Contents

Chapter 1

Introduction

1.1 Parallel sparse matrix-vector multiplication

Matrices are one of the most important mathematical objects, as they can be used to represent a wide variety of data in many scientific disciplines: they can encode the structure of a graph, define Markov chains with finitely many states, or possibly represent linear combinations of quantum states or also the behaviour of electronic components.

In most real-world computations, the systems considered are usually of large size and involve **sparse** matrices, because the variables at hand are usually connected to a limited number of others (for example, a large graph in which each node has just a handful of incident edges); therefore, the matrices involved have the vast majority of entries equal to 0. More formally, let us consider a matrix of size $m \times n$ with N nonzeros. We say that the matrix is sparse if $N \ll mn$. Without loss of generality, we assume that each row and column has at least one nonzero (otherwise those rows and columns can easily be removed from the problem).

One of the most fundamental operations performed in these real-world computations is the sparse matrix-vector multiplication, in which we compute

$$u := Av, \tag{1.1}$$

where A denotes our $m \times n$ sparse matrix, v denotes a dense vector of length n, and u the resulting vector of length m.

The computation of this quantity following the definition of matrix-vector multiplication, i.e. with the sum

$$u_i = \sum_{j=0}^{n-1} a_{ij} v_j, \quad \text{for } \ 0 \leq i < m,$$

requires $\mathcal{O}(mn)$ operations; this is not efficient with a sparse matrix: if we perform the multiplications only on the nonzero elements, we obtain an algorithm with running time $\mathcal{O}(N)$, and by definition of sparsity, $N \ll mn$.

As mentioned, the systems considered are large, with sparse matrices with thousands (even millions) of rows and columns and millions of nonzeros; for such big instances, even a running time of $\mathcal{O}(N)$ might be non-negligible, especially since sparse matrix-vector multiplications are usually just a part of a bigger iterative algorithm, and need to be performed several times.

It is an important goal then to be able to perform such computations in the least amount of time possible:

however, as there is a natural tradeoff between power consumption and the speed of the processing units [1], it is not feasible to rely only on fast CPUs, but rather focus on parallelism and employ a large number of them with lower processing speed (and, as a result, with fairly low energy requirements).

To describe an efficient way of performing parallel sparse matrix-vector multiplications, we follow the approach described in [2]: before the actual computation takes place, the sparse matrix is distributed among the p processors, creating a **partitioning** of the set of the nonzeros: A is split into $A_0, \ldots, A_{p-1}$ disjoint subsets. Moreover, also the input vector v and the output vector u are distributed among the p processors (note that their distribution might not necessarily be the same, and usually it is not).

Figure 1.1 shows a possible partitioning of a 9×9 matrix with 18 nonzeros. As the actual values of the nonzeros are not important, we only show the sparsity pattern (a colored cell means that there is a nonzero in that position). The two colors denote, respectively, the two resulting subsets of nonzeros.

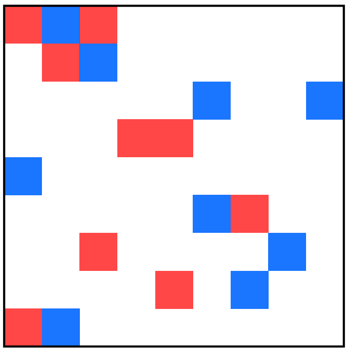


Figure 1.1: Example of a distribution among two processors of a 9×9 matrix with 18 nonzeros. Only the sparsity pattern is shown.

After this distribution, every processor has to compute its local contribution toward the matrix-vector multiplication: to do so, it requires the appropriate vector components which might have been assigned to another processor during the data distribution; if this is the case, communication is required. Once all the required vector components are obtained, the processor starts computing all its local contributions, which are afterwards sent to their appropriate owner, according to the distribution of u. The three phases that describe this process for processor $s = 0, \ldots, p-1$, are summarized in Algorithm 1.1, from [2, 3].

In reality there is also a fourth phase, in which each processor sums up all the contributions received in phase (2) for all of its owned components of u; this is a small sum with negligible computational cost and for this reason it has been omitted from the algorithm.

