives us the value of (2.4). As a result, we see that:

$$\begin{aligned}&\sum_{\alpha=1}^{p}\sum_{\beta=0}^{\alpha}\mathbb{1}[\beta \leq k \ \wedge \ \alpha-\beta \leq p-k] \\ &= (k+1)\left(\frac{k}{2}+p-2k+\frac{k}{2}\right)+k \\ &= (k+1)p-k^2.\end{aligned} \tag{2.13}$$

From the final formula in equation (2.13) we can conclude the following: For even p, the number of states we have to consider for the second row/column is highest if $k = p/2$, where k is the number of processors used in the first row/column. For odd p, we have to consider the most states for the second row/column if $k = \lfloor p/2 \rfloor$ or $k = \lceil p/2 \rceil$.

Symmetry lower layers B & B tree:
We have examined how we can eliminate symmetric assignments by looking at the assignments of the first and second row/column, but there can also be assignments that are symmetric to one another in lower layers of the branch and bound tree. We will illustrate this with an example.

Assume that $p = 4$, and that the first row/column is assigned state "01" and the second row/column is assigned state "0", then for the third row/column we only need to consider eleven of the possible fifteen states. These eleven states are shown in Table 2.2. Some rows in Table 2.2 contain both a state outside brackets and a state inside brackets. If we would consider both states in these rows for the third row/column in our partial partitioning, we would get symmetric partial partitionings. Therefore, we only have to consider the eleven states outside the brackets.

$p = 4$	
0	23
1	012 [= “013”]
2 [“3”]	023
01	123
02 [= “03”]	0123
12 [= “13”]	

Table 2.2: The states we have to consider for the third row/column, after the assignment of state “01" and “0" to the first respectively second row/column. The states we have to consider for the third row/column are given outside brackets. Assigning a state both shown inside of a bracket and outside of a bracket to the third row/column will lead to a symmetric partial partitioning.

In the first layer of the decision tree of this example we only made a distinction between two groups of processors. The “chosen ones", i.e. processors 0 and 1, and the “unused processors", processors 2 and 3. In the second layer of the tree we made a further distinction between processors 0 and 1. So at the beginning of layer 3 we have 3 disjoint groups of processors $\{0\}, \{1\}, \{2, 3\}$. There is a difference between processors of different groups, but there is no real difference between processors in the same group. This last point leads to symmetric assignments.

We can see this as follows: at the root of the tree all processors are in a single group. When we assign a state to the first row/column we divide the group of processors into two groups: “the chosen ones" and “unused processors". After the assignment of the next row/column we check again if there is now a difference between processors within the same group. If this is the case, we split that group into groups of processors that are interchangeable. This means that at every layer in the tree the groups either stay the same or are split up into smaller groups. We can continue this process until every processor is “really" different, so until we have p groups. If there are less than p groups, there are states that lead to symmetric partitionings, meaning we can eliminate several states and only have to consider a subset of the possible states for the next row/column. However, if there are p groups there is no symmetry between assignments for the next row/column.

Assume that, after we just assigned the l^{th} row/column, the set of processors $\mathscr{P}$ is divided into n disjoint nonempty subsets, with $\cup_{i=1}^{n} \mathcal{P}_i = \mathscr{P}$, so we have an $n-$way partitioning of $\mathscr{P}$ with n fixed, $1 \leq n \leq p$ and $\mathcal{P}_i$ is the i^{th} subset of processors. Then for the $l + 1^{th}$ row/column we have to consider the following states:

Rule 2 *Start with the set S_0 that consists of one state s, s being the state that contains 0 processors. Then determine the set S_1 by using the recursive function $S_{i+1} = F(S_i, \mathcal{P}_{i+1})$ (see below for its definition). Repeat this procedure n times, to get the set S_n. Remove the state that contains 0 processors from the set S_n; this gives the final set of states we have to consider for the $l + 1^{th}$ row/column.*

$S_{i+1} = F(S_i, \mathcal{P}_{i+1})$:
For every $s \in S_i$ and $\forall \gamma$, $0 \leq \gamma \leq |\mathcal{P}_{i+1}|$, create the state s_γ by picking the γ lowest numbered processors from the subset $\mathcal{P}_{i+1}$, and add them to the processors that are already in state s. This procedure gives the set of states S_{i+1}.

We will illustrate the use of this rule with an example. Assume we have assigned the first row/column state “01" and $\mathscr{P} = \{0, 1, 2, 3\}$, so $n = 2$, $\mathcal{P}_1 = \{0, 1\}$ and

$\mathcal{P}_2 = \{2, 3\}$. We will use Rule 2 to determine the states which we have to consider for the second row/column. We start with $S_0 = \{\text{“ ”}\}$, where “ " symbolizes the state containing no processors. Using the recursive function $F(S_i, \mathcal{P}_{i+1})$ gives $S_1 = F(S_0, \mathcal{P}_1) = \{\text{“ ”}, \text{“0”}, \text{“01”}\}$. We use the recursive function again to determine S_2, this gives $S_2 = \{\text{“ ‘”}, \text{“2”}, \text{“23”}, \text{“0”}, \text{‘02”}, \text{“023”}, \text{“‘01”}, \text{“012”}, \text{“0123”}\}$. Now we only have to remove the state containing no processors from S_2. Therefore, the set of states that we have to consider for the second row/column in this example is:
{“2”, “23”, “0”, ‘02”, “023”, “‘01”,
“012”, “0123”}.

Furthermore, given the n processor sets $\mathcal{P}_i$, the number of different states that need to be considered for the $l + 1^{th}$ row/column are:

$$\begin{aligned} &\prod_{\forall \mathcal{P}_i} (\sum_{\gamma=0}^{|\mathcal{P}_i|} 1) - 1 \\ =&\prod_{\forall \mathcal{P}_i} (|\mathcal{P}_i| + 1) - 1. \end{aligned} \tag{2.14}$$

The minus 1 is because we also consider the situation that we pick zero processors from every processor set, resulting in a state that contains no processors, so this state needs to be subtracted from the set.

We can confirm formula (2.13) for the second row/column. If we look at the states for the second row/column, then $n = 2$ so we have two subsets. One subset $\mathcal{P}_1$ with used processors and the other $\mathcal{P}_2$ with unused processors. Assume as we did before that $|\mathcal{P}_1| = k$ and $|\mathcal{P}_2| = p - k$, using equation (2.14) gives $(k + 1) \cdot (p - k + 1) - 1$ and this is equal to (2.13).

2.5 Lower bounds

An essential part of the branch and bound algorithm are the lower bounds on the communication volume of the partial solutions to prune the branch and bound tree. If we have a partial partitioning $\hat{B}$ of a matrix A, what can we say about the communication volume of a solution that is an extension of this partial solution $\hat{\mathcal{B}}$? We will use the ideas on lower bounds in the case that $p = 2$ from [12, 19] and extend these lower bounds to the case where $p = 3, 4$ or $p \in \mathbb{N}_{\geq 2}$.

The most trivial bound is to use the assignments of the row/columns that are already assigned to one or more processors. This gives the first lower bound L_1. Assume that $p = 3$, then a partial solution of a matrix A can be represented by $\hat{\mathcal{B}} = \{\hat{B}_0, \hat{B}_1, \hat{B}_2, \hat{B}_{01}, \hat{B}_{02}, \hat{B}_{12}, \hat{B}_{012}\}$. We know that all rows/columns that are assigned to 2 processors are already explicitly cut; these are the rows/columns in the sets $\hat{B}_{01}, \hat{B}_{02}, \hat{B}_{12}$. The rows/columns in set $\hat{B}_{012}$ that are assigned to 3 processors have 2 explicit cuts. We can count the number of rows/columns in these sets and the number of cuts associated with the set 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 in the case of $p = 3$, L_1 is equal to:

$$L_1(\hat{\mathcal{B}}) = |\hat{B}_{01}| + |\hat{B}_{02}| + |\hat{B}_{12}| + 2|\hat{B}_{012}|. \tag{2.15}$$

This lower bound L_1 can easily be extended to the case where $p \in \mathbb{N}_{\geq 2}$:

$$L_1(\hat{\mathcal{B}}) = \sum_{x_1 \cdots x_k \subseteq \mathscr{P}} (k - 1)|\hat{B}_{x_1 \cdots x_k}|. \tag{2.16}$$

Other than explicit cuts, a partial solution also renders implicit cuts. The second lower bound L_2 counts these implicit cuts of the partial solution. If we have a row that is still unassigned in this partial solution which intersects with a column c_i and c_j, where $c_i \in \hat{B}_0$ and $c_j \in \hat{B}_1$, then this row contains a nonzero that is owned by processor 0 and another nonzero that is owned by processor 1. So every state that can be assigned to this row in the future must at least contain both processor 0 and processor 1. This implies that this row will have at least 1 cut in every future assignment. Likewise, if the row would also have a nonzero in column $c_k \in \hat{B}_2$, the future state of the row will be "012", so the row will have 2 cuts. Another way in which these implicit cuts can arise is if a row intersects with column c_i and c_j, with $c_i \in \hat{B}_{01}$ and $c_j \in \hat{B}_2$. That row will also have (at least) 1 cut in the future. Because the only valid assignments for this row must contain one of the two processors 0,1 and contain processor 2. In the same way the implicit cuts can be counted for the unassigned columns.

This idea can be extended to the case where $p \in \mathbb{N}_{\geq 2}$, by determining for every unassigned row/column if the states of the already assigned nonzeros in that row/column lead to unavoidable cut(s) in a future assignment of that row/column; these give the implicit cuts and the lower bound L_2. In the following we will discuss lower bounds that use the partial assignment of an unassigned row/column.

Possible Partial statuses

Another way to determine a lower bound on the communication volume of a partial solution $\hat{\mathcal{B}}$ of a matrix A, is by looking at the partially assigned rows and 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 subset of the processors, but not to all of the p processors, and vice-versa for partially assigned columns. So in the case $p = 3$, a partially assigned row is an unassigned row that has at least one nonzero in a column that has already been assigned to one or two processors, but not to all three processors $0, 1, 2$. The nonzeros of a column that has been assigned to all 3 processors can still be owned by any of the processors; this does not force a fixed assignment of a nonzero in the row. Therefore, it will not partially assign the row to a processor or processors.

In this thesis we will only take into account rows and columns that are partially assigned to one or two processors.

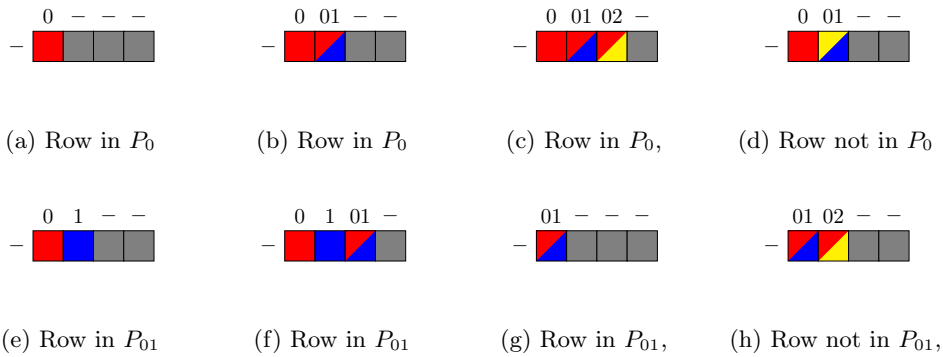
In the case that an unassigned row has at least one nonzero in a column that has been assigned to one of the processors $0, 1, 2$, the row is partially assigned to that one processor; the same holds true for unassigned columns. For example, r_2 in Figure 2.4 is partially assigned to processor 0, because it has a nonzero in a column (c_1) that has been assigned to processor 0. We will denote the set of rows and columns that are partially assigned to processor 0 with P_0; in the same manner we define P_1 and P_2. Furthermore, we define P_0^r to be the subset consisting of the rows in P_0 and P_0^c to be the set of columns in P_0. In the same way we define P_1^r, P_1^c etc..

A row/column can also be partially assigned to two processors. A row is partially assigned to two processors if it is unassigned and has at least one nonzero in a column that has been assigned to the processor x and a nonzero in a column that has been assigned to a different processor y, so $x, y \in \{0, 1, 2\}$ with $x \neq y$; vice versa for partially assigned columns. For example, row 5 in Figure 2.4 is partially assigned to the two processors 0 and 1. We will denote the set of rows and columns partially assigned to the two processors 0 and 1 with P_{01}; in the same manner we define P_{02}, P_{12}. Additionally, we will say that an unassigned row that has a nonzero in a column that has been assigned state "01" is partially assigned to processor 0 and 1, if the other assigned nonzeros in the row are in columns which have been assigned a state that contains both "0" and "1". If, for instance, it has a nonzero in a column assigned to processor 0, the row would be

in P_0, see Figure 2.3b. Hence in the example of Figure 2.4, $P_0 = \{r_2, r_4\}$, $P_1 = \{r_3\}$, $P_{01} = \{r_5\}$ and $P_{02} = \{c_2, c_4\}$, $P_{12} = P_2 = \emptyset$.

Note that an unassigned row with a nonzero in a column that has been assigned to processor 0 is not in the set of P_0 if another nonzero in this row is in a column with an assignment that "conflicts" with this partial assignment, see for an example of such a conflict Figure 2.3d.

Therefore, more generally, an unassigned row is in P_x with $x \in \mathscr{P}$, if it has a nonzero that has been assigned to processor x and no nonzeros are in columns that have been assigned a state that does not contain processor x. In the same way, an unassigned row is in P_{xy} with $x, y \in \mathscr{P}$ and $x \neq y$ if it has a nonzero assigned to processor x and a nonzero assigned to processor y, and the rest of the nonzeros that are assigned are in columns that have already been assigned a state that contains either x or y, or both of them. Alternatively, an unassigned row is in P_{xy}, with $x, y \in \mathscr{P}$ and $x \neq y$ if it has a nonzero in a column assigned state "xy", and the other assigned nonzeros in the row are in columns which have been assigned a state that contains both "x" and "y". The same is true for unassigned columns. For examples of partially assigned rows that are in the set P_0 or P_{01} in the case of $p = 3$, see Figure 2.3.



(a) Row in P_0 (b) Row in P_0 (c) Row in P_0, (d) Row not in P_0

(e) Row in P_{01} (f) Row in P_{01} (g) Row in P_{01}, (h) Row not in P_{01},

Figure 2.3: Examples of unassigned rows that are partially assigned to processor 0 ((a)-(c)). An example of an unassigned row that is not in the set P_0 (d). Ways in which an unassigned row can be in the set P_{01} ((e)-(g)). An example of an unassigned row that is not in the set P_{01} (h). 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. The assignment of the nonzeros is indicated by their color: red nonzeros are assigned to processor 0, blue nonzeros are assigned to processor 1, yellow nonzeros are assigned to processor 2 and grey nonzeros are unassigned. If a nonzero has multiple colors, then it is assigned to the processors corresponding to those colors.

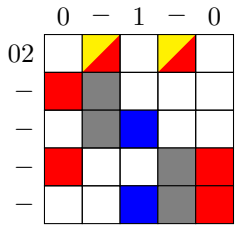


Figure 2.4: A partial partitioning of a matrix. Next to each row and column are the numbers of the processors to which a row/column is assigned. For example, '0' indicates that the row/column is assigned to processor 0, while '-' indicates that the row/column is unassigned. The assignment of the nonzeros is indicated by their color: red nonzeros are assigned to processor 0, blue nonzeros are assigned to processor 1, yellow nonzeros are assigned to processor 2 and grey nonzeros are unassigned (the zeros in the matrix are white). If a nonzero has multiple colors, then it is assigned to the processors corresponding to those colors.

2.6 Local packing bound (L_3)

The third lower bound, L_3 makes use of the partially assigned rows/columns and the load balance equation (1.2). Again, assume that $p = 3$ and assume that we have a partial assignment $\hat{\mathcal{B}} = \{\hat{B}_0, \hat{B}_1, \hat{B}_2, \hat{B}_{01}, \hat{B}_{02}, \hat{B}_{12}, \hat{B}_{012}\}$ of a matrix A. If a row is partially assigned to processor 0, so $r \in P_0$, then to avoid cutting the row we need to assign all free nonzeros (unassigned nonzeros) in this row also to processor 0, but this may not always be possible. The maximum size of any part A_i, and thus of part A_0, is M, determined by equation (1.2). So we can only avoid cutting the row if we are still allowed to assign all free nonzeros in the row to part 0. Assume we already have assigned n_0 nonzeros to processor 0, then we can still assign $M - n_0$ (unassigned) nonzeros to part 0. In order to determine a lower bound we can add up all free nonzeros of the rows in P_0^r, giving s_0. If $s_0 > M - n_0$ then not all the free nonzeros in rows $r \in P_0^r$ can be assigned to processor 0. So at least some of the free nonzeros from one row in P_0^r need to be assigned to processor 1 or 2, so at least one row $\in P_0^r$ needs to be cut. To make sure we find a lower bound, i.e. cutting the least number of rows, we start by cutting the row with the largest number of unassigned nonzeros, and we deduct this number of free nonzeros from s_0. We repeat this process by cutting the row with the second largest number of unassigned nonzeros, and subtract these nonzeros from s_0. We repeat this procedure until $s_0 \leq M - n_0$. Because we start with the row with the highest number of free nonzeros we guarantee that we cut as few rows as possible and so find a lower bound on the communication volume induced by P_0^r and the load balance equation. We can do the same for the columns and for processors 1 and 2. The L_3 bound is the sum of these 6 bounds.