Note that we assume that all of the nonzero values are represented with the same amount of bits. Doing so, we can focus excusively on the coordinates of the nonzeros, omitting completely their values, as it does not the cost of a parallel sparse matrix-vector multiplication.

Figure 1.2 shows an example of the communication involved in supersteps (0) and (2), for the example partitioning shown in Figure 1.1: the vertical arrows represent the fan-out, while the horizontal arrows represent the fan-in; the color of an arrow indicates which processor is sending data.

As our main interest is to **minimize** the time spent by the parallel machine computing this sparse matrix-vector multiplication, we need to compute explicitly the cost of Algorithm 1.1: we can immediately note that such an algorithm, which follows the Bulk Synchronous Parallel model [4], consists of two communication supersteps separated by a computation superstep.

The time spent by a parallel machine in a computation superstep is exactly the time taken by the processor that finishes last: more formally, the time cost of step (1):

$$T_{(1)} = \max_{0 \le s < p} |A_s|. \tag{1.2}$$

Input: A_s, the local part of the vector v
Output: The local part of the vector u

$I_s := \{i | \exists j : a_{ij} \in A_s\}$
$J_s := \{j | \exists i : a_{ij} \in A_s\}$

(0) ▷ Fan-out

for all $j \in J_s$ **do**
  Get v_j from the processor that owns it.
end for

(1) ▷ Local sparse matrix-vector multiplication

for all $i \in I_s$ **do**
  $u_{is} := 0.$
  for all j such that $a_{ij} \in A_s$ **do**
    $u_{is} = u_{is} + a_{ij} v_j.$
  end for
end for

(2) ▷ Fan-in

for all $i \in I_s$ **do**
  Send u_{is} to the owner of u_i.
end for

Algorithm 1.1: Parallel sparse matrix-vector multiplication.

It is easy to understand that, in order to have efficient parallelization, the computation load has to be distributed evenly. Usually, however, it is not possible to achieve a perfect load balance (e.g. when dividing an odd number of computations among an even number of processors) and we have to reason in terms of an allowed imbalance ε. Consequently, we impose the following hard constraint about the maximum size of the subsets of nonzeros assigned to each processor, according to [2, eq. 4.27]:

$$\max_{0 \leq s < p} |A_s| \leq (1 + \varepsilon) \frac{N}{p}. \tag{1.3}$$

Typical values for the allowed ε in this constraint are 0.03, i.e. a 3% imbalance.

It is reasonable, after all, that the problem of finding an efficient way of performing this computation step boils simply down to a hard constraint for the data distribution. This is because we still have to perform all the multiplications of the form $a_{ij} v_j$, no matter our choice. The communication costs, represented by the first and last supersteps in Algorithm 1.1, are instead the most interesting aspect about maximizing the efficiency of a parallel sparse matrix-vector multiplication algorithm, as there is extreme variability. As a simple example, suppose $p = 2$ and consider the matrix represented in Figure 1.3.

Two possible partitionings of this matrix into two sets are given in Figure 1.4. In Figure 1.4(a) no communication is necessary, whereas in Figure 1.4(b), all of the rows and columns are split, and therefore the maximum possible communication is required during the sparse matrix-vector multiplication algorithm.

Previously, we claimed that the matrix and both the vectors have to be partitioned: in reality it is sufficient to consider only the problem of distributing the nonzeroes, and the partitioning of the vector can be executed according to this: because of the structure of the communication supersteps in Algorithm 1.1, communication is required if and only if the rows/columns of the matrices are *cut*, i.e. assigned to more than one processor.