In the general case $p \in \mathbb{N}_{\geq 2}$, the procedure to determine L_3 is similar: $\forall i \in \mathscr{P}$ determine the least number of rows in P_i^r that need to be cut, to make sure that $s_i \leq M - n_i$ and do the same for the columns in P_i^c. The sum of all these necessary cuts gives the lower bound L_3.

We will now give an example on how to use the three lower bounds L_1, L_2 and L_3 in order to determine the total lower bound on the communication volume of a partial partitioning. In Figure 2.4 an example of a partial partitioning is shown for $p = 3$ and

$\epsilon = 0.03$. For this partial partitioning $\hat{\mathcal{B}}$; $\hat{B}_0 = \{c_1, c_5\}$, $\hat{B}_1 = \{c_3\}$, $\hat{B}_{02} = \{r_1\}$ and $\hat{B}_2 = \hat{B}_{01} = \hat{B}_{12} = \hat{B}_{012} = \emptyset$. The partial partitioning $\hat{\mathcal{B}}$ has one explicit cut in row r_1, so $L_1(\hat{\mathcal{B}}) = 1$, and one implicit cut in row r_5, so $L_2(\hat{\mathcal{B}}) = 1$. Furthermore, this partial partitioning has the following properties: $M = 4$, $n_0 = 4$, $n_1 = 2$, $P_0 = \{r_2, r_4\}$, $P_1 = \{r_3\}$, $P_{01} = \{r_5\}$, $P_{02} = \{c_2, c_4\}$ and $P_{12} = P_2 = \emptyset$. Now we will determine the packing bound L_3 of $\hat{\mathcal{B}}$. There are two rows partially assigned to processor 0, and $s_0 = 2$, so since $M - n_0 = 0$ we have to cut both rows r_2 and r_4. There is one row partially assigned to processor 1, and $s_1 = 1$, in this case $M - n_1 > s_1$, so we do not have to cut row r_3. Therefore, $L_3(\hat{\mathcal{B}}) = 2$. The total lower bound on the communication volume of the partial partitioning $\hat{\mathcal{B}}$ is equal to $LB(\hat{\mathcal{B}}) = L_1(\hat{\mathcal{B}}) + L_2(\hat{\mathcal{B}}) + L_3(\hat{\mathcal{B}}) = 4$

2.7 Local matching bound (L_4)

In order to define the matching bound L_4 on the communication volume of the partial partitioning of a matrix A, we will study the nonzeros in the matrix where the partially assigned rows and columns conflict (i.e. the row and column of the nonzero are partially assigned to different processors.), similar to the matching bound L_4 in [19]. Such a conflict occurs for example at entry $(2, 2)$ in the partial partitioning in Figure 2.5, therefore the second column or the second row must be cut, resulting in one extra communication volume. Note that this bound uses the same rows and columns as the L_3 bound, so we cannot use L_3 and L_4 both at the same time to determine the lower bound on the communication volume, but we can take the maximum of these 2 bounds, $\max(L_3, L_4)$.

We will first explain how to compute the matching bound L_4, in the case that $p = 3$. To get the lower bound L_4 on the communication volume caused by these nonzeros, we define conflict submatrices as proposed in [19]. There are several different options for the conflict submatrices in the case $p = 3$. One option is to divide the three processors in two disjoint sets $\{x, y\}$ and $\{z\}$, so $x, y, z \in \{0, 1, 2\}$ and $\{x, y\} \cup \{z\} = \{0, 1, 2\}$ and define two conflict submatrices. The rows of one of the submatrices consists of the rows that are partially assigned to processor x and the rows that are partially assigned to processor y, whereas the columns of this submatrix are the columns that are partially assigned to processor z. The second conflict submatrix consists of the rows of P_z and the columns of P_x and P_y. So for the example in Figure 2.5 if we let $\{x, y\} = \{0, 1\}$ and $\{z\} = \{2\}$, we get one conflict submatrix consisting of row 2 up to and including row 5 and column 2 of the original matrix, see Figure 2.6a. Then the lower bound is found by defining a bipartite graph for the submatrix and finding a maximum matching for this bipartite graph as in [19]. The rows $i \in P_x \cup P_y$ form vertex set $V_{x \cup y}$ and the columns $j \in P_z$ form vertex set V_z, with $(i, j) \in E$ if a_{ij} is a (free) nonzero. In Figure 2.6b the bipartite graph corresponding to the conflict submatrix of our example is shown. Finding the lower bound L_4 on the communication volume corresponds with first finding a maximum matching $\mathcal{M}$ for this bipartite graph [19]. Additionally, a maximum matching $\mathcal{M}'$ can be found for the second conflict submatrix; combining these two gives the lower bound $L_4 = |\mathcal{M}| + |\mathcal{M}'|$.

We can add up the matchings $\mathcal{M}, \mathcal{M}'$, because the nonzeros corresponding to the edges in the matching $\mathcal{M}$ are different from the nonzeros corresponding to the edges in the matching $\mathcal{M}'$ and the same holds for the end points of the edges. An edge in the matching $\mathcal{M}$ represents a conflicting nonzero a_{ij}, where row $i \in P_x \cup P_y$ and column $j \in P_z$. An edge in the matching $\mathcal{M}'$ represents a conflicting nonzero a_{kl}, where row $k \in P_z$ and column $l \in P_x \cup P_y$. So we look at different rows and columns in both matchings and therefore look at different conflicts of the partial assignments. Solving one of these conflicts in the matching $\mathcal{M}$ by cutting a row or column will not solve

a conflict in the matching $\mathcal{M}'$. Therefore we can add the maximum matchings of the different graphs and so the matching bound L_4 is the sum of the two matchings $\mathcal{M}$ and $\mathcal{M}'$.

The maximum matching in the graph of Figure 2.6b consists of one edge, so $|\mathcal{M}| = 1$. Therefore for the partially assigned matrix in Figure 2.5, $L_4 = 1$.

For these definitions of conflict submatrices we can also combine the two bipartite graphs into one (disconnected) bipartite supergraph and just determine the maximum matching $\mathcal{M}$" for this graph. We define a vertex set $V_{\{x,y\}}$ consisting of all rows and columns that are partially assigned to processor x or processor y, and we define a second vertex set $V_{\{z\}}$ consisting of all columns and rows that are partially assigned to processor z. Together with the edge set E that corresponds to the conflicting nonzeros, this gives the bipartite supergraph $G = (V_{\{x,y\}} \cup V_{\{z\}}, E)$.

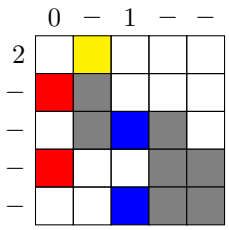
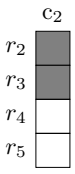
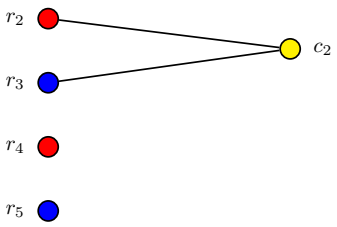


Figure 2.5: A partial partitioning of a matrix, with $r_2,\ r_4 \in P_0$, $r_3, r_5 \in P_1$ and $c_2 \in P_2$. Next to each row and column are the numbers of the processors to which a row/column is assigned. For example, '0' indicates that the row/column is assigned to processor 0, while '-' indicates that the row/column is unassigned. The assignment of the nonzeros is indicated by their color: red nonzeros are assigned to processor 0, blue nonzeros are assigned to processor 1, yellow nonzeros are assigned to processor 2 and grey nonzeros are unassigned. (The zeros in the matrix are white.)



(a) The conflict submatrix. The nonzeros are gray and the zeros are white.



(b) The bipartite graph. The vertices corresponding to rows $i \in P_0$ are red, the vertices corresponding to rows $i \in P_1$ are blue, and the vertices corresponding to the columns $j \in P_2$ are yellow.

Figure 2.6: The conflict submatrix (a) and corresponding bipartite graph (b) which both correspond to the partial partitioning in Figure 2.5. The edges in the bipartite graph represent the nonzeros in the conflict submatrix.

General $p \in \mathbb{N}_{\geq 2}$

We can generalize this idea for the matching bound L_4 for any arbitrary number of p processors. We divide the set of the p processors $\mathscr{P}$ into two disjoint subsets Θ and Φ, such that $\Theta \cup \Phi = \mathscr{P}$. We let $P_\Theta = \cup_{\theta \in \Theta} P_\theta$ and $P_\Phi = \cup_{\phi \in \Phi} P_\phi$ and define a bipartite graph with vertex set V_Θ and vertex set V_Φ with edges E corresponding to the nonzeros, $(p, q) \in E$ if $p \in V_\Theta$ and $q \in V_\Phi$ and;

- $p = r_i \in P_\Theta$ and $q = c_j \in P_\Phi$ with a_{ij} nonzero or,
- $q = r_i \in P_\Phi$ and $p = c_j \in P_\Theta$ with a_{ij} a nonzero.

Then a maximum matching $\mathcal{M}$ for the graph $G = (V_\Theta \cup V_\Phi, E)$ gives the matching bound L_4 for the partial partitioning of matrix A, with $L_4 = |\mathcal{M}|$.

However, a drawback of determining the matching bound L_4 in the aforementioned way is that we underestimate the communication volume. We miss the cases where a row i of P_x and a row j of P_y have a nonzero in the same column k of P_z, and vice versa. This situation increases the communication volume by two, but when we look at a maximum matching we will only count one communication volume. For the partial partitioning in the example in Figure 2.5, either two of row r_2, r_3 and column c_2 must be cut, or column c_2 must be "twice" cut. The two cuts result in two extra communication volume. However, the lower bound L_4 for this partial partitioning, given by the maximum matching in the graph of Figure 2.6b, is equal to one.

This issue can easily be resolved by splitting the vertices of the bipartite graph. If there are p processors, then if $v \in P_x$ with $x \in \mathscr{P}$ we split the vertex v in $p-1$ vertices v^y, with $y \in \mathscr{P} - \{x\}$. So we split every vertex that is partially assigned to one processor into $p-1$ vertices. There is an edge between a vertex v^θ and w^ϕ iff $v \in P_\theta$ and $w \in P_\phi$ and there is a nonzero in the matrix in the row (or column) corresponding to vertex v^θ and the column (or row) corresponding to the vertex w^ϕ.

An example of splitting the vertices of the bipartite graph 2.6b is given in Figure 2.7. We see that the maximum matching $\mathcal{M}$ of the bipartite graph in Figure 2.7 has a cardinality equal to two, instead of being equal to one. Therefore, we see that, if we determine the matching bound L_4 on the bipartite graph with the split vertices, we do take into account the situation in which a row $r_i \in P_x$ and a row $r_j \in P_y$ have a nonzero in the same column $c_k \in P_z$.

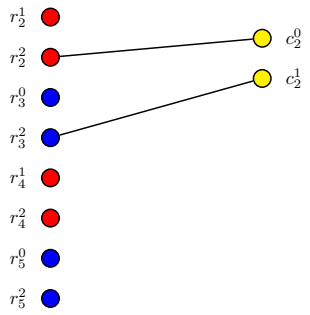


Figure 2.7: Bipartite graph using split vertices, corresponding to the bipartite graph in Figure 2.6b. The cardinality of the maximum matching of this bipartite graph is equal to two. The vertices corresponding to rows/columns in P_0 are red, while the vertices corresponding to rows/columns in P_1 are blue and the vertices corresponding to rows/columns in P_2 are yellow.

2.7.1 Combined L_3 and L_4 bound: L_5

As explained in the previous section, the lower bounds L_3 and L_4 make use of the same partially assigned rows and columns, so one cannot add up these bounds. Therefore, we have to take the maximum of the two bounds, $\max(L_3, L_4)$ and use this to determine the overall lower bound of the partial solution. However, we would prefer it if we could combine these bounds. This is possible if we first compute the L_4 bound and then remove all the partially assigned rows and columns that are matched in this bound from the sets of partially assigned rows and columns P_i, with $i \in \mathscr{P}$. Second, compute the L_3 bound for the remaining partially assigned rows and columns. In this way, the L_3 bound will not take into account any rows or columns that were already used by the L_4 bound.

For example, Figure 2.8 displays a partial partitioning $\hat{\mathcal{B}}$ of a matrix for $p = 3$. The sets of partially assigned rows and columns are; $P_0 = \{r_2, r_4, r_5\}$, $P_1 = \{r_3\}$ and $P_2 = \{c_2\}$. The maximum size of any part is $M = \lfloor 1.03\lceil \frac{12}{3} \rceil \rfloor = 4$ and $n_0 = 3$, $n_1 = 1$ and $n_2 = 1$. So for the example in Figure 2.8 $L_1 = L_2 = 0$, $L_3(\hat{\mathcal{B}}) = 2$ and $L_4(\hat{\mathcal{B}}) = 2$, so $\max(L_3, L_4) = 2$. Therefore, the lower bound on the partial partitioning is $LB(\hat{\mathcal{B}})) = 2$.

However, if we first compute the L_4 bound and then remove the rows/columns that have been matched in the L_4 bound, i.e. remove r_2, r_3 and c_2 from the set of partially assigned row/columns, we obtain $P_0 = \{r_4, r_5\}$ and $P_1 = P_2 = \emptyset$. Subsequently, we compute the L_3 bound for the remaining partially assigned rows/columns, and we see that $M - n_0 = 4 - 3 = 1$ and $s_0 = 4$, so both rows r_4 and r_5 need to be cut giving $L_3 = 2$. Now we can add up both bounds giving the combined bound of $L_3 + L_4 = 2 + 2 = 4$. Using the combined bound results in a lower bound equal to 4 instead of 2 for the partial partitioning in Figure 2.8. We will denote the combination of the local packing L_3 and matching bound L_4, as the local lower bound L_5.

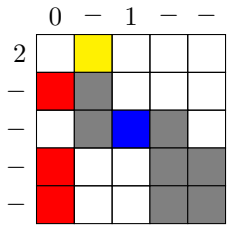
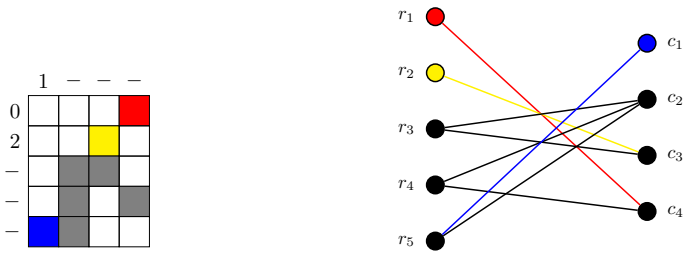


Figure 2.8: Partial partitioning of a matrix. Next to each row and column are the numbers of the processors to which a row/column is assigned. For example, '0' indicates that the row/column is assigned to processor 0, while '-' indicates that the row/column is unassigned. The assignment of the nonzeros is indicated by their color: red nonzeros are assigned to processor 0, blue nonzeros are assigned to processor 1, yellow nonzeros are assigned to processor 2 and grey nonzeros are unassigned (the zeros in the matrix are white).

2.8 Global matching bound (GL_4)

In the previous sections we discussed lower bounds that look at the direct neighbourhood of a partially assigned row/column. In this section we will discuss how to extend the local packing bound L_4 to make more use of the connectedness of rows and columns through nonzeros. In [12] it was discussed how to do this in the case $p = 2$. We will use a very similar approach for the case $p = 3, 4$, and more generally $p \in \mathbb{N}_{\geq 2}$.

We will start by illustrating this through an example for the case $p = 3$. In Figure 2.9, a partial partitioning and the corresponding graph representation of a matrix are given, with $p = 3$. For this partial partitioning $\hat{\mathcal{B}}$, $\hat{B}_0 = \{r_1\}, \hat{B}_1 = \{c_1\}, \hat{B}_2 = \{r_2\}$ and the rest of the sets $\hat{B}_{x_1 \cdots x_k}$ with $x_1, \cdots, x_k \in \mathscr{P}$ and $x_i \neq x_j$ if $i \neq j$ are empty. Furthermore, the sets of partially assigned rows and columns are; $P_0 = \{c_4\}$, $P_1 = \{r_5\}$, $P_2 = \{c_3\}$. Given the 4 lower bounds of the previous sections, the lower bound on the communication volume of this partial partitioning $\hat{\mathcal{B}}$ is 0, so $LB(\hat{\mathcal{B}}) = 0$. However, we can see (in the following) that this partial partitioning will always give a communication volume with a value equal to at least 2, so every full solution which is an extension of this partial partitioning will have a communication volume of at least 2.



(a) (b)

Figure 2.9: An example of a partial partitioning of a matrix (a), and its corresponding graph representation (b). The paths r_5, c_2, r_3, c_3 and r_5, c_2, r_4, c_4 lead to a conflict between the partial assignment of r_5 and c_3 respectively r_5 and c_4. The partial partitioning has at least a communication volume of 2. Black vertices in the graph indicate that the corresponding row/column is unassigned.

This can be seen by starting in row r_5. In order to avoid cutting this row, we need to assign a_{52} to part 1, but then to avoid cutting column c_2 we need to color the rest of the nonzeros in c_2 blue. Subsequently, in order to avoid a cut in r_3, we have to color a_{33} blue, but this means that we have to cut c_3, because nonzero a_{23} is already yellow, and to avoid cutting this column c_3 we should have colored all nonzeros in this column yellow.

Similarly, if we start in row r_5 and colored all nonzeros in r_5 and c_2 blue in order to avoid a cut in r_5 and c_2, then to avoid a cut in row r_4 we need to color the nonzero a_{44} blue, conflicting with the partial assignment of c_4 to red. We therefore see that the path r_5, c_2, r_3, c_3 starts with a row that is partially assigned to processor 1 and the last row/column of the path is partially assigned to processor 2. So we cannot assign the rows and columns on this path in such a way that no row or column is cut in a future assignment. Similarly, for path r_5, c_2, r_4, c_4 the first row/column and last row/column are both partially assigned to a different processor, and at least one row/column on this path must be cut. Therefore, we see that the partial assignments conflict in both paths. While calculating local matching bound L_4 we looked at the nonzeros in the matrix for which the partial assignment of the row and column conflict. Instead of looking at partial assignments that conflict via a nonzero, i.e. a path of one edge, we can also look at partial assignments that conflict in a path of length greater than 1. This is the key observation of the global matching bound GL_4.

However, two matching paths will not necessarily lead to an increase of 2 in the communication volume. Notice that for the example in Figure 2.9, if r_2 would be assigned to part 0, then column c_3 would be partially assigned to part 0, similar to column c_4. In this case both paths would only lead to one unavoidable cut in future assignments, because both paths start with a row/column partially assigned to part 1 and end with a row/column partially assigned to part 0 and the paths are not vertex disjoint. Both paths contain row r_5 and column c_2. Therefore, we could just cut row r_5 or column c_2 and make one of the two red/blue to “cancel" the conflicting partial assignments of r_5, c_3 and r_5, c_4. Summarizing, two paths do not guarantee a lower bound of 2. Instead, two paths that start with a row/column with the same partial state, end in a row/column with the same partial state, and are not vertex disjoint, lead only to an increase in communication volume of 1.

We will define a matching $P_i - P_j$ path as a path $v_1, \ldots, v_k$, where $v_1 \in P_i$ and

$v_k \in P_j$ are partially assigned and the internal vertices are free. A vertex is free if none of the nonzeros in the row/column corresponding to this vertex are assigned. If $p = 3$ and there is a vertex v_i that is partially assigned to one processor w.l.o.g we can assume that $v_i \in P_0$. We can find at most two matching paths for a vertex $v_i \in P_0$ that will increase the communication volume, one matching $P_0 - P_1$ path and one matching $P_0 - P_2$ path. These two paths can contain an internal vertex that is the same, so they do not have to be vertex disjoint, as is the case in the example in Figure 2.9. Notice that if we find a matching $P_0 - P_j$ path $v_i, \ldots, v_k$ with $v_k \in P_j$ $j \neq 0$, then in the future we can only find one additional matching path for vertex v_k, because v_k is already on a matching path to a vertex in P_0.

Subsequently, we try to find new matching paths for the next vertex v_j that is partially assigned to one processor. The new matching paths of vertex v_j must be internally vertex disjoint with the matching paths of v_i.

We will generalize this for the case $p \in \mathbb{N}_{\geq 2}$. Let the bipartite graph $G = (V, E)$ be the graph representation of the partial solution and the set $V' \subset V$ be the subset of vertices, i.e. row/columns, that are partially assigned to one processor. We will try to find new matching paths for $v'_i \in V'$, assuming that v'_i is partially assigned to processor ϕ. Furthermore, assume that v'_i is already the end point of η matching paths. The start points of all these η paths must have a different partial assignment. We define Φ' as the subset of processors to which v'_i is already matched via these paths, and we set $\eta' = |\Phi'|$. Notice that in this case $\eta = \eta'$. However, if we include rows/columns that are partially assigned to two processors, it is possible that $\eta < \eta'$.

Therefore, we can still try to find $\nu = (p-1) - \eta'$ matching $P_\phi - P_\psi$ paths starting in v'_i, with $\psi \in \mathscr{P} - \{\phi\} - \Phi'$. We will denote these matching paths with $v'_i, \ldots, v'_{k_1}$, $v'_i, \ldots, v'_{k_2}$, $\ldots\ldots$, $v'_i, \ldots, v'_{k_\tau}$, where $0 \leq \tau \leq \nu$. We denote the processor to which the end point v_{k_l} is partially assigned with $\psi_l \in \mathscr{P}$. These new matching paths must be internally vertex disjoint from all previously found matching paths. Furthermore the new paths must also fulfill the following requirement: $\psi_l \neq \psi_j$ if $l \neq j$. The global packing bound GL_4 is equal to the maximum set of matching paths that can be found in the graph $G = (V, E)$.

We can also take into account rows/columns that are partially assigned to two processors. Let v be a vertex representing a row that is partially assigned to two processors, $v \in P_{xy}$ with $x \neq y$ and $x, y \in \mathscr{P}$. The process of finding a matching path from such an vertex is similar, the difference being that ϕ is changed to a set of processors $\Phi = \{x, y\}$ in this case. Similarly, if v would appear as the end point v_{k_l} of a matching path, we will have a subset of processors $\Psi_l \subset \mathscr{P}$, instead of one processor ψ_l.

We have implemented a greedy way of the global matching bound GL_4. We start with the first row/column v'_1 that is partially assigned to one processor, perform a BFS starting from this vertex to find as many matching paths as possible starting in v'_1. Subsequently, we try to find as many matching paths as possible starting in vertex v'_2. After we have traversed all rows and columns partially assigned to one processor, we will continue to find matching paths from vertices that are partially assigned to two processors, as was defined in section 2.5. The sum of the matching paths gives the global matching bound GL_4. The pseudo code is given in Algorithm 1. In the pseudo code we use the following operations on a *queue*:

- *.pop_front*(): remove the first element from the queue.
- *.push_back*(): add element to the back of the queue.
- *.front*(): access the first element of the queue.

At most we will perform a BFS, which costs $O(V + E)$ time, for all vertices in V. So the computation of the GL_4 bound costs $O(V(V + E))$, i.e. $O((m+n)(m+n+nz(A)))$.

This is a very pessimistic upper bound on the computation time. In practice, the computation time will probably be far less than is shown here. Furthermore, we observed that this bound is beneficial and enabled us to find optimal 3− and 4−way partitionings for more matrices than if we had only used the local matching bound.

We also remark that it is not necessarily the case that $GL_4 \geq L_4$, because we have implemented a greedy version of the global matching bound GL_4.

Algorithm 1 Global matching bound; $GL_4(G = (V, E))$

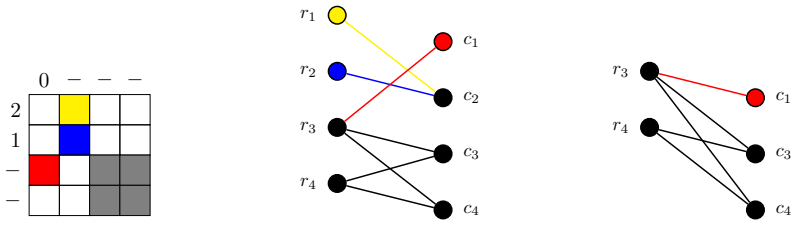
Input: $G = (V, E)$; Graph representation of partial partitioning of a matrix A.
Output: GL_4; Number of matching paths.

```
GL_4 ← 0
V' ← ∅
∀v ∈ V : visited[v] ← False
∀v ∈ V : Matches[v] ← ∅                          ▷ Processor matches of a vertex v
for v ∈ V do
    if v ∈ P_x or (v ∈ P_xy and p > 2) then
        V' ← v
reorder V'                                       ▷ First v' ∈ P_x, then v' ∈ P_xy
for every rc ∈ V' do                             ▷ rc; row/column
    new_Matches ← 0
    queue.push_back(rc)
    while queue ≠ ∅ do
        v = queue.front()
        queue.pop_front()
        for ∀ w s.t. (v, w) ∈ E do
            if rc ∈ P_Φ and w ∈ P_Ψ with Φ ∩ Ψ = ∅ and
            Φ ∩ Matches[w] = ∅ and Ψ ∩ Matches[rc] = ∅ then
                add Ψ to Matches[rc]
                add Φ to Matches[w]
                new_Matches+ = 1
            else if w free and not visited[w] then
                queue.push_back(w)
                visited[w] ← true
    for ∀ visited[u] do
        if u not on matching path then
            visited[u] ← false                   ▷ u can still be used in next BFS
    GL_4+ = new_Matches
```

2.9 Global packing bound (GL_3)

In the previous section we discussed how to generalize the (local) matching bound L_4 and how to make use of the whole graph to find matching paths. We would like to do the same for the local packing bound L_3. The L_3 bound only takes into consideration the unassigned nonzeros in a row/column which is partially assigned to one processor. Based on the nonzeros in these rows/columns and M, where M is the upper bound on the number of nonzeros that we are allowed to assign to a single processor, it determines whether or not a row/column needs to be cut. So in the L_3 bound we only look at the direct neighbourhood of a partially assigned row/column, because we only look at nonzeros in the row/column itself. This is not optimal for all situations.

See for example the partially partitioned matrix and the corresponding bipartite graph in Figure 2.10a and 2.10b. For this partial partitioning $L_1 = 0$, $L_2 = 1$, $P_0 = \{r_3\}$, $P_1 = P_2 = \emptyset$ and the maximum allowed size of a part is $M = 3$. Therefore, the L_3 bound for this example is equal to 0, because $M - n_0 = 3$ and $s_0 = 2$, so $M - n_0 > s_0$. Here, n_0 is the number of nonzeros already assigned to processor 0, and s_0 is the number of nonzeros we need to assign to processor 0 to prevent a cut. However, if we want to avoid cuts, we not only have to assign the free nonzeros a_{33}, a_{34} in row r_3 to part 0, since a_{33} needs to be assigned to part 0, but the rest of the nonzeros in column c_3 must also be assigned to part 0, otherwise there will be a cut in column c_3. The same is true for column c_4, because of nonzero a_{34}. Therefore, all the nonzeros $a_{33}, a_{34}, a_{43}, a_{44}$ need to be assigned to processor 0 to avoid a cut, but this is not possible since $M - n_0 = 3$. At least one of these 4 nonzeros needs to be assigned to another part, and thus one of row r_3, r_4 or column c_3, c_4 needs to be cut. Figure 2.10c shows that the nonzeros need to be assigned to part 0 and form a "neighbourhood" of row r_3 in order to avoid an increase in communication volume. This neighborhood is made starting from $r_3 \in P_0$, and is adjacent to a red edge, i.e. adjacent to a nonzero assigned to part 0.



(a) (b) (c)

Figure 2.10: An example of a partial partitioning of a matrix (a), and its corresponding graph representation (b). A neighborhood adjacent to processor 0 started from row r_3 (c). All the edges in this neighborhood need to be colored red to avoid a cut in a row/column in the partial partitioning.

For each partially assigned row/column in P_i, $i \in \mathscr{P}$, we try to find neighborhoods starting in these rows/columns. A neighborhood $G' = (V', E')$ adjacent to part i consists of at least one vertex $v' \in V'$ with $v' \in P_i$; the rest of the vertices are also in P_i or correspond to free rows/columns. An edge $e' \in E'$ corresponds to a free nonzero in the partial partitioning. The definition of a free vertex is that the corresponding row/column in the partial partitioning of the matrix only contains unassigned nonzeros. A free nonzero is a unassigned nonzero. We obtain the following definition for a neighborhood, which is equal to the definition in the case of $p = 2$:

Definition 1 *[12] Let A be a matrix and $G = (V, E)$ its graph representation. A neighbourhood $G' = (V', E')$ adjacent to processor i satisfies the following requirements:*

- *V' contains at least one vertex $v' \in P_i$.*
- *For $v' \in V'$, if $v' \notin P_i$, then v' is free.*
- *All edges in E' are free.*
- *If $e_1, e_2 \in E'$ and u is an end point of both edges, then $u \in V'$.*
- *G' is connected.*

If we have a neighborhood $G' = (V', E')$ adjacent to row $r_j \in P_i$, then to avoid extra communication volume in the future we need to assign all edges in E' to processor i. As was the case for the L_3 bound this can only be done if $M - n_i \geq |E'|$. Otherwise we are not allowed to assign all $|E'|$ nonzeros to part i. Therefore, if $M - n_i < |E'|$, we need to assign at least some of the nonzeros (=edges) to a different processor than i. This means that at least one of the vertices in V' is cut.

In order to determine the GL_3 bound, we try to find neighborhoods for each partially assigned row/column in P_i with $i \in \mathscr{P}$, starting from these partially assigned rows/columns. These neighborhoods need to be pairwise disjoint, i.e. vertex disjoint. If we have a set of neighborhoods $G'_1 = (V'_1, E'_1), \cdots, G'_l = (V'_l, E'_l)$ adjacent to part i, with $i \in \mathscr{P}$, then to avoid communication volume in the future we need to assign all $S_i = \sum_{k=1}^{l} |E'_k|$ edges of these neighborhoods to processor i. However, if $M - n_i < \sum_{k=1}^{l} |E'_k|$, we need to cut some of the neighborhoods. We start by cutting the neighborhood with the largest number of edges and subtract this number of edges from S_i. We then repeat this until $M - n_i \geq S_i$. Each cut subgraph leads to an increase of one communication volume. We can do this for every $i \in \mathscr{P}$, and sum over all subgraphs that need to be cut to get the global packing bound GL_3.

In the same way as for the L_3 and L_4 bound we need to take the maximum of the GL_4 and GL_3 bounds. We cannot add both global bounds together because they might use the same partial rows/columns and the same unassigned rows/columns. However, we can use the same approach as was done for the "local" L_3, L_4 bounds.

2.9.1 Combined GL_3 and GL_4 bound: GL_5

We have seen how we can combine the local packing bound (L_3) and matching bound (L_4), and we will now use the same approach to combine the global bounds GL_4 and GL_3. First, we will determine the GL_4 bound for the partial partitioning of the matrix, and then we will remove the matching paths that were found from the bipartite graph. Second, we will compute the GL_3 bound for the remainder of the graph. In this way we do not use the edges, i.e. nonzeros, that were used in the GL_4 bound. Moreover, we will only try to find subgraphs adjacent to partially assigned rows/columns which are not on a matching path, and, similar to before, we only look at rows/columns which are partially assigned to one processor.

Pseudo code for GL_3 bound

We will now discuss the implementation of the GL_3 bound after the GL_4 bound has already been determined. The pseudo code for finding the neighborhoods for the GL_3 bound can be found in Algorithm 2.

We will denote the bipartite graph representation of the matrix A with $G = (V, E)$, where the vertices represent the rows/columns, and the edges E represent the nonzeros of the matrix A. Furthermore, we call a row/column free if it is unassigned and also does not intersect with columns/rows that are already assigned in a nonzero. By saying that a row/column is partially assigned to multiple processors, we mean to say that the row/column is unassigned, not free and also not partially assigned to one processor. For the explanation of a row/column being partially assigned to one processor, see section 2.5.

We will use stacks with the following operations: *.pop*() means we remove the top element of the Stack, *.push*() means add element on top of the stack and *.top*() get the top element of the Stack. For a queue, we use *.pop_front*() to remove the first element from the queue and *.push_back*() to add an element to the back of the queue, and *.front*() to get the first element of the queue.

We start with the rows and columns which are all partially assigned to one processor and are not on a matching path in the GL_4 bound. For each of these rows and columns we try to find a neighborhood $G'_i = (V'_i, E'_i)$. We try to expand these neighborhoods simultaneously.