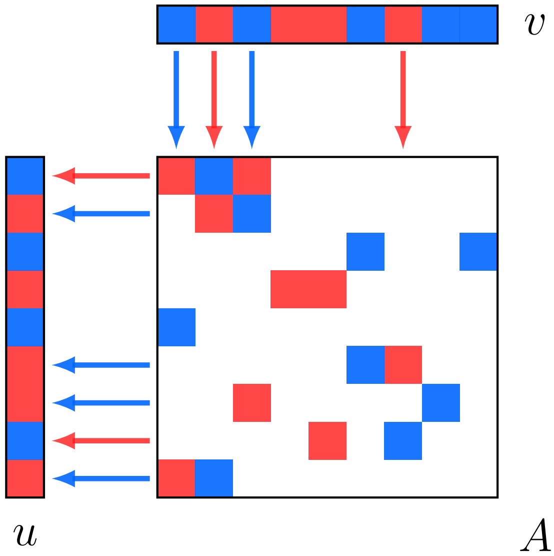


Figure 1.2: Communication for the parallel sparse matrix-vector multiplication with a matrix partitioned as in Figure 1.1. Vertical arrows represent step (0) while horizontal ones represent step (2). The color of an arrow denotes which processor is sending their data for that row/column.

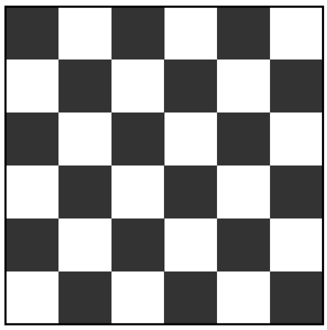
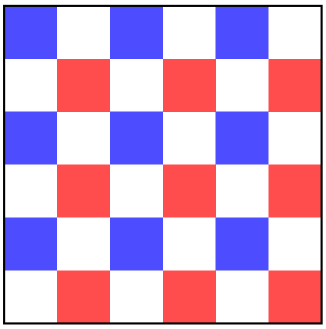


Figure 1.3: Example matrix with checkered sparsity pattern. Black boxes represent the nonzeros.

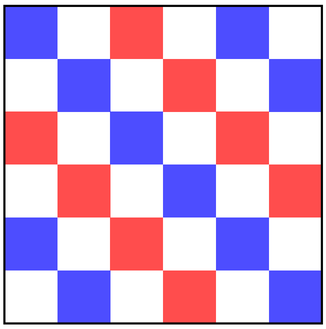
If a full column of our matrix A is assigned to the same processor, we can freely assign the corresponding component of v to the same processor, eliminating completely one source of communication (namely, the fan-out for that column). The same reasoning can be done for the rows. This simplification is possible because imposing a hard constraint similar to (1.3) also to the vector distribution is not helpful, as it only affects the time of linear vector operations outside the matrix-vector multiplication, which are in generally much cheaper [3, Sec. 3].

We can describe more formally the communications cost, following the notation of [3, Def. 2.1]: let $A_0, \ldots, A_{p-1}$ be a p-way (with $p \geq 1$) partitioning of the sparse matrix A of size $m \times n$. Let λ_i denote the number of processors which have a nonzero of row i and let μ_j be the number of processors that have a nonzero of column j; note that, because we assumed that all the rows and columns are nonempty, $\lambda_i, \mu_j \geq 1$. Then the total time costs for the communication steps in our Algorithm 1.1 are:

$$\begin{aligned} T_{(0)} &= \sum_{j=0}^{n-1} (\mu_j - 1), \\ T_{(2)} &= \sum_{i=0}^{m-1} (\lambda_i - 1). \end{aligned} \tag{1.4}$$



(a) Rows and columns are not split, therefore there is no need for communication.



(b) Every row and column is split and causes communication during fan-in and fan-out.

Figure 1.4: Different partitionings of the matrix from Figure 1.3. Red and blue squares represent nonzeros assigned to the two different processors.

These costs are quite straightforward: it is reasonable to assume that the owner of the appropriate vector component is one of the processors that have a nonzero in that row/column, and therefore communication is not necessary for that processor. Adding these costs together, we define the **communication volume** V of the considered partitioning as

$$V := V(A_0, \dots, A_{p-1}) = T_{(0)} + T_{(2)} = \sum_{i=0}^{m-1}(\lambda_i - 1) + \sum_{j=0}^{n-1}(\mu_j - 1). \tag{1.5}$$

As we can see, the communication volume V depends entirely on the matrix A and the considered partitioning. Therefore, the problem of minimizing the cost of a matrix-vector multiplication is shifted toward finding an efficient way of distributing the sparse matrix among the available processors, such that our balance constraint (1.3) is satisfied. The following sections and chapters and, ultimately, this whole Master Thesis, are therefore dedicated to it.