We will denote the queue of partially assigned rows and columns for which we try to find a neighborhood by *Subgraphs*. We start by trying to expand the neighborhood of the first row/column in *Subgraphs*, which is the neighborhood $G'_1 = (V'_1, E'_1)$ of rc_1. Expanding means determining if we can add an edge (i.e. nonzero) to this neighborhood, and in some cases also adding a vertex to V'_i. If we add a vertex w to a neighborhood it is "*Claimed*", meaning that edges $(w, \cdot) \in E$ cannot be added to any of the other neighborhoods. After we successfully expanded the neighborhood $G'_1 = (V'_1, E'_1)$ of rc_1, we will add rc_1 to the back of the queue *Subgraphs* and try to expand the neighborhood $G'_2 = (V'_2, E'_2)$ of rc_2. We repeat this process until we encounter a row/column rc_i for which we are not able to expand the neighborhood anymore. If this is the case, rc_i is removed from the queue *Subgraphs* and the neighborhood $G'_i = (V'_i, E'_i)$ has obtained its final size. We continue until the queue *Subgraphs* is empty. In order to do this we need to keep track of the *Stack* of a neighborhood and keep track of which edges have already been visited. We will do this by using $Stack[\cdot]$ and $Intersect[\cdot]$:

- $Stack[\cdot]$: For every $rc_i \in Subgraphs$ we maintain the $Stack[rc_i]$; the vertex that was last added to the neighborhood $G'_i = (V'_i, E'_i)$ of rc_i is on top of the $Stack[rc_i]$. When we try to expand the neighborhood of rc_i, we start by trying to expand it from the vertex on top of the $Stack[rc_i]$; if this fails we move on to the next vertex in $Stack[rc_i]$.
- $Intersect[\cdot]$: In order to prevent that we add a nonzero more than once, we need to keep track of which nonzeros we already have visited. We do this by keeping track of which edges we have already visited from vertex u. If $Intersect[u] = j$, it means that we have already checked the first $(j-1)$ edges $(u, \cdot) \in E$. So the next time we try to expand the neighborhood from vertex u, we will try to expand the neighborhood with the j^{th} edge $(u, \cdot) \in E$. We denote the other end point of this j^{th} edge with $E_u[j]$.

Now assume we want to expand the neighborhood $G'_i = (V'_i, E'_i)$ of $rc_i \in Subgraphs$, with $rc_i \in P_x$, vertex u on the top of the stack of the neighborhood of rc_i, $Stack[rc_i]$, $Intersect[u] = j$, and vertex $w = E_u[j]$. As explained above, we will try to expand the neighborhood $G'_i = (V'_i, E'_i)$ from vertex u by adding the edge (u, w) to the neighborhood of rc_i, and possibly add vertex w to the neighborhood of rc_i. There are a few possible cases that can occur:

Case 1: Vertex w is on a matching path, where at least one vertex on this matching path must be cut, that vertex potentially being vertex w. We therefore cannot add vertex w to the neighborhood of rc_i, because vertex w could have already been cut. However, we can add the edge (u, w) to the neighborhood of rc_i, because the nonzero corresponding to (u, w) is unassigned since $u \in V'$ and w is on a matching path.

Vertex w can also be partially assigned to multiple processors. In this case, the cuts for w have already been counted in the L_2 bound. Therefore, the vertex w cannot be added to the neighborhood G_i. However, since u is in the neighborhood of rc_i, u is still free or partially assigned to x, and w is unassigned, so (u, w) is unassigned, and to prevent a cut in u we need to assign (u, w) to processor x. Therefore, we can add edge (u, w) to the neighborhood G'.

Another possibility is that w has been assigned to multiple processors, and one of these processors is x. Since vertex w is already assigned, it cannot be added to

the neighborhood of rc_i. However, the nonzero (u, w) is now assigned to processors $x_1, \cdots, x_k$. In order to prevent a cut in a row/column corresponding to vertex u in the future, we need to assign nonzero (u, w) to processor x, and since $x \in x_1, \cdots, x_k$ this is still possible. Therefore, we can add edge (u, w) to the neighborhood of rc_i.

Case 2: If vertex w is free, we can add vertex w to the neighborhood of rc_i if it is not already claimed by a different neighborhood. So we claim vertex w, and we will place w on top of the stack of neighborhood rc_i. Since $u \in G'$ and w is free and not claimed, (u, w) is not in another neighborhood and (u, w) is unassigned. In future assignments we need to assign (u, w) to processor x in order to prevent a cut in the row/column corresponding to vertex u. Therefore, we add edge (u, w) to the neighborhood of rc_i.

Case 3: In this case, we do nothing. If w is partially assigned to one processor and not on a matching path, then we also try to find a neighborhood starting from w. In the implementation of the GL_3 bound, we chose to implement that a neighborhood cannot add nonzeros which are in a row/column that is the starting point of another neighborhood.

If w is assigned to one processor, then all nonzeros in w have already been assigned. Therefore, we cannot add the edge (u, w) to the neighborhood of rc_i.

Similarly, if w is assigned to multiple processors, but x is not one of them, nonzero (u, w) can never be assigned to x in an extension of this partial partitioning, meaning that we cannot add the edge (u, w) to the neighborhood of rc_i.

If w is free, but already claimed by another neighborhood G", then we try to expand this other neighborhood G" from w. The neighborhood G'_i cannot claim nonzeros which are in the row/column corresponding to vertex w, because they may already be in the other neighborhood G".

To conclude, we essentially simultaneously perform a DFS for all vertices $rc \in Subgraphs$. Although we observed that the global packing bound is stronger than the local version in most cases, it is possible that $GL_3 < L_3$.

Algorithm 2 Find all neighborhoods $G'_i = (V'_i, E'_i)$

Input: $Matching\_paths$; Matching paths found in GL_4.
$G = (V, E)$; Partially partitioned matrix
Output: $G'_1 = (V'_1, E'_1), \cdots, G'_l = (V'_l, E'_l)$

```
∀v ∈ V : Claimed[v] ← False
∀v ∈ V : Intersect[v] ← 0                    ▷ Which edge we will examine for vertex v.
∀v ∈ V : Stack[v] ← ∅
for each v ∈ V with v ∉ Matching_paths and v ∈ P_x do
    Subgraphs.push_back(v)                   ▷ Initialize queue: Subgraphs
    Stack[v].push(v)
while Subgraphs ≠ ∅ do
    rc_i ← Subgraph.front()                  ▷ rc_i; row/column
    if Stack[rc_i] = ∅ then                  ▷ Cannot expand the neighborhood of rc_i anymore
        Subgraphs.pop_front()
        continue;
    nonzero_added ← false
    u ← Stack[rc_i].top()
    while not nonzero_added do
        j ← Intersect[u]                     ▷ Look at the j^th edge (u, ·) ∈ E.
        if j = deg(u) then                   ▷ Have checked all edges (u, ·) ∈ E
            Stack[rc_i].pop()                ▷ Remove u from stack of rc_i
            break;
        Intersect[u] ← j + 1
        w = E_u[j]                           ▷ j^th edge is (u, w)
        if w ∈ Matching_paths or w partially assigned to multiple
        processors or ( w is assigned state "x_1 ··· x_k", k > 1 and
        x ∈ {x_1, ··· , x_k} where rc_i ∈ P_x ) then          ▷ Case 1
            Add (u, w) to E'_i               ▷ E'_i: edges in neighborhood of rc_i
            nonzero_added ← true
        else if w is free and not Claimed[w] then             ▷ Case 2
            Add (u, w) to E'_i
            Add w to V'_i                    ▷ V'_i: vertices of the neighborhood of rc_i
            Stack[rc_i].push(w)
            Claimed[w] ← true
            nonzero_added ← true
    if nonzero_added then
        Subgraph.pop_front()
        Subgraph.push_back(rc_i)
```

2.10 Implementation

In the previous sections we discussed the implementation of the global matching and global packing bound. In this section we will describe the other aspects of the implementation of the branch and bound method described in the previous sections. Our algorithm and its implementation work for arbitrary values of $p \in \mathbb{N}_{\geq 2}$. However, we have focused on the cases $p = 3$ and $p = 4$.

2.10.1 Branching strategy

We have already decided that we will branch on the assignment of a row or column to a subset of the processors $\mathscr{P}$. Additionally, we need to decide on the order of the rows and columns which we will branch on. There are many possible orders and the choice of a specific order can have a dramatic influence on the performance of the branch and bound algorithm. It seems natural to start with the rows/columns with the most nonzeros, since the assignment of these rows/columns has the most influence on the load balance and partial assignments of other columns/rows. The following orders are implemented:

`order1`: Branch on the rows and columns in order of descending number of nonzeros in a row/column.

`order2`: First we branch on the row/column with the highest number of nonzeros. Second, we alternate between rows and columns, picking the row/column with highest number of nonzeros each time.

`order3`: Branch on the rows and columns in order of descending number of unassigned nonzeros. We thus first branch on the row/column with the largest number of nonzeros. After the assignment of this row/column, its nonzeros are removed from the matrix, so intersecting columns/rows have one less unassigned nonzero.

If there are two or more more equally likely options for the next row/column, the first row/column is chosen. We use `order3` as our standard order, but for some matrices a different order was used if partitioning the matrix with `order3` did not lead to a result. In [16] different orders were examined, among which `order1` and `order3`. It was found that, on average, `order3` is the best order. The orders were tested in [16] using MondriaanOpt, and so $p = 2$.

Furthermore, if we branch on a row/column, and multiple states are possible for this row/column, we need to decide which subtree we will traverse first. We will traverse the subtrees of the states in ascending order of the number of processors in the state. If there are several states with an equal number of processors, the order in which we will traverse these states is based on the number of processors with minimal part size that are contained within the state, starting with the state which contains the most processors with minimal part size. Minimal part size means that for every processor $i \in \mathscr{P}$, we check the number of nonzeros n_i it owns, and the processor that has the least amount of nonzeros thus has minimal part size. It is possible that there are multiple processors which own the least amount of nonzeros.

For example, assume $p = 3$, and we want to branch on row r. All $2^p - 1$ states are possible in this row, i.e. “0”, “1”, “2”, “01”, “02”, “12”and“012”. Assume also that processors 1 and 2 both have minimal partition size. The order of the subtrees will then become: “1”, “2”, “0”, “12“, “01”, “02”, “012”; states “1" and “2" can be interchanged within this order. The same is true for states “01" and “02". For more information see also section 6.1.

2.10.2 Upper bound

In order to make use of the lower bounds, we need to have an upper bound. If we find a new and better solution while using the branch and bound algorithm, i.e. a $p-$way partitioning of a matrix with lower communication volume than the upper bound, then this will become our new upper bound. However, before we have found a first feasible solution with the branch and bound algorithm, we need to have a initial upper bound.

There are several options for the initial upper bound in the branch and bound algorithm. The trivial upper bound is $(\min(m,n)) \cdot (p-1) + p - 1$ [22]. This upper bound is not tight in most cases; the optimal value is often far removed from this upper bound. An option to get a tighter initial upper bound is to use the output of a heuristic algorithm.

We have chosen to use iterative deepening. We run our branch and bound algorithm with a strict initial upper bound $U_1 = 1$, and we rerun our algorithm with a strict upper bound $U_{i+1} = \lceil 1.25U_i \rceil$ until we have found a solution. This gives a tight upper bound. The iterations of the algorithm before we find a solution are negligible compared to the run with upper bound U_i, in which we find a solution with our algorithm.

2.10.3 Part sizes

During the branch and bound algorithm we need to keep track of the sizes, i.e. the number of nonzeros, of the different parts. We need to keep track of how many nonzeros are already assigned to a single processor. These are the values n_k, $k \in \mathscr{P}$. Additionally, for instance in the case $p = 3$, we need to keep track of how many nonzeros are already assigned to processors 0 and 1 combined. The set of nonzeros already assigned to processor 0 and 1 combined consist of the nonzeros that are assigned state “01" through their row/column assignment, the nonzeros assigned to processor 0 and the nonzeros assigned to processor 1. We will denote the number of nonzeros assigned state “01" with n_{01}. Processors 0 and 1 combined may own $2M$ nonzeros at most, thus we have to make sure that $n_{01} + n_0 + n_1 \leq 2M$. This is of course also true for the other combinations of two processors out of three processors. We will denote the size of nonzeros assigned to processor 0 and 1 combined with the part size of state “01".

Therefore, during the application of the branch and bound algorithm, we will maintain the part sizes of $2^p - 2$ states. This is because we do not have to keep track of how many nonzeros are assigned to all processors combined. When a row/column is assigned, the state of at most C_{max} nonzeros is changed, where C_{max} is the maximum number of nonzeros in a row or column. Every time the state of a nonzero changes we need to update $O(2^p - 2)$ part sizes. Therefore, keeping track of the part sizes costs $O((2^p - 2) \cdot C_{max})$ time.

Before we branch on a state for a row/column we will first check if we can assign this state w.r.t. the part sizes. Therefore, during the whole branch and bound algorithm we will only have feasible partial solutions.

2.10.4 Implementation of the lower bounds

In the previous sections we have described six different lower bounds, 4 local and 2 global. We will now describe how we have implemented these lower bounds.

Explicit cuts, L_1:
When we assign a row/column a state we use a “state" vector of p booleans to describe this state, where a 1 in the state vector means that a processor is used in the state, so a state vector 110 corresponds to the state “01”. Therefore, the number of explicit cuts for a row/column is equal to the sum of the elements of the “state " vector minus 1. This computation takes $O(p)$ time.

Implicit cuts, L_2:
For the second lower bound L_2 we need to maintain a list of which of the $2^p - 2$ states already occur in a row/column for every unassigned row/column. The $2^p - 2$ states include all possible states, minus the state that contains all processors. A state “occurs" in a row/column if there is a nonzero in that row/column that is assigned that state.

Updating the list after a row/column is assigned a state costs $O(C_{max})$, because we need to update the list of the columns/rows that intersect the recently assigned row/column and at most there are C_{max} such columns/rows.

In order to compute the L_2 bound for an unassigned row/column, we need to determine the state with the lowest communication volume which can still be assigned to this row/column given the list of the states which already occur in this row/column. For this we need to determine which states can still be assigned to the unassigned row/column. In the worst case this costs $O(2^p - 2)$ time. So if a row/column is assigned, the recalculation of the L_2 bound costs $O(C_{max} \cdot (2^p - 2))$.

Matching bound, L_4:
We keep track of the partial status of an unassigned row/column, i.e. whether or not a row/column is either free, partially assigned to one or two processors, or none of the aforementioned. When we assign a row/column a state, we need to update the partial status of $O(C_{max})$ unassigned intersecting rows/columns. So because the partial status of at most C_{max} rows/columns changes, we have at most $(p-1) \cdot C_{max}$ vertices that are deleted or added to the bipartite graph $G = (V, E)$ which represents the conflict submatrices. We have a factor of $p - 1$ since every vertex is split into $p - 1$ vertices. For every deletion/addition the calculation of the new maximum matching takes $O(|E|)$ time, [19]. Therefore, the matching bound L_4 can be determined in $O((p-1) \cdot C_{max} \cdot |E|)$ time.

Packing bound, L_3:
For this lower bound we need to keep track of the values s_i^c and s_i^r, with $i \in \mathscr{P}$. In order to do this we maintain a "color" count for each row/column. We keep track of how many nonzeros in a row/column are assigned to a single processor i. If a row $r \in P_i$ we can then determine the number of free nonzeros in this row using this color count.

We use the observation that a row/column contains at most C_{max} nonzeros, and thus contains at most $C_{max} - 1$ unassigned nonzeros. Therefore, we can keep track of the values s_i^r for every processor $i \in \mathscr{P}$ by maintaining a packing set of the rows for processor i consisting of C_{max} buckets. We add an unassigned row to bucket k if it is in P_i and has k unassigned nonzeros. We also maintain such packing sets for the other processors and the columns.

Given a packing set for the rows and processor i we determine the packing bound by iterating over all the buckets. We start by cutting rows from the bucket $C_{max} - 1$. The same can be done for the other processors and columns. So determining the local packing bound costs $O(C_{max})$.

After a row/column is assigned we need to update the color count of at most $O(C_{max})$ intersecting columns/rows and thus also update the packing sets at most $O(C_{max})$ times. Therefore, determining the local packing bound costs $O(C_{max})$ time.

Combined global bounds, GL_5:
After determining the global matching bound, we compute both the local and global packing bound, because the global packing bound is not necessarily larger than the local packing bound. We add the largest of the two to GL_4, resulting in the combined bound GL_5. We can compute the local packing bound L_3 after determining the GL_4 bound, by removing the partially assigned rows/columns that were matched from the packing sets and then computing the local packing bound for the updated packing sets.

Total Lower bound, LB:
If we have a partial partitioning $\hat{\mathcal{B}}$, we first determine the number of explicit and implicit cuts, and then we check whether or not the combination of these lower bounds exceeds

the current upper bound. If this is not the case, we will determine the local packing bound for $\hat{\mathcal{B}}$. If $LB(\hat{\mathcal{B}}) = L_1(\hat{\mathcal{B}}) + L_2(\hat{\mathcal{B}}) + L_3(\hat{\mathcal{B}})$ is larger than the current upper bound, we prune the B & B tree. If this is not the case, the algorithm determines the combination of the local matching and packing bound, L_5. The algorithm again checks if the lower bound using $L_5(\hat{\mathcal{B}})$ (instead of $L_3(\hat{\mathcal{B}})$) exceeds the current upper bound. If this is not the case, it will determine the combination of the global matching bound and a packing bound; GL_5. In this case the lower bound on the partial partitioning will be: $LB(\hat{\mathcal{B}}) = L_1(\hat{\mathcal{B}}) + L_2(\hat{\mathcal{B}}) + \max[L_3(\hat{\mathcal{B}}), L_5(\hat{\mathcal{B}}), GL_5(\hat{\mathcal{B}})]$

3 Nonzero based branch and bound

In order to verify the results and performance of the row-column based branch and bound method described in the previous section, we will take a look at other methods in order to determine the optimal communication volume. One such method again uses a branch and bound algorithm, only now branching on the part to which a nonzero entry is assigned. If we use this method to partition a matrix A with $nz(A)$ nonzeros in p parts, we get a branch and bound tree with $nz(A)$ levels. At each level of the tree a nonzero of the matrix A is assigned to a processor. In this case, the branch and bound tree will have $p^{nz(A)}$ leaves, as was explained in section 2.2. In this section we will discuss how we implemented this nonzero based branch and bound method.