1.2 Hypergraph model

The problem of distributing the nonzeros of a matrix in order to minimize the communication volume, or, in short, the matrix partitioning problem, can also be viewed from the graph theory point of view. We recall that a (unweighted, undirected) graph $G = (V, E)$ is a set of vertices (or nodes) V and edges E which connect them.

The graph partitioning problem has been used in the past to model the load balancing in parallel computing: data are represented as vertices, while their connections (the dependencies) are represented with edges. For a more rigorous definition of the graph partitioning problem, we follow the notation given in [5], performing the simplification in which all the edges have unitary weight. Given the graph $G = (V, E)$ we say that $(V_0, \dots, V_{p-1})$ is a p-way partitioning of G if all these subsets are nonempty, mutually disjoint and their union is the whole set of nodes V.

Moreover, we can consider a balance criterion similar to (1.3):

$$\max_{0 \le s < p} |V_s| \le (1 + \varepsilon)\frac{|V|}{p}, \tag{1.6}$$

where ε, similarly as before, represents the allowed imbalance.

Now, given a partition $(V_0, \dots, V_{p-1})$ of the graph G, we say that the edge $e = (i, j)$ is *cut* if $i \in V_k, j \in V_l$, with $k \neq l$; otherwise, it is said to be *uncut*. Previously, we claimed that communication during the parallel matrix-vector multiplication can be avoided if a row/column is uncut, and here the goal is the same: we want to minimize the *cutsize*, i.e. the number of edges cut.

However, despite all the similarities between the matrix partitioning problem and the graph partitioning one, it has been shown [5] that this cut-edge metric is not an accurate representation of the communication volume. Additional criticism [6] comes from the fact that the graph partitioning approach can only handle square symmetric matrices, even though it has been shown to be a good enough approximation in that case [7]. Moreover, it was also shown [8] that these disadvantages hold for all application of graph partitioning in parallel computing, and not only our problem of matrix partitioning for sparse matrix-vector multiplication. An exact way of modeling the matrix partitioning problem is through the concept of hypergraph partitioning [5].

A hypergraph is simply a generalization of a graph: we do not consider edges that connect two nodes, but rather *hyperedges* (or *nets*), which are subsets of nodes. Apart from considering only non-empty hyperedges, note that there is no other restriction on their cardinality.

Hypergraphs, and in particular the hypergraph partitioning problem are already well known in literature: they have a natural application in the designing of integrated circuits (VLSI), in finding efficient storage of large databases on disks, and data mining [9], as well as urban transportation design and study of propositional logic [10].

Because of this extensive application basis, translating our matrix partitioning problem to a hypergraph partitioning problem seems quite convenient, as all the methods already developed can be analyzed and employed also in our case.

Figure 1.5 shows an example of such a hypergraph. Each colored set represents a different hyperedge; we can see that we can have hyperedges which contain only one node.

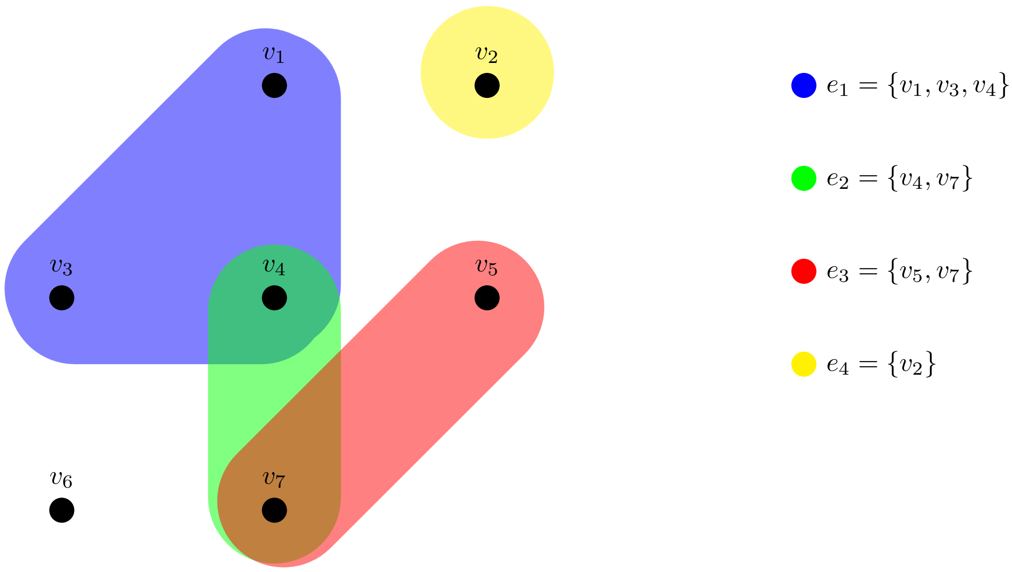


Figure 1.5: Example of a hypergraph with 7 nodes and 4 hyperedges.

The definition of hypergraph partitioning problems is identical to the case of a graph, with the difference that now we do not have cut edges, but cut hyperedges: given the hyperedge $e = \{v_1, \dots, v_k\}$, we say that e is cut if there are i, j such that $v_i \in V_r, v_j \in V_s$, with $r \neq s$, i.e. at least two nodes belong to different sets of the partition. As usually, we want to minimize the cut hyperedges.

If, similarly to (1.2), we define λ_e as the number of different sets the vertices in the hyperedge e are assigned to, the total cost of the partition $(V_0, \dots, V_p)$ is:

$$C = C(V_0, \ldots, V_{p-1}) = \sum_{e \in E} (\lambda_e - 1). \tag{1.7}$$

We can see how closely these equations resemble the ones given in the previous section: it is clear that the hypergraph partitioning problem closely resembles our original matrix partitioning problem.

Note that the hypergraph partitioning problem, along with the graph partitioning problem, are known to be NP-hard [11, Ch. 6].

Now, we will describe three possible models for the decomposition of a sparse matrix into a hypergraph, and discuss their advantages and disadvantages.

In the **column-net** model, our matrix A is represented as a hypergraph for a row-wise decomposition: rows of the matrix are nodes ($V = \{v_1, \ldots, v_m\}$, while columns are hyperedges ($E = \{e_1, \ldots, e_n\}$). We have that the node v_i belongs to the hyperedge e_j (in short $v_i \in e_j$) if and only if $a_{ij} \neq 0$. With this model, the size of the hyperedge e_j is exactly the number of nonzeros in column j, whereas the node v_i belongs exactly to as many hyperedges as there are nonzeros in row i.

As already said, performing a partitioning on the hypergraph consists of assigning each vertex to one of the sets $V_0, \ldots, V_{p-1}$. In this model, this corresponds to assigning a row completely to a processor.

However, as vertices are not exactly nonzeros of our matrix, (1.3) and (1.6) are not exactly equivalent; we need to adjust our balance constraint by introducing a weight for each vertex, as in [2, Def. 4.34]. For $v_i \in V$, we define its weight c_i as

$$c_i := |\{\, j : \; a_{ij} \neq 0\}|,$$

which simply is the number of nonzeros in row i of the matrix A. Note that, following the same notation as in the previous section, we can see the total number of nonzeros N as $N = \sum_{v_i \in V} c_i$.

Our modified balance constraint is as follows:

$$\max_{0 \leq s < p} W(V_s) := \max_{0 \leq s < p} \sum_{v_i \in V_s} c_i \leq (1 + \varepsilon) \frac{N}{p}. \tag{1.8}$$

The **row-net** model is similar to the one just described (as can be guessed from the name): it is exactly the transposed of the colum-net model, in the sense that now rows are hyperedges and columns are vertices of the hypergraph. The reasoning just described applies also to this model, with the little modification that now the weight of a vertex is the number of nonzeros in that column.