Order
We first had to decide on an order, the order that is used is; essentially assigning the nonzeros row by row. We sort the $nz(A)$ nonzeros on their row location. Subsequently we assign the nonzeros that are in one specific row one by one and then continue by assigning the nonzeros that are in the next row. This order is chosen to allow us to count possible cuts in a row as quickly as possible. Similarly, we could have chosen to instead sort the nonzeros per column and assign the nonzeros per column, in this way we would be able to count the cuts in the columns as quickly as possible.

Symmetry
Many of the $p^{nz(A)}$ possible partitionings are symmetric, similar to the row-column based branch and bound algorithm. For these symmetric partitionings, the communication volume, and the groups in which the nonzeros are divided, is the same, only the assignments of groups to processors differ per symmetric full partitioning. In order to remove these symmetric assignments we use the method described in [16] and [21]. If we are at level $l+1$, meaning we want to assign the $l+1^{th}$ nonzero, and the assignment of the first l nonzeros is $(x_1, x_2, \cdots, x_l)$ with $x_i \in \mathscr{P}$, $\forall\ 1 \leq i \leq l$, then for the assignment of the $l+1^{th}$ nonzero we only consider the processors $x_{l+1} \in \mathscr{P}$ with $x_{l+1} \leq \max(\{x_j : j < l+1\}) + 1$.

Lower bounds
Furthermore, we use lower bounds on partial partitionings to prune the branch and bound tree. We will use three lower bounds to prune this branch and bound tree. The lower bounds bear resemblance to the lower bounds used in the row-column based branch and bound algorithm. The first lower bound we use determines the explicit cuts in the rows and columns. After a nonzero a_{ij} is assigned to a processor, we look at the row and column this nonzero is in, i.e. row r_i and column c_j and determine if after this assignment there is an extra cut in the r_i respectively column c_j. If this is the case we

add the extra cut(s) to the lower bound LB of the partial partitioning.

Additionally we can determine a matching bound. Since we assign the nonzeros row by row, there is at most one row that is partially assigned to a subset of the processors $\mathscr{P}$ at all times. For each other row all the nonzeros in that row are either not assigned, or all nonzeros in the row are assigned to a processor. Therefore, there is at most one row that we can use in determining the matching bound. In order to determine the matching bound we look at the partially assigned row, and see if it has nonzeros in a column that is partially assigned to different processor(s). Therefore, this lower bound can be at most equal to $p-1$. This is similar to the local matching bound L_4 and is also used in [21].

The third lower bound is a packing bound. As stated in the previous paragraph, there is at most one row partially assigned. However, there can be multiple columns that are partially assigned. Since there is only one row partially assigned we will ignore this row for the packing bound and determine the packing bound on the partially assigned columns in the same way as the local packing bound L_3 was determined. We look at all columns that are partially assigned to only one processor $k \in \mathscr{P}$ and, similar to the L_3 bound, we need to cut columns if $s_k > M - n_k$, where M is the maximum allowed size of a part, n_k is the number of nonzeros which are already assigned to part k, and s_k is the number of nonzeros that we need to assign to part k in order to avoid a cut. We determine the least number of columns in P_k^c that need to be cut to make sure that $M - n_k \leq s_k$, giving a lower bound on the number of columns in P_k^c which need to be cut. This can be done for every processor in $k \in \mathscr{P}$ and the packing bound is the sum of these p bounds.

Similar to the row-column based branch and bound method, the packing bound and matching bound conflict, so we need to take the maximum of the matching bound and packing bound when we determine a lower bound on a partial partitioning.

An example of a partial partitioning of a matrix determined by the nonzero based branch and bound method is given in Figure 3.1, with $p = 3$ and $\epsilon = 0.03$. There are three explicit cuts in this partial partitioning: one in row r_2 and two in column c_2. Row r_3 is partially assigned to processor 2 and it intersects with the columns c_3 and c_4 in a nonzero; both columns are partially assigned to processor 0. Therefore, the matching bound is equal to one. Notice that the matching bound is not equal to two, since column c_3 and c_4 are partially assigned to the same processor. We will now determine the value of the packing bound. Three columns are partially assigned to one processor: column c_1, c_3 and c_4 are all partially assigned to processor 0. For this matrix $M = 5$, $n_0 = 4$ and $s_0 = 6$, so $1 = M - n_0 < s_0 = 6$. Therefore, we need to cut the columns c_3 and c_4. The packing bound for this partial partitioning is thus equal to two. We can conclude that the lower bound on the communication volume of this partial partitioning is equal to $3 + \max(1, 2) = 5$.

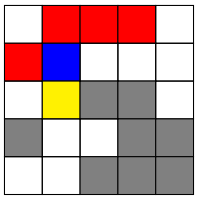


Figure 3.1: Partial partitioning of a matrix, for $p = 3$. The assignment of the nonzeros is indicated by their color: red nonzeros are assigned to processor 0, blue nonzeros are assigned to processor 1, yellow nonzeros are assigned to processor 2 and grey nonzeros are unassigned (the zeros in the matrix are white).

4 | Integer linear program

In section 2 we described a branch and bound based method to solve the matrix partitioning problem. Another way to optimally solve the sparse matrix partitioning problem is to formulate it as an integer linear program (ILP). A general ILP problem has the following form:

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

with given vectors c, b and matrix A. It is known that integer linear programming is NP-complete [17]. Therefore, we cannot expect to solve large problem instances with this method.

Several software packages exist that can solve integer linear programs, although we cannot expect that these packages are able to solve large instances optimally. One of these software packages is CPLEX [1], which we will use in this thesis. The ability to solve an ILP highly depends on the formulation of the integer linear program. We can often formulate a problem as several different integer linear programs. The formulation of the ILP which performs the best is the formulation for which the feasible set of solutions of the linear programming relaxation $\{x \in \mathbb{R}^n | Ax \leq b\}$ is as close as possible to the convex hull of the feasible solutions of the integer linear program. The convex hull of the feasible solutions of the integer linear program is the smallest polyhedron that contains all feasible integer solutions of the ILP problem. Therefore, formulations with less variables or constraints are not necessarily the best.

In section 1.1, we saw that we can model the matrix partitioning problem as a hypergraph partitioning problem. In this section we will formulate a hypergraph partitioning ILP model that is similar to the ones in [13, 18].

4.1 Fine-grain hypergraph model

The partitioning of an $m \times n$ matrix A with $nz(A)$ nonzeros in p parts can be modeled as partitioning the vertices of the hypergraph $H = (V, N)$ with $|V| = nz(A)$ vertices and $|N| = m + n$ nets in p parts, see also section 1.1. We define the following decision variables:

$$\begin{aligned} x_{ij} &= \begin{cases} 1 & \text{if vertex } i \text{ is in part } j \\ 0 & \text{otherwise} \end{cases} \\ y_{kj} &= \begin{cases} 1 & \text{if net } k \text{ has vertices in part } j \\ 0 & \text{otherwise,} \end{cases} \end{aligned} \tag{4.1}$$

with $1 \leq i \leq |V|$, $j \in \{1, \cdots, p\}$ and $1 \leq k \leq |N|$. The objective is to minimize the communication volume in the partitioning of hypergraph $H = (V, N)$. This gives the following objective function:

$$\min \sum_{k=1}^{|N|} (\sum_{j=1}^{p} y_{kj} - 1) \tag{4.2}$$

For every net k we count the number of parts that contain vertices of net k, i.e. $y_{k1} + \cdots + y_{kp} = \lambda_k$, so net k contributes $\lambda_k - 1$ to the communication volume of the partitioned hypergraph. Notice that we implicitly assumed that every net is non-empty, and thus assumed that all rows/columns contain nonzeros. If there are rows/columns without nonzeros, we can remove them from the matrix before formulating the integer linear program.

Now we will determine the constraints of this ILP formulation. Every vertex can be in only one partition. This gives the following vertex constraints:

$$\sum_{j=1}^{p} x_{ij} = 1, \quad \forall i \tag{4.3}$$

We also need to incorporate the load balance constraint into the constraints for the ILP. Let M be the maximum allowed size of a part, then the load imbalance constraints are:

$$\sum_{i=1}^{|V|} x_{ij} \leq M, \quad \forall j \tag{4.4}$$

Furthermore, if $x_{ij} = 1$ and $i \in n_k$, then net n_k has a vertex in partition j, so y_{kj} has to be equal to 1. We therefore get the following net constraints:

$$x_{ij} \leq y_{kj}, \quad \forall j,\ k,\ i \in n_k \tag{4.5}$$

Together, these equations (4.1) - (4.5) give the following ILP problem:

$$\begin{aligned}
\text{minimize} \quad & \sum_{k=1}^{|N|} (\sum_{j=1}^{p} y_{kj} - 1) \\
\text{subject to} \quad & \sum_{j=1}^{p} x_{ij} = 1 && \forall i, \\
& \sum_{i=1}^{|V|} x_{ij} \leq M && \forall j, \\
& x_{ij} \leq y_{kj} && \forall j,\ k,\ i \in n_k, \\
& y_{kj} \in \{0, 1\} && \forall k,\ j, \\
& x_{ij} \in \{0, 1\} && \forall i,\ j.
\end{aligned} \tag{4.6}$$

This formulation has $nz(A) \cdot p$ variables x_{ij} and $(m + n) \cdot p$ variables y_{kj}, so in total there are $p(nz(A) + m + n)$ decision variables. There are $nz(A)$ vertex constraints, p load balance constraints and $2 \cdot p \cdot nz(A)$ net constraints. Since every nonzero is in two nets, every nonzero appears in two net constraints; one for its row and one for its column. Because there are $nz(A) \cdot p$ nonzero variables, there are also $2 \cdot p \cdot nz(A)$ net constraints. Therefore, the total number of constraints is $nz(A) + p(2 \cdot nz(A) + 1)$.

As was the case for the branch and bound method, every feasible solution of the ILP model (4.6) has $p!$ equivalent solutions including itself, see also section 2.2. In order to

remove part of these equivalent symmetric feasible solutions, we fix the assignment of the first vertex (vertex 1) to part 1, so we add the following symmetry constraint:

$$x_{11} = 1. \tag{4.7}$$

The final ILP model, including the symmetry constraint, that we will use is:

$$\begin{aligned}
\text{minimize} \quad & \sum_{k=1}^{|N|}(\sum_{j=1}^{p} y_{kj} - 1) & \\
\text{subject to} \quad & \sum_{j=1}^{p} x_{ij} = 1 & \forall i, \\
& \sum_{i=1}^{|V|} x_{ij} \leq M & \forall j, \\
& x_{ij} \leq y_{kj} & \forall j, k, i \in n_k, \\
& x_{11} = 1, & \\
& y_{kj} \in \{0, 1\} & \forall k, j, \\
& x_{ij} \in \{0, 1\} & \forall i, j.
\end{aligned} \tag{4.8}$$

5 | Recursive bipartitioning

In section 2, we described an exact branch and bound method for computing an optimal partitioning of a matrix A into p parts. This algorithm directly partitions the $nz(A)$ nonzeros of a matrix into p parts. Another method that is often used to get a $p-$way partitioning of a matrix A is recursive bipartitioning (RB). The recursive bipartitioning method works as follows: we start with the set of all $nz(A)$ nonzeros, then we split (bisect) the nonzeros of the matrix A 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 p subsets, meaning we have obtained a $p-$way partitioning. Therefore, if $p = 2^q$, there are q moments at which we split each subset. N.B., if p is not a power of 2, the recursive bipartitioning method can be adapted to work in that situation as well [3].

We will study the recursive bipartitioning method which uses an exact method in order to bisect a subset, i.e. it bisects a subset of nonzeros into two sets of approximately equal size in such a way that it minimizes the communication volume that arises from the bisection.

The recursive bipartitioning method is greedy: although it bisects a subset optimally, it does not take into account the successive bisections while doing this. It only takes into account the current subset and bisects this subset in such a way that it minimizes the communication volume and upholds the load balance criterion. It does not consider the minimization of the communication volume of the whole matrix; it only considers the minimization of the communication volume of the subset it is currently bisecting. We must thus ask ourselves the extent to which the recursive bipartitioning method is able to approach the optimal minimal communication volume that is possible for a $p-$way partitioning.

We will try to find an answer to this question. We will do this by comparing the results obtained by the recursive bipartitioning of a matrix A for the case $p = 4$ with the optimal value that was determined by using direct $4-$way partitioning.

First we explain how we can efficiently determine the communication volume when using RB. We will define a $p-$way partitioning of a matrix A as the partitioning of the $nz(A)$ nonzeros of a matrix A into p mutually disjoint subsets $\{A_0, \cdots, A_{p-1}\}$, with $\cup_{h=0}^{p-1} A_h = A$. The communication volume of a $p-$way partitioning is the communication of the SpMV for the matrix $\cup_{h=0}^{p-1} A_h = A$, where the nonzeros of subset A_h belong to processor h. Assume we have a $p-$way partitioning of the nonzeros of a matrix A into p mutually disjoint subsets $A_0, \cdots, A_{p-1}$ and we subsequently split the subset A_{p-1}. The communication volume of the resulting $(p+1)-$way partitioning is the sum of the old communication volume of the $p-$way partitioning and the communication volume that arises from the bisection of the subset A_{p-1}. This is formalized in the following theorem.

Theorem 1 *[3, p. 215] Let A be an $m \times n$ matrix and let $A_0, \cdots, A_{k-1} \subset A$ be mutually*

disjoint subsets of nonzeros, with $k \geq 1$. Then

$$CV(A_0, \cdots, A_k) = CV(A_0, \cdots, A_{k-1} \cup A_k) + CV(A_{k-1}, A_k). \tag{5.1}$$

Therefore, given that $p = 4$, if we use RB and first split the nonzeros of a matrix A into two subsets A_{01} and A_{23}, and subsequently split the sets A_{01}, A_{23} into the subsets A_0, A_1 respectively A_2, A_3, the communication volume of the resulting $4-$way partitioning $\{A_0, A_1, A_2, A_3\}$ of matrix A is;

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

If we use the recursive bipartitioning method we also need to decide which load imbalance parameter ϵ' we will use at each bisection to guarantee a final load imbalance of at most ϵ for the final $p-$way partitioning.

Assume we want to partition a matrix A into $p = 2^q$ parts, with final load imbalance of at most ϵ. From the load balance constraint (1.2) it follows that the size of a part in the final $p-$way partitioning $\{A_0, \cdots, A_{p-1}\}$ may have a size of at most $M = (1+\epsilon)\lceil \frac{nz(A)}{p} \rceil$. To simplify the calculations we will define $M' = (1+\epsilon) \cdot \frac{nz(A)}{p}$, so $M' \leq M$.

We will now determine the maximum possible size of a subset of nonzeros which can occur in the final $p-$way partitioning when using RB with load imbalance parameter ϵ' for every bisection. Assume we are at the first step of the RB algorithm, and we bisect the nonzeros of A into two subsets A_0 and A_1, with A_1 possessing the maximum allowed size, meaning $nz(A_1) = (1+\epsilon') \cdot \frac{nz(A)}{2}$. We will subsequently bisect A_1 into two subsets of which A_1' has the maximum allowed size such that $nz(A_1') = (1+\epsilon') \cdot \frac{nz(A_1)}{2} = (1+\epsilon')^2 \cdot \frac{nz(A)}{2^2}$. If we repeat this process and recursively split the maximum subset q times, then the maximum possible size of a subset in the final $p-$way partitioning has size:

$$(1+\epsilon')^q \cdot \frac{nz(A)}{2^q}. \tag{5.2}$$

In order to guarantee that this size is smaller than M', we need to have:

$$(1+\epsilon')^q \leq 1+\epsilon. \tag{5.3}$$

Solving this for ϵ' gives us the maximum possible value $\epsilon' = (1+\epsilon)^{1/q} - 1$. However, we can also use the first order approximation $(1+\epsilon')^q \approx 1 + q\epsilon'$. Substituting this approximation into (5.3) gives:

$$\epsilon' = \frac{\epsilon}{q}. \tag{5.4}$$

This will be the ϵ' we will use at the first split of the recursive bisection. After the first split we will use a so-called "adaptive" epsilon.