We see how the column-net model and row-net model have the advantage of fully assigning a row (or a column) to a processor; this has the advantage of eliminating completely one source of communication in our parallel sparse matrix-vector multiplication algorithm (respectively, the fan-in and fan-out). However, this advantage can easily become a weakness, because now the partitioning is forcedly 1-dimensional, and this is usually too strong of a restriction.

Now, as a last example of possible decomposition of a matrix into a hypergraph, and as a partial address to the drawbacks of the previous two models, we will describe a 2-dimensional approach, the so-called *fine-grain* model [12]. In this model, the N nonzeros are the vertices ($V = \{v_1, \ldots, v_N\}$) and the m rows and n columns are hyperedges ($E = E_r \cup E_c = \{e_1, \ldots, e_m\} \cup \{e_{m+1}, \ldots, e_{m+n}\}$). With this notation, E_r represents the row hyperedges and E_c represents the column hyperedges. The relationship between the vertices and the hyperedges is fairly obvious: $v_k = a_{ij}$ is in both e_i and e_{m+j}.

Now, as the vertices correspond exactly to nonzeros of our matrix, we can use the original equation (1.6) as balance constraint; if we combine this with (1.7), which describes the cost of a hypergraph partition, we can clearly see how this is identical to our original matrix partitioning problem, described by (1.3) and (1.5).

On a higher level, one of the benefits of this decomposition model is easy to understand: we have a lot of freedom and we can assign individually each nonzero to a different partition. Similarly as before, however, this advantage can easily become a drawback because now the size of the hypergraph is consistently larger, with N vertices compared to m and n of the previous two models. Thus, computations on the fine-grain model take substantially more time than row-net or column-net models and therefore there is a restriction on the size of the problem that can be efficiently solved.

1.3 Earlier work

Among the models used to translate matrix partitioning into hypergraph partitioning, we already mentioned row-net and column-net [5], proposed in 1999, and a more recent fine-grained approach [12], proposed in 2001. New models are relatively rare, and recently Pelt and Bisseling proposed the **medium-grain** model [13]. As this model is at the base of our work, a more detailed explanation will be given in Section 1.4.

In addition to these models, there has been some research effort towards the creation of more sofisticated methods, which often comprise several stages and combine different models.

For example, Uçar and Aykanat [14] first employ an elementary 1-dimensional hypergraph model, and then they transform it in several ways to different hypergraph models suitable for both symmetric and unsymmetric matrix partitionings; it is important to note that these models also include the input and output vectors, and therefore a few extra vertices are added to the hypergraph.

A different 2-dimensional approach is given by the *coarse-grain* method [15]: first the column-net hypergraph model is used, obtaining a row partitioning of the matrix in p parts, then a multi-constraint column partitioning in q parts is performed, yielding a final 2-D cartesian partitioning in $p \times q$ parts.

Moreover, Vastenhouw and Bisseling proposed a 2-dimensional recursive method for data distribution [3]; this greedy method splits recursively a rectangular matrix into 2 parts. At each step of the recursion, there is the choice on the direction to be taken in the next step: two different strategies are proposed, alternating splitting directions or simply trying to split both vertically and horizontally and taking greedily the best of the two.

Besides these general purpose models and methods, it is also possible to take into account the structure of the matrix to be partitioned: Hu, Maguire and Blake present in [16] an algorithm for nonsymmetric matrices that performs rows and columns permutations, getting a bordered block diagonal form and then trying to assign matrix rows such that the number of cut columns is minimized.

In general, as there is such a wide variety of different methods and models, it might be difficult to choose the best one, given a matrix to partition. Çatalyürek, Aykanat, and Uçar propose a partitioning recipe [17] that chooses a partitioning method according to some matrix characteristics.

Regarding the actual implementations of the just discussed models, methods and algorithms, there are a few existing software partitioners available. Among the sequential ones we have PaToH (a multilevel Partitioning Tool for Hypergraphs) [18], hMetis [19] (specifically targeted at partitioning hypergraphs for VLSI design), Mondriaan [3] (among the ones here described, this is the one more specifically designed to solve the matrix partitioning problem), MONET (Matrix Ordering for minimal NET-cut)[16]. Zoltan-PHG (Parallel Hypergraph Partitioner) [20] performs instead matrix partitioning in parallel; the relative scarcity of parallel software partitioners is to be explained by the fact that this field is relatively new, and therefore most of the research efforts have been directed toward a sequential approach.

The partitioners just mentioned produce close but slightly different results, with respect to both solution quality and execution time, having at the core the same method for finding good initial solutions. The initial partitioning method employed for small subproblems is the well-known Kernighan-Lin [21] method, with the optimizations of Fiduccia-Mattheyses [22]. This local search heuristic was originally designed for bipartitioning graphs and, given a partitioning that obeys the balance constraint (1.3), it applies a series of small changes to improve the quality of the solution.

To solve large instances, all these partitioners use a multi-level method: the large problem is progressively coarsened until a smaller instance is obtained, then the problem is solved on this small instance and the solution is gradually uncoarsened, with a refinement at each step to improve the solution quality.

Finally, these existing software partitioners are all based on *recursive bisection*: instead of partitioning the hypergraph directly into the desired number of parts, they execute a sequence of bisections of the partitions. This is a good simplification in the sense that it just suffices to find good algorithms for bipartitioning, and also because splitting a hypergraph in just two parts is much easier; there is however one major flaw with this approach: using this recursive bisection we might not be able to reach the same quality of a solution as with direct splitting into the desired number of parts. This flaw is, however, often ignored and direct splitters into more than two parts are rare.

1.4 Medium-grain model

All of the possible ways of translating the matrix partitioning problem into a hypergraph partitioning problem have different advantages and drawbacks: the 1-dimensional ones, row-net and colum-net, eliminate completely one source of communication but are somewhat too restrictive; fine-grain, on the contrary, does not provide any kind of limitation on the choices for the partitioning, but the resulting hypergraph is often too big to manage.

A new model has recently been proposed by Pelt and Bisseling [13], which can be described as a sort of middle ground between the 1-dimensional models and fine-grain model. The resulting partitioning is 2-dimensional by design (thus avoiding the limitations of the row-net and column-net models), but it still imposes that clusters of nonzeros from the same rows and columns are assigned to the same processor, thus reducing the size of the final hypergraph, avoiding the main disadvantage of the fine-grain model.

The key of the medium-grain model lies into the splitting of our original matrix A in two parts, A_r and A_c, such that $A_r + A_c = A$. Then, we proceed to construct the auxiliary block-matrix B, of size $(m + n) \times (m + n)$, defined as

$$B := \begin{bmatrix} I_n & A_r^T \\ A_c & I_m \end{bmatrix}, \tag{1.9}$$

where I_n and I_m denote, respectively, the identity matrices of size n and m. The final hypergraph is finally obtained by applying the row-net model to this matrix B.

Figure 1.6 illustrates this process for a 3×6 rectangular matrix A.

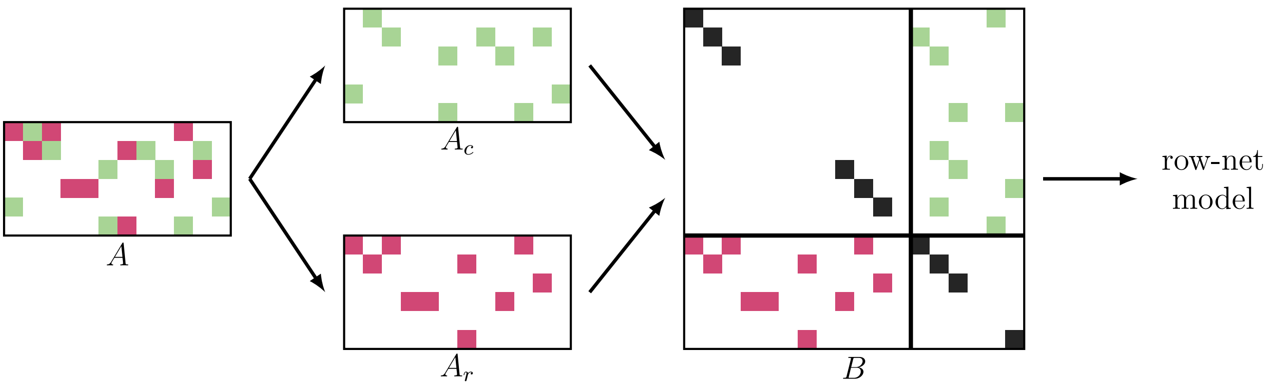


Figure 1.6: Example of the construction of the matrix B from a 6×12 matrix A, for which the sets A_r and A_c were previously established and colored differently. In the resulting matrix, the dummy nonzeros are depicted in black.