After the first split we will have two subsets, A_0 and A_1; one of these subsets can be smaller than the other one. Both subsets will be partitioned into $p/2$ subsets. We can determine for both subsets the value of the load imbalance parameter, ϵ_k, such that the size of any of the $p/2$ subsets is at most M'. Therefore, the value of ϵ_k, with $k = 0, 1$, is equal to:

$$\begin{aligned} (1+\epsilon_k)\frac{nz(A_k)}{p/2} = (1+\epsilon)\frac{nz(A)}{p} \iff \\ \epsilon_k = \frac{M'}{nz(A_k)} \cdot \frac{p}{2} - 1. \end{aligned} \tag{5.5}$$

We can see that ϵ_k is larger if the subset is smaller; the smaller the subset, the more freedom we have to bisect this subset. This freedom can lead to better bisections w.r.t. the communication volume. This value of ϵ_k is valid if we directly partition the subsets in $p/2$ parts. We will now determine the value of ϵ'_k if we will use RB to partition the subset A_0 respectively A_1 in $p/2$ parts. Let $\epsilon'_k = \epsilon_k$; if we use this value of ϵ'_k to recursively bisect the subset A_0 respectively A_1 in $p/2 = 2^{q'}$ subsets, then again we have to make sure that the final partition fulfills the load imbalance criterion. So we want the following:

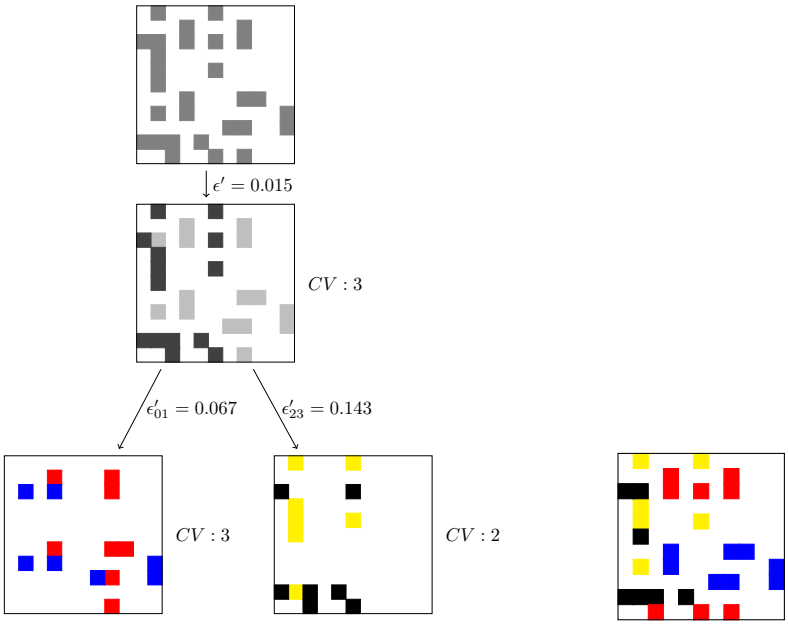
$$(1 + \epsilon'_k)^{q'} \leq \frac{nz(A)}{p \cdot nz(A_k)} \cdot \frac{p}{2}(1 + \epsilon). \tag{5.6}$$

We will use the linear approximation $(1+\epsilon_k)^{q'} \approx 1 + q'\epsilon_k$ and substituting this in (5.6), ϵ'_k becomes equal to:

$$\epsilon'_k = \frac{\frac{M'}{nz(A_k)} \cdot \frac{p}{2} - 1}{q'}. \tag{5.7}$$

This is the ϵ' we will use in the second step of the recursive bipartitioning. Although we use the approximation $(1+\epsilon'_k)^{q'} \approx 1 + q'\epsilon_k$, for $q' = 1$ the approximation is exact, so the final partitioning obtained with RB method will fulfil the load imbalance criterion.

An example of a 4−way partitioning of a matrix obtained by recursive bipartitioning, and the optimal 4−way partitioning of this matrix, is given in Figure 5.1. The load imbalance parameter for the optimal partitioning was set to $\epsilon = 0.03$, so $M = 8$. In the first step of the recursive bipartitioning, we bisect the nonzeros (grey) into two groups, A_{01} and A_{23}, shown in figure 5.1a as light-grey nonzeros and dark-grey nonzeros. For this first bisection we used the load imbalance parameter from equation (5.4), so $\epsilon' = 0.015$. For the second step in the recursive bipartitioning we used the load imbalance parameter from equation 5.7. Subset A_{01} has size 15, so the load imbalance parameter $\epsilon'_{01} \approx 0.067$. Subset A_{23} has size 14, so $\epsilon_{23} \approx 0.143$. During the second step we bisect A_{01} into the two subsets A_0 (red) and A_1 (blue) using ϵ_{01}, and bisect subset A_{23} into the two subsets A_2 (yellow) and A_3 (black) using ϵ_{23}, see Figure 5.1a. Next to each matrix in Figure 5.1a is the communication volume arising from bisection of the nonzeros in that matrix. Using Theorem 1, we find that the communication volume of the final 4−way partitioning obtained by recursive bipartitioning is equal to 3+3+2. We see that this communication volume is not necessarily equal to the minimum possible communication volume of a 4−way partitioning of this matrix, because the optimal communication volume is equal to 7; see also figure 5.1b for the optimal 4−way partitioning of this matrix.



(a) Recursive bipartitioning (b) Optimal 4−way partitioning

Figure 5.1: The recursive bipartitioning of the matrix `Tina_AskCal` with $nz(A) = 29$ (a), and an optimal 4−way partitioning of the matrix `Tina_AskCal`, with $\epsilon = 0.03$ (b). The first recursive bipartitioning step entails splitting the nonzeros (grey) into two subsets, A_{01} (light-grey) and A_{23} (dark-grey). During the second bipartitioning step, the nonzeros of subset A_{01} are split into subsets A_0 (red) and A_1 (blue). Similarly, the nonzeros of subset A_{23} are split 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 bisection. The communication volume of the final 4−way partitioning is equal to eight. The communication volume of the optimal 4−way partitioning is equal to seven.

6 | Experimental results

We have implemented the row-column based branch and bound method from section 2 in the C++ programming language. In the remainder of this thesis we will call this matrix partitioner the general matrix partitioner "GMP".

First, we will make some general observations and remarks about the GMP method. Second, we look at the performance of the nonzero based branch and bound method, described in section 3. Third, we will compare the GMP method with the ILP method from section 4. Both methods determine the optimal matrix partitioning for $p \in \mathbb{N}_{\geq 2}$. We will analyze the performance of these methods on determining optimal 2−, 3− and 4−way partitionings. For the case $p = 2$ we will also compare both methods with the optimal matrix bipartitioners "MondriaanOpt" [19] and the "MatrixPartitioner MP" [12]. Finally, we will use the communication volumes of the optimal 4−way partitionings to analyse the performance of the recursive bipartitioning method, see section 5.

For all the matrix partitionings that were executed, the load imbalance parameter was set to $\epsilon = 0.03$.

The ILP model (4.8) was implemented using IBM ILOG CPLEX 20.1.0 [1]. We set the value of the parameter controlling the thread count to one, meaning we let CPLEX only use 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. The source code of the GMP method and the ILP method can be found at [1].

Most experiments were run on a system with an AMD 3800XT with 8 Cores and 16 threads running at 4.3GHz, and 16 GB of random-access memory running at 3000MHz. The experiments relating to the comparison of the ILP method to the GMP method for the case $p = 3$ were instead executed on a system with an Intel i7-7700k with 4 cores and 8 threads, running at 4.2 Ghz, and 8 GB of RAM.

6.1 Remarks GMP method

As said in the implementation section, we have used `order3` as our default order, since both our observations and the observations made in [16] confirm that that specific order is the fastest. However, `order3` is not the best choice for all matrices, see for instance Table 6.1. The computation time of determining an optimal 4−way partitioning for three matrices is given in Table 6.1. When comparing `order2` and `order3`, we are able to see that `order2` is significantly faster for these three matrices.

Whenever we did not succeed in determining an optimal 3− or 4−way partitioning for a matrix, we instead tried partitioning the matrix using `order2`. We choose `order2` over `order1`, because, in general, there was a greater difference in the order of the rows and columns between `order2` and `order3` than between `order1` and `order3`.

While trying to determine an optimal 3− or 4−way partitioning for a subset of our matrix sets, we observed that the GMP method was not able to leave the first subtree

[1] `https://github.com/lienjenns/Thesis_Matrix_Part`

and move onto the second subtree during its iterations. In order to resolve this issue, we tried to reverse the order of the subtrees of the first row/column. For instance, if $p = 3$, instead of traversing the subtrees in the order "0", "01", "012", we would traverse them in order of "012", "01", "0".

	Computation time (s)	
name	`order2`	`order3`
farm	3739	92685
kleemin	198	77736
GD00_a	1.2	3409

Table 6.1: The computation times of determining an optimal 4−way partitioning using the GMP method with two different orders are given in seconds. The load imbalance parameter was set to $\epsilon = 0.03$.

We tried to find the optimal communication volume for the 60 matrices of the SuiteSparse Matrix Collection with the least amount of nonzeros. As such, we gave the GMP partitioner at least 48 hours of computation time, to try to find a solution within 48 hours. If the algorithm was able to find a solution, we then let it run for at least 5 more days to see if it could find the optimal communication volume. For the rest of the matrices in the SuiteSparse Matrix Collection with less than 1000 nonzeros, we focused on the matrices that had a low communication volume, and were solvable in the case of $p = 2$ in at most a few seconds. We noticed that both the communication volume of the optimal bipartitioning and the computation time needed to determine this optimal bipartitioning gave us a good indication whether or not we would be able to find an optimal 3−way partitioning. In general, the higher the optimal communication volume and the computation time (for $p = 2$), the longer it will take to determine an optimal 3−way partitioning.

The same approach was used for determining a 4−way partitioning. The difference being that for the rest of the SuiteSparse Matrix Collection we now use the results for $p = 3$, and focus on the matrices that have both a low communication volume and were solvable in the case of $p = 3$ in at most a few minutes.

For 101 matrices of the SuiteSparse Matrix Collection with less than 1000 nonzeros, we succeeded in computing the optimal 3−way partitioning using the GMP method. In the case that $p = 4$, we were able to find the optimal communication volume of 62 matrices of the SuiteSparse Matrix Collection with less than 1000 nonzeros. The optimal communication volumes and computation times for $p = 3,\ 4$ can be found in Appendix A.

We were able to find the optimal $p = 3,\ 4-$way partitioning for a number of matrices using `order2`, or by reversing the order of the subtrees for the first row or column. These matrices are indicated with an ^{2} respectively ^{R} in Appendix A.

6.2 Nonzero based B & B method

In section 3 we described another branch and bound method based on branching on the assignment of nonzeros. We will compare this method, which we will call the "NZ_BB" method, to the GMP method. Since the NZ_BB method only contains local bounds, we will also compare the NZ_BB method with the GMP method while only the local bounds are activated, i.e. $LB = L_1 + L_2 + \max(L_3, L_5)$. We will denote this method with "GMP (local)".

The computation times for $p = 3,\ 4$ of the three separate algorithms are given in

Table 6.2 for a small set of sparse matrices. We can clearly see that the nonzero based branch and bound method NZ_BB performs much worse than both the row-column based branch and bound method, with all bounds activated, and the GMP method, with only the local bounds activated. Only for the matrix `relat_3` in the case of $p = 4$ does the NZ_BB method outperform both GMP methods.

Notice also that the GMP (local) is faster than the GMP method with global bounds in approximately half of the cases. This is most likely because this test set consists of very small matrices, for which the global lower bounds have little to no added value. Although the global lower bounds can potentially prune larger parts of the branch and bound tree than the local bounds, they are also more expensive to compute.

		Computation time					
		GMP		GMP (local)		NZ_BB	
Name	p	Abs (s)	rel	Abs (s)	rel	Abs (s)	rel
cage3	3	0.249	1	0.263	1.06	3.183	12.78
	4	10.816	1	9.694	0.9	24.106	2.23
lpi_galenet	3	0.032	1	0.056	1.75	0.407	12.72
	4	0.245	1	0.399	1.63	2.069	8.44
relat3	3	7.062	1	5.678	0.8	22.126	3.13
	4	310.945	1	271.221	0.87	69.404	0.22
lpi_itest6	3	0.044	1	0.061	1.39	0.804	18.27
	4	12.101	1	11.469	0.95	130.412	10.78
Tina_AskCal	3	0.535	1	0.544	1.02	31.089	58.11
	4	20.496	1	18.921	0.92	423.747	20.67
GD98_a	3	0.071	1	0.054 s	0.76	61.632	868.06
	4	8.49	1	6.036	0.71	1041.85	122.71

Table 6.2: Comparison of computation time between the three methods mentioned in the above text: the nonzero based branch and bound method “NZ_BB" (described in section 3), the row-column based branch and bound method "GMP" and the GMP method with only the local bounds activated "GMP (local)". Computation time is given in seconds (Abs) and in time relative to the GMP method.

6.3 Comparison for $p = 2$

In this section we will compare the matrix bipartitioners “MondriaanOpt" [19] and “MatrixPartitioner (MP)" with the matrix partitioners that work for general $p \in \mathbb{N}_{\geq 2}$: the ILP method and GMP method. We compute the optimal bipartitionings with $\epsilon = 0.03$ for the 160 matrices of the SuiteSparse Matrix Collection with less than 500 nonzeros. We allowed for a maximum computation time of 12 hours per matrix.

In Figure 6.1, the performance profile plot of the four methods is shown. 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 timelimit of 12 hours. The ILP method was also able to compute the bipartitionings of all matrices within the set time limit, but was slower than the matrix bipartitioner MP. However, it outperforms the GMP method with regards to both the computation time and the number of solved matrices. Similarly, the ILP method seems to outperform MondriaanOpt with regards to the number of solved matrices and the computation time. However, the ILP method seems to be slower for

matrices that take under one second to solve. The GMP method is able to determine the communication volume for more matrices than MondriaanOpt, see also Figure 6.2. However, we also see in Figure 6.1 that MondriaanOpt is in general faster than the GMP method.

In Table 6.3, the number of matrices for which a method failed to determine an optimal bipartitioning within the time limit of 12 hours is shown (no failed). Furthermore, the normalized geometric averages are shown for the four methods. The geometric average is based on the normalized computation time (w.r.t computation time of the MP algorithm). The geometric average was based on the 143 matrices that all methods were able to solve. Once again, we see that the MP algorithm is the fastest algorithm, and that the GMP is the slowest algorithm. However, it is also interesting to see that the geometric averages indicate that, on average, MondriaanOpt is approximately 3 times faster than the ILP method. This was not immediately clear from the performance profile plot in Figure 6.1.

	MP	MondriaanOpt	GMP	ILP
no failed	0	16	11	0
Geometric avg.	1	12.7	168.7	39.5

Table 6.3: Number of unsolved matrices for each method (no failed), and the geometric average based on the normalized computation time of the 143 matrices that all methods were able to solve. Computation times were normalized w.r.t the computation time of MP.

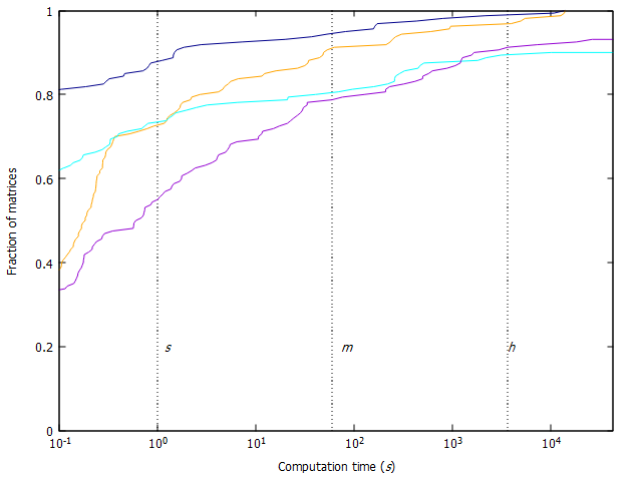


Figure 6.1: Performance profile plot comparing the computation time of the four methods “MondriaanOpt" (blue), matrix partitioner MP (dark-blue), the ILP method (orange) and the GMP method (purple). Note the logscale for the computation time (x-axis).

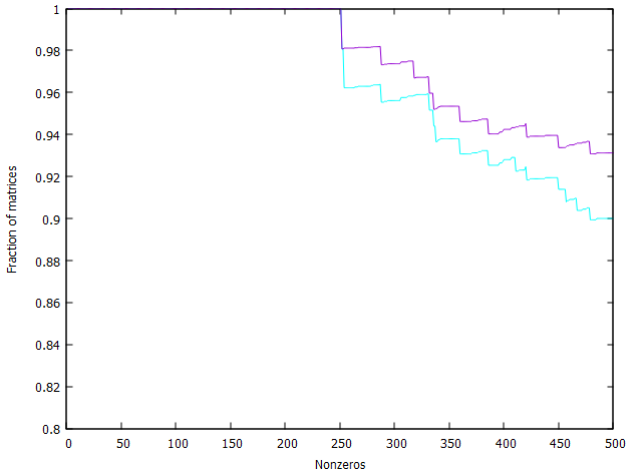


Figure 6.2: Fraction of matrices with a number of nonzeros smaller than or equal to the $x-$axis that were solved by respectively Mondriaan_Opt (blue) or GMP (purple).

6.4 Comparison for $p = 3,\ 4$

For the case $p = 3,\ 4$, we compared the GMP method with the ILP method. For $p = 3$ this comparison was done for the 101 matrices with less than 1000 nonzeros that the GMP method was able to solve. For $p = 4$ we compared the computation time of both methods for the 62 matrices for which the GMP algorithm was able to determine the optimal $4-$way partitioning. Since both methods are exact, the communication volumes of the optimal $3-$ and $4-$way partitionings found by both methods should be the same, as was the case here. Figure 6.3 shows the fraction of the 101 matrices for which either method was able to determine the optimal $3-$way partitioning within a given time. Figure 6.4 displays the performance of both methods in computing $4-$way partitionings. We see that the ILP method outperforms the GMP method. The GMP method needs hours or days to compute the optimal partitioning for part of the matrices in the test sets. The ILP method however is able to determine the optimal communication volume for all matrices in the test set within minutes.

Additionally, in Table 6.4, the normalized geometric averages (normalized w.r.t the computation time of the ILP method) are given for both methods. This confirms that the ILP method is much faster than the GMP method, which we saw before in Figures 6.3 and 6.4. In the case of $p = 4$, the ILP method is on average 182 times faster than the GMP method. The geometric averages in Table 6.3 and 6.4 also indicate that the difference between the performance of the ILP method and the performance of the GMP method increases if p increases.

p	ILP	GMP
3	1	70.5
4	1	182.8

Table 6.4: The geometric average of the normalized computation times (w.r.t to the ILP method) for both ILP method and GMP method.

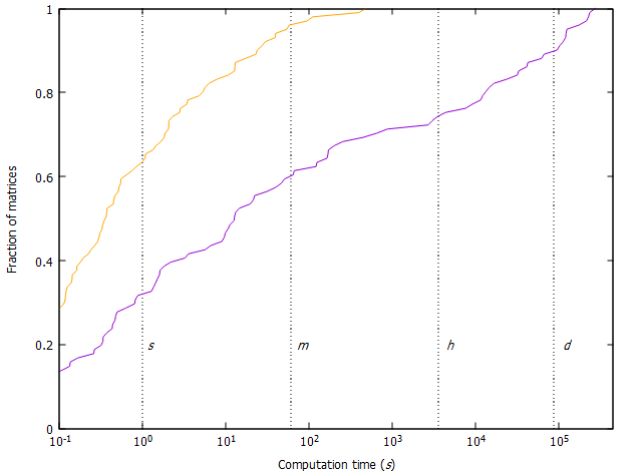


Figure 6.3: Fraction of matrices of all the 101 matrices that were solved for which the optimal communication volume was found within the given time. The orange graph is the ILP method and the purple graph is the branch and bound based method. For the case $p = 3$. Note the logscale for the computation time

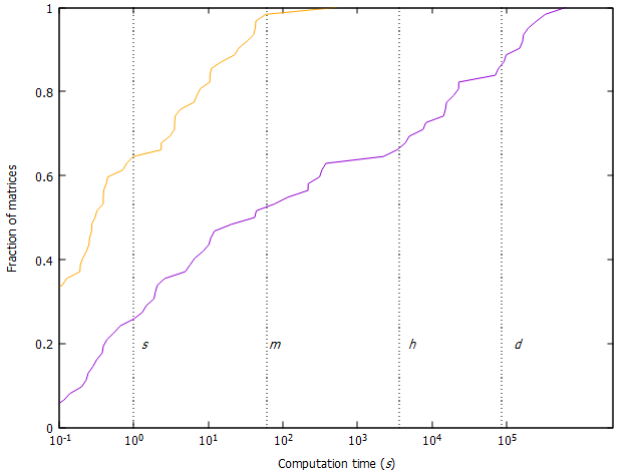
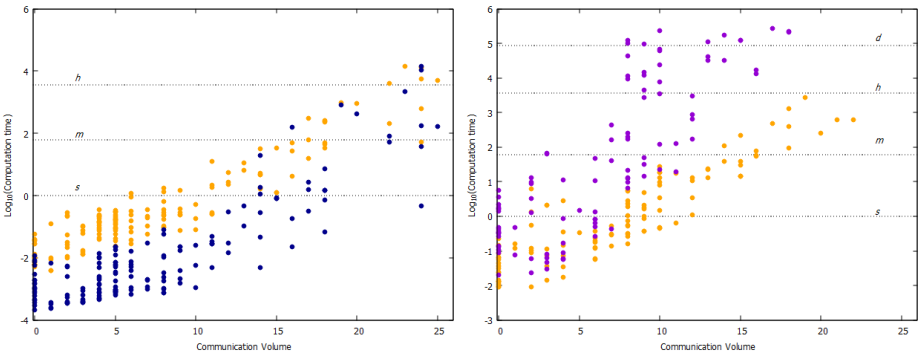


Figure 6.4: Fraction of matrices of all the 62 matrices that were solved for which the optimal communication volume was found within the given time. The orange graph is the ILP method and the purple graph is the branch and bound based method. For the case $p = 4$. Note the logscale for the computation time

After observing that the ILP method is much faster than the GMP method, we decided to use the ILP method to supplement the results of the matrices of the SuiteSparse Matrix Collection with less than 250 nonzeros for which we were not able to determine an optimal 3− or 4−way partitioning using the GMP method. We let the ILP method run for 24 hours at most. The ILP method was able to find an optimal $3, 4-$way partitioning for most matrices mentioned above, but the method was not able to find a solution for a number of matrices due to memory constraints. This is because CPLEX stores information about the branch and cut tree, and when this tree gets very large the program runs into memory issues. The program generally used 5 GBs of RAM during iterations on matrices for which it failed to find a solution, with one outlier as high as 10

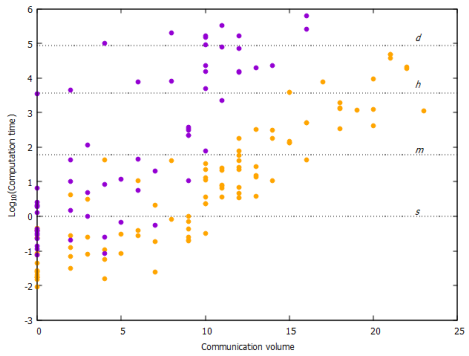
GBs while trying to determine the 4−way partitioning of the matrix `ibm32`. This could possibly be solved by storing the information regarding the branch and cut tree on the disk. However, further research needs to be done on this topic, to find out whether or not this would be able to solve the problem mentioned above, and if this will ultimately affect the total computation time. The optimal solutions and the computation time for 3− and 4−partitionings can be found in Appendix A.

There is no single specific characteristic of a matrix that acts as a predictor of whether or not it will be hard to find an optimal partitioning. However, we have seen a connection between the optimal communication volume of a matrix, and the time it takes to compute this optimal partitioning. In general, the higher the optimal communication volume, the more time it takes to compute the optimal partitioning. In Figure 6.5, the log of the computation time is given as a function of the optimal communication volume. For $p = 2$, the data of the MP algorithm and ILP method are shown, and for $p = 3, 4$ the data of the ILP method and GMP method are shown.



(a) $p = 2$

(b) $p = 3$



(c) $p = 4$

Figure 6.5: The log of the computation time as function of the communication volume, for $p = 2, 3, 4$. Data of the MP algorithm is indicated with dark-blue, the ILP method with orange and the GMP method with purple.

6.5 Recursive bipartitioning

We will now examine the performance of the recursive bipartitioning (RB) method, see section 5. We will examine the quality of the p−way partitioning of a matrix A obtained by the recursive bipartitioning method for the case $p = 4$. We will do this by partitioning a matrix A into 4 parts, setting the load imbalance parameter $\epsilon = 0.03$ and

comparing the communication volume determined by the RB method with the optimal communication volume.

The recursive bipartitioning method uses the exact bipartitioner MatrixPartitioner (MP) to bisect the subsets. The first step of this bipartitioner uses the load imbalance parameter from equation (5.4), so $\epsilon' = 0.015$. During the second step the load imbalance parameter for each of the subsets is determined using the formula (5.7), which simplifies to $\epsilon'_k = \frac{M'}{nz(A_k)} \cdot 2 - 1$. After the first bisection it is possible that some nonzeros remain free, meaning that they can be assigned to either processor 0 or processor 1. In this case we will traverse these free nonzeros, and sequentially assign the nonzeros to the processor which owns the least nonzeros.

We test the recursive bipartitioning method on the set of 100 matrices in the SuiteSparse Matrix Collection for which we were able to determine the communication volume for an optimal $4-$way partitioning (by using either the GMP method or the ILP method), see Appendix A. We observe that the communication volume determined by the RB method, CV_{RB} is equal to the optimal communication volume CV_{Opt} for more than half of the set of 100 matrices, see also Figure 6.6. Furthermore, the difference between the optimal communication volume and the communication volume computed by RB is at most 3. Therefore, the absolute difference between CV_{RB} and CV_{Opt} is small.

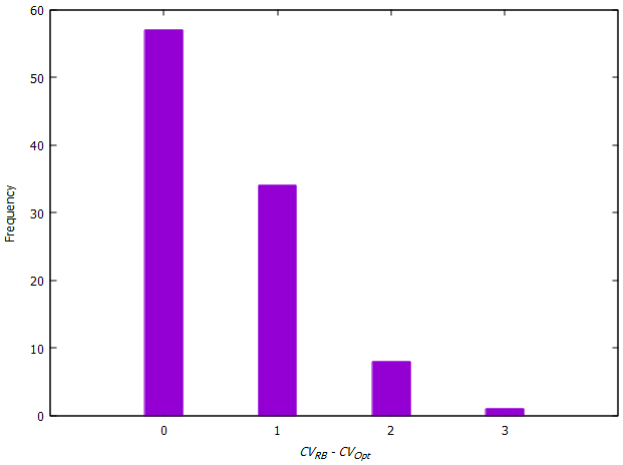


Figure 6.6: Absolute difference between the communication volume determined by the recursive bipartitioning method, and the optimal communication volume, for a $4-$way partitioning of a matrix. Determined on a set of 100 matrices.

In Figure 6.7, the performance profile plot of the RB method is displayed. The $x-$axis shows the communication volume of the RB method relative to the optimal communication volume. The matrices with optimal communication volume equal to zero were left out. We see that in 97% of the matrices the relative communication volume of recursive bipartitioning is equal to 1.2 at most. We can also observe that the relative communication volume of the RB method is equal to 1.67 at most. We remark that the computation of a $4-$way partitioning using the recursive bipartitioning method took less than one second for all but one matrix. This is in general much faster than computing the optimal $4-$way partitioning using an exact method. Therefore, we can conclude that the recursive bipartitioning method computes $4-$way partitionings with communication volume not far from the optimal communication volume in a fraction of the time it takes an exact algorithm to compute the optimal communication volume.

The communication volumes determined by the recursive bipartitioning method can be found in Appendix B.

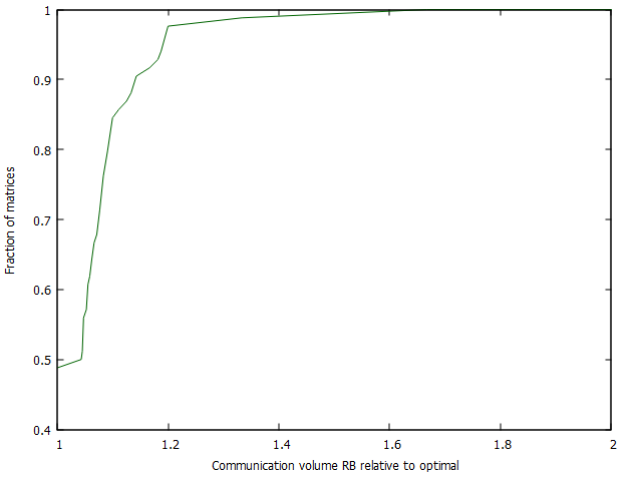


Figure 6.7: Performance profile plot comparing the communication volume determined by the recursive bipartitioning method CV_{RB} with the actual optimal communication volume CV_{Opt} of the 4−way partitioning.

7 | Conclusion

In this thesis we developed a sparse matrix partitioning algorithm, which works for general $p \in \mathbb{N}_{\geq 2}$. Given a matrix, load imbalance parameter ϵ and a value p, the algorithm computes an optimal $p-$way partitioning that satisfies the load balance constraint. That is, it determines the $p-$way partitioning with the lowest communication volume of all possible $p-$way partitionings for that specific matrix and value of ϵ. The algorithm uses a branch and bound method, and branches on the assignment of the rows/columns to a subset of the p processors. This algorithm is called the general matrix partitioner "GMP". We have observed that the row-column based branch and bound method is faster than a nonzero based branch and bound method.

We have also formulated an ILP model based on fine-grain hypergraph partitioning, and used this model to solve the sparse matrix partitioning problem. This method also works for general $p \in \mathbb{N}_{\geq 2}$. We have implemented the ILP model using the software package CPLEX, specifically version 20.1.0 [1].

For $p = 2$, we compared both "general" models with the matrix bipartitioners "MondriaanOpt" [19] and "MatrixPartitioner MP" [12]. For the optimal bipartitioning of a matrix, we determined that the specialized algorithm "MatrixPartitioner MP" is the fastest. We saw that the matrix bipartitioner "MatrixPartitioner MP" outperformed the other three methods, and that it is approximately 12 times faster than the second fastest method ("MondriaanOpt"). Both the "MondriaanOpt" and the "GMP" method were not able to find the optimal solution for all matrices in our test set of 160 matrices.

While the "GMP" method has access to stronger lower bounds than the "MondriaanOpt" method, we found that the "GMP" method is on average 13 times slower than the "MondriaanOpt" algorithm. We expect that there is a lot of performance loss due to the fact that the "GMP" method works for all $p \in \mathbb{N}_{\geq 2}$.

Furthermore, we compared the performance of the "ILP method" and "GMP" method while determining optimal $3, 4-$way partitionings. The "ILP method" outperformed the "GMP" method: it is faster in general, specifically around 70 times faster in the case of $p = 3$, and around 182 times faster in the case of $p = 4$. Additionally, the "ILP method" was able to find the optimal $p-$way partitioning for a higher number of matrices than the "GMP" method. However, we also saw that the "ILP method" runs into memory constraints, see section 6.4 for further information on this topic.

We used the results obtained with regards to optimal $4-$way partitionings, in order to examine the performance of the recursive bipartitioning method. We saw that at least for $p = 4$, the recursive bipartitioning method finds the optimal communication volume for more than half of the matrices. In 97% of the matrices, the communication volume determined by the recursive bipartitioning method is no more than 1.2 times higher than the actual optimal communication volume. Additionally, the recursive bipartitioning method uses only a fraction of time when compared to the run time of the "ILP method" and the "GMP" method.

Therefore, we can conclude that the ILP method is better than the branch and bound based "GMP" method, for general $p \in \mathbb{N}_{\geq 2}$. Furthermore, recursive biparti-

tioning performs very well, as it determines 4−way partitionings close to the optimal communication value.

7.1 Future research

We expect that there is a lot of performance loss due to making the GMP algorithm work for all $p \in \mathbb{N}_{\geq 2}$. It would be worth investigating whether or not a matrix partitioner created purely for the purpose of finding 3−way partitionings based on the branch and bound method used in this thesis, and [19, 12], would outperform the ILP method. We base this on the fact that, for $p = 2$, a specialized matrix bipartitioner, based on the branch and bound method, is the fastest, and thus outperforms the ILP method.

Furthermore, the ILP method itself could stand to be improved. For instance, we only eliminated part of the symmetric solutions in the set of feasible solutions. We saw that although the ILP method is much faster than the other methods, it also suffered from memory issues which arose during its iterations. Further research could focus on finding potential solutions to these memory issues, and checking whether these solutions improve (decrease) its total running time.

A | Results $p = 3,\ 4$

Table A.1: Results for 114 matrices with less than 1000 nonzeros of the SuiteSparse Matrix Collection in the case that $p = 3$. This table shows the matrix name, the number of rows and columns and the number of nonzeros in the matrix. The fourth column shows the communication volume of the optimal 3−way partitioning. The computation times of the row-column based branch and bound method ("GMP") is given in column (6). Column (7) shows the computation time of the ILP method. A bar ('-') signifies that the method in question was not able to compute the optimal communication volume in the allotted time, see section 6 for the specifics. The exponents ^{2} respectively ^{R} mean that the B & B method used `order 2` respectively reversed the order of the subtrees in order to compute the optimal communication volume, see section 6.1.

Name	m	n	nz	CV	Computation time (s)	
					B & B	ILP
Trec4	2	3	3	2	0.012	0.003
GL7d10	1	60	8	2	0.023	0.009
mycielskian3	5	5	10	3	0.03	0.014
Trec5	3	7	12	4	0.06	0.017
b1_ss	7	7	15	4	0.168	0.034
ch3-3-b2	6	18	18	0	0.02	0.009
rel3	12	5	18	6	0.26	0.194
cage3	5	5	19	7	0.438	0.14
lpi_galenet	8	14	22	3	0.046	0.114
relat3	12	5	24	8	9.593	0.163
lpi_itest2	9	13	26	4	0.058	0.143
lpi_itest6	11	17	29	3	0.063	0.037
Tina_AskCal	11	11	29	6	0.635	0.18
n3c4-b1	15	6	30	6	0.808	0.057
n3c4-b4	6	15	30	6	0.82	0.061
ch3-3-b1	18	9	36	6	1.36	0.124
Tina_AskCog	11	11	36	6	0.495	0.118
GD01_b	18	18	37	2	0.059	0.102
Trec6	6	15	40	8	6.625	0.267
mycielskian4	11	11	40	10	121.604	0.448
Tina_DisCal	11	11	41	9	14.608	0.367
farm	7	17	41	7	164.394	0.291
kleemin	8	16	44	8	12.733	1.074
LFAT5	14	14	46	4	0.089	0.074
bcsstm01	48	48	48	0	0.088	0.009

Tina_DisCog	11	11	48	9	31.254	0.377
cage4	9	9	49	12	169.169	3.493
jgl009	9	9	50	10	22.28	0.467
GD98_a	38	38	50	3	0.082	0.072
GD95_a	36	36	57	1	0.076	0.163
n3c4-b2	20	15	60	15	124625	39.487
klein-b1	30	10	60	8	21.472	0.514
klein-b2	20	30	60	9	2695.69	0.846
n3c4-b3	15	20	60	15	121976	29.668
Ragusa18	23	23	64	9	50.689	1.825
bcsstm02	66	66	66	0	0.136	0.01
lpi_bgprtr	20	40	70	6	10.629	0.241
wheel_3_1	21	25	74	13	41245.5	12.853
jgl011	11	11	76	11	19.7	0.641
rgg010	10	10	76	12	873.374	1.085
Ragusa16	24	24	81	12	647.004	10.738
LF10	18	18	82	8	12.603	0.753
problem	12	46	86	5	1.515	0.333
GD02_a	23	23	87	12	3012.99	12.821
n3c5-b1	45	10	90	10	23731.7	0.452
Stranke94	10	10	90	18	226007	94.873
GD95_b	73	73	96	2	3.236	0.121
ch4-4-b3	24	96	96	0	0.108	0.014
Hamrle1	32	32	98	10	3470.89	1.447
lp_afiro	27	51	102	7	39.881	0.3
rel4	66	12	104	8	170.756	0.309
bcsstm03	112	112	112	0	0.265	0.011
p0033	15	48	113	9	4376.94^{2}	4.772
football	35	35	118	13	33052.8	23.041
n4c5-b11	10	120	120	2	13.144	0.089
GlossGT	72	72	122	8	9174.98	1.872
wheel_4_1	36	41	122	18	-	1318.3
bcspwr01	39	39	131	8	255.02	0.548
bcsstm04	132	132	132	0	0.319	0.013
p0040	23	63	133	8	11511.7	0.327
GD01_c	33	33	135	11	124.691	17.287
bcsstm22	138	138	138	0	0.337	0.013
lpi_woodinfe	35	89	140	0	0.134	0.062
mycielskian5	23	23	142	20	-	251.686
ch4-4-b1	72	16	144	15	-	216.908
Trec7	11	36	147	13	112406^{R}	23.853
lp_sc50b	50	78	148	9	14521.6	2.047
d_ss	53	53	149	9	12114.6	1.671
GD99_c	105	105	149	1	0.476	0.119
bcsstm05	153	153	153	0	0.372	0.013
refine	29	62	153	6	46.558	0.372
karate	34	34	156	14	31679.6	39.307
can_24	24	24	160	16	16583.1	57.624
lp_sc50a	50	78	160	7	445.517	0.522
bcspwr02	49	49	167	10	7569.91	13.015
lap_25	25	25	169	18	214570	397.509

relat4	66	12	172	9	94891.3	0.988
pores_1	30	30	180	17	271551	480.99
wheel_5_1	57	61	182	19	-	2682.58
GD96_b	111	111	193	4	0.881	0.347
GD98_b	121	121	207	0	0.336	0.058
n2c6-b1	105	15	210	15	-	14.525
n3c6-b1	105	105	210	15	-	14.492
n4c5-b1	105	15	210	15	-	14.638
can_62	62	62	218	10	66372	7.866
dwt_72	72	72	222	8	120081	5.35
divorce	50	9	225	16	-	76.52
GD96_d	180	180	229	0	0.433	0.024
GD02_b	80	80	232	10	-	9.567
cage5	37	37	233	21	-	631.101
Maragal_1	32	14	234	22	-	614.522
d_dyn	87	87	238	10	-	18.191
d_dyn1	87	87	238	10	-	18.037
lpi_forest6	66	131	246	10	62001.2	3.412
Sandi_authors	86	86	248	8	200.49	2.293
ex5	27	27	279	16	13266.3	54.235
will57	57	57	281	10	233919	27.143
dwt_66	66	66	320	8	42382	2.793
n2c6-b10	30	306	330	0	0.467	0.045
olm100	100	100	396	4	11.211	2.851
tub100	100	100	396	8	101683	5.611
odepb400	400	400	399	0	1.6	0.023
gams10a	114	171	407	3	66.339	2.072
gams10am	114	171	407	3	64.554	2.075
bcsstm06	420	420	420	0	1.807	0.032
lp_adlittle	56	138	424	14	173569	110.722
GD00_a	352	352	458	0	3.559	0.225
GD97_c	452	452	460	2	1.283	0.55
bcsstm20	485	485	485	0	2.185	0.037
08blocks	300	300	592	2	8.984^R	6.299
ch5-5-b4	120	600	600	0	1.622	0.086
Sandi_sandi	314	360	613	2	9.886	1.341
n3c6-b11	60	675	720	0	1.435	0.144
bcsstm19	817	817	817	0	5.641	0.065

Table A.2: Results for 100 matrices with less than 1000 nonzeros of the SuiteSparse Matrix Collection in the case that $p = 4$. This table shows the matrix name, the number of rows and columns and the number of nonzeros in the matrix. The fourth column shows the communication volume of the optimal 4−way partitioning. The computation times of the row-column based branch and bound method ("GMP") is given in column (6). Column (7) shows the computation time of the ILP method. A bar ('-') signifies that the method in question was not able to compute the optimal communication volume in the allotted time, see section 6 for the specifics. The exponents ^{2} respectively ^{R} mean that the B & B method used `order 2` respectively reversed the order of the subtrees in order to compute the optimal communication volume, see section 6.1.

Name	m	n	nz	CV	Computation time (s)	
					B & B	ILP
GL7d10	1	60	8	3	0.024	0.009
mycielskian3	5	5	10	4	0.086	0.016
Trec5	3	7	12	7	0.545	0.024
b1_ss	7	7	15	5	0.66	0.085
ch3-3-b2	6	18	18	2	0.2	0.031
rel3	12	5	18	10	77.242	0.321
cage3	5	5	19	9	10.816	0.256
lpi_galenet	8	14	22	4	0.245	0.057
relat3	12	5	24	9	310.945	0.207
lpi_itest2	9	13	26	6	5.681	0.273
lpi_itest6	11	17	29	5	12.101	0.303
Tina_AskCal	11	11	29	7	20.496	0.19
n3c4-b1	15	6	30	9	218.124	0.192
n3c4-b4	6	15	30	9	217.413	0.428
ch3-3-b1	18	9	36	9	377.386	0.713
Tina_AskCog	11	11	36	9	333.198	0.986
GD01_b	18	18	37	3	0.992	0.082
mycielskian4	11	11	40	12	15689.3	7
Trec6	6	15	40	10	4958.19	2.331
farm	7	17	41	10	92685.7	2.333
Tina_DisCal	11	11	41	11	2225.81	3.54
kleemin	8	16	44	11	77736.7	6.411
LFAT5	14	14	46	10	15158.4	3.593
bcsstm01	48	48	48	0	0.077	0.017
Tina_DisCog	11	11	48	13	19462.6	15.067
cage4	9	9	49	16	618941	515.312
GD98_a	38	38	50	4	8.49	0.11
jgl009	9	9	50	14	22901.5	10.524
GD95_a	36	36	57	2	1.475	0.069
klein-b1	30	10	60	12	70646.5	58.908
klein-b2	20	30	60	11	-	7.76
n3c4-b2	20	15	60	18	-	1359.25
n3c4-b3	15	20	60	18	-	1323.72
Ragusa18	23	23	64	12	163856	25.519
bcsstm02	66	66	66	0	0.115	0.009
lpi_bgprtr	20	40	70	8	8255.39	0.804
wheel_3_1	21	25	74	16	-	509.082

jgl011	11	11	76	16	252837	43.377
rgg010	10	10	76	18	-	341.265
Ragusa16	24	24	81	15	-	130.113
LF10	18	18	82	12	14325.3	3.515
problem	12	46	86	6	43.986	0.396
GD02_a	23	23	87	15	-	132.647
Stranke94	10	10	90	20	-	1208.23
n3c5-b1	45	10	90	15	-	147.813
ch4-4-b3	24	96	96	0	0.139	0.015
GD95_b	73	73	96	3	116.541	0.251
Hamrle1	32	32	98	13	-	28.017
lp_afiro	27	51	102	11	330509^R	7.787
rel4	66	12	104	13	-	3.73
bcsstm03	112	112	112	0	0.23	0.025
p0033	15	48	113	12	-	22.165
football	35	35	118	19	-	1148.27
n4c5-b11	10	120	120	2	41.872	0.126
GlossGT	72	72	122	10	-	12.871
wheel_4_1	36	41	122	21	-	36860
bcspwr01	39	39	131	10	150257^R	22.448
bcsstm04	132	132	132	0	0.283	0.02
p0040	23	63	133	13	-	14.041
GD01_c	33	33	135	17	-	7741.03
bcsstm22	138	138	138	0	0.319	0.017
lpi_woodinfe	35	89	140	6	7602.03	10.498
Trec7	11	36	147	20	-	419.1
lp_sc50b	50	78	148	11	-	24.673
GD99_c	105	105	149	2	4391.95	0.274
d_ss	53	53	149	12	-	41.278
bcsstm05	153	153	153	0	0.378	0.017
refine	29	62	153	10	22745.7^R	10.99
karate	34	34	156	18	-	1899.43
can_24	24	24	160	20	-	9538.19
lp_sc50a	50	78	160	11	-	21.2
bcspwr02	49	49	167	14	-	177.436
lap_25	25	25	169	22	-	20971.5
relat4	66	12	172	12	-	4.63
pores_1	30	30	180	22	-	18796.1
GD96_b	111	111	193	7	-	2.098
GD98_b	121	121	207	0	0.391	0.232
n2c6-b1	105	15	210	21	-	46900.7
n3c6-b1	105	105	210	21	-	47661.4
n4c5-b1	105	15	210	21	-	46421.3
can_62	62	62	218	14	-	307.659
dwt_72	72	72	222	12	-	79.271
divorce	50	9	225	23	-	1117.07
GD96_d	180	180	229	0	0.438	0.084
GD02_b	80	80	232	13	-	317.44
d_dyn	87	87	238	15	-	3872.35
d_dyn1	87	87	238	15	-	3871.97
lpi_forest6	66	131	246	12	-	180.623

Sandi_authors	86	86	248	10	169016	32.8
n2c6-b10	30	306	330	4	98603.9	42.636
olm100	100	100	396	8	196704	40.572
odepb400	400	400	399	0	1.865	0.026
bcsstm06	420	420	420	0	1.953	0.027
GD00_a	352	352	458	0	3409.37	0.388
GD97_c	452	452	460	3	4.902	3.057
bcsstm20	485	485	485	0	2.602	0.044
08blocks	300	300	592	0	1.305	0.391
ch5-5-b4	120	600	600	0	2.078	0.449
Sandi_sandi	314	360	613	2	10.16	4.258
bcsstm19	817	817	817	0	6.591	0.078

B | Results CV: exact & RB

Table B.1: Results for 100 matrices in the SuiteSparse Matrix Collectio with less than 1000 nonzeros. This table shows the matrix name, the number of rows and columns and the number of nonzeros in the matrix. Columns (5)-(7) display the optimal communication volume in the case that $p = 2,\ 3$ and 4. Column (8) displays the communication volume obtained by recursively bipartitioning the matrix into 4 parts using the method from section 5. The load imbalance parameter was $\epsilon = 0.03$.

Name	m	n	nz	CV			
				$p = 2$	$p = 3$	$p = 4$	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
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
n2c6-b10	30	306	330	0	0	4	4
olm100	100	100	396	2	4	8	8
odepb400	400	400	399	0	0	0	0
bcsstm06	420	420	420	0	0	0	0
GD00_a	352	352	458	0	0	0	0
GD97_c	452	452	460	1	2	3	3
bcsstm20	485	485	485	0	0	0	0
08blocks	300	300	592	0	2	0	0
ch5-5-b4	120	600	600	0	0	0	0
Sandi_sandi	314	360	613	0	2	2	2
bcsstm19	817	817	817	0	0	0	0

Bibliography

[1] IBM ILOG CPLEX Optimization Studio 20.1.0. URL `https://www.ibm.com/products/ilog-cplex-optimization-studio`.

[2] Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. Engineering a direct *k-way* hypergraph partitioning algorithm. In *Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 28–42. 2017. doi:10.1137/1.9781611974768.3.

[3] R.H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP*. Oxford University Presss, Oxford, 2nd edition, 2020. doi:10.1093/oso/9780198788348.001.0001.

[4] U. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999. doi:10.1109/71.780863.

[5] 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*, pages 1199–1204, 2001. doi:10.1109/IPDPS.2001.925093.

[6] A.E. Caldwell, A.B. Kahng, and I.L. Markov. Optimal partitioners and end-case placers for standard-cell layout. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(11):1304–1313, 2000. doi:10.1109/43.892854.

[7] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1), 2011. ISSN 0098-3500. doi:10.1145/2049662.2049663.

[8] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, 2006. doi:10.1109/IPDPS.2006.1639359.

[9] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In Alfonso Ferreira, José Rolim, Horst Simon, and Shang-Hua Teng, editors, *Solving Irregularly Structured Problems in Parallel*, pages 218–225. Springer Berlin Heidelberg, 1998. doi:10.1007/BFb0018541.

[10] B. Hendrickson and T. G. Kolda. Partitioning Rectangular and Structurally Unsymmetric Sparse Matrices for Parallel Processing. *SIAM Journal on Scientific Computing*, 21(6):2048–2072, 2000. doi:10.1137/S1064827598341475.

[11] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, page 343–348, New York, 1999. ACM Press. doi:10.1145/309847.309954.

[12] T. E. Knigge and R. H. Bisseling. An improved exact algorithm and an NP-completeness proof for sparse matrix bipartitioning. *Parallel Computing*, 96:102640, 2020. doi:10.1016/j.parco.2020.102640.

[13] D. Kucar, S. Areibi, and A. Vannelli. Hypergraph partitioning techniques. *Dynamics of Continuous, Discrete Impulsive Systems. Series A: Mathematical Analysis*, 11, 04 2004.

[14] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley and Sons, Chichester, UK, 1990. doi:10.1007/978-3-322-92106-2.

[15] S. Maleki, U. Agarwal, M. Burtscher, and K. Pingali. Bipart: A parallel and deterministic hypergraph partitioner. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021. doi:10.1145/3437801.3441611.

[16] A. Mumcuyan, B. Usta, K. Kaya, and H. Yenigün. Optimally bipartitioning sparse matrices with reordering and parallelization. *Concurrency and Computation: Practice and Experience*, 30(21):e4687, 2018. doi:10.1002/cpe.4687.

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

[18] D. Pelt. Matrix Partitioning: Optimal bipartitioning and heuristic solutions. *Master thesis, Utrecht University*, 2010.

[19] D. Pelt and R.H. Bisseling. An exact algorithm for sparse matrix bipartitioning. *Journal of Parallel and Distributed Computing*, 85:79 – 90, 2015. doi:10.1016/j.jpdc.2015.06.005.

[20] A. Trifunović and W. J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *Journal of Parallel and Distributed Computing*, 68(5):563–581, 2008. doi:10.1016/j.jpdc.2007.11.002.

[21] B. Usta. Optimal hypergraph partitioning. *Master thesis, Sabanci University*, 2018.

[22] M. van Oort. Accelerating the Mondriaan sparse matrix partitioning package. *Master thesis, Utrecht University*, 2017.

[23] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005. doi:10.1137/S0036144502409019.

[24] M.G. Wrighton and A.M. DeHon. SAT-based optimal hypergraph partitioning with replication. In *Asia and South Pacific Conference on Design Automation, 2006.*, 2006. doi:10.1109/ASPDAC.2006.1594782.