After we apply the row-net model and obtain a partitioning of the hypergraph, it is immediate to retrieve a partitioning of our matrix A, as depicted in Figure 1.7.

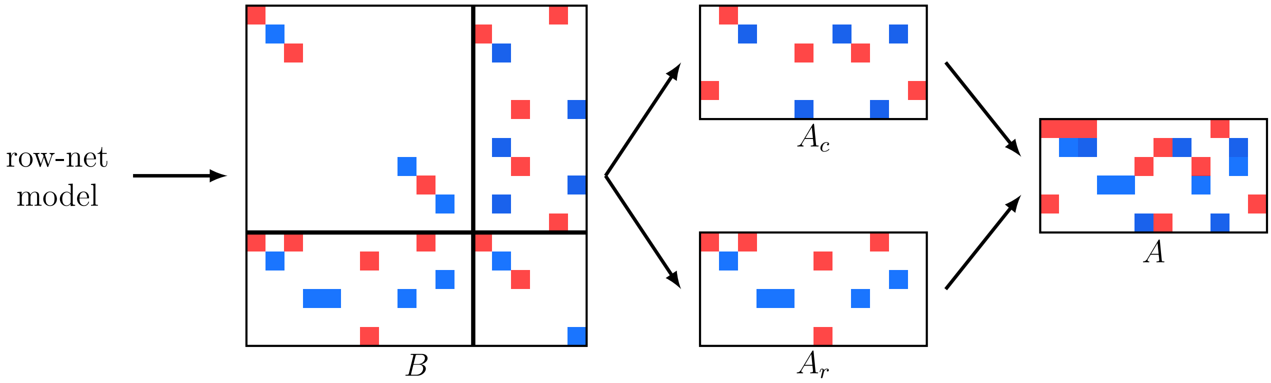


Figure 1.7: Process of obtaining a matrix partitioning starting from a partitioning of the hypergraph following the medium-grain model. In this case $p = 2$.

The usefulness of A_c and A_r is clear if we consider that we use the row-net model. The first is left as-is, while the second is transposed; then, when partitioning 1-dimensionally such that the columns are kept together, we see that we are effectively keeping together elements within the same columns of A_c and A_r^T. The resulting partitioning is fully 2-dimensional, because there are clusters of nonzeros: rows for A_r and columns for A_c (hence the subscripts).

The diagonal elements of B are used only to compute the communication volume. Let us consider the kth column of A; the corresponding nonzeros can be found in the kth column of A_c and in the kth row of A_r^T. If both these parts are nonempty, i.e. the kth column of A was not fully assigned to either A_r or A_c, we need to be careful when we compute the communication volume of a given partitioning: if these parts are to different processors, communication is needed in Algorithm 1.1.

Therefore the diagonal nonzero $B_{k,k}$, assigned by the row-net model to the same processor as the kth column of A_c, that belongs to same row of B as the kth row of A_r, has the purpose of ensuring a correct computation of the communication volume [13, Th. 3.1]. Note that, implementation-wise, there is no need to have the complete diagonal of B: we put a nonzero if and only if the corresponding row of A_r^T and column of A_c are both nonempty.

Experimental results, performed with both the Mondriaan and PaToH packages seem to confirm that this model has indeed some advantages compared to the column-net, row-net and fine-grain models, both regarding partitioning time and solution quality. Because of these good results, it is our goal to investigate further the properties of this model, following two possible directions.

First of all, as the outcome of the medium-grain model depends remarkably on the initial split of A into A_r and A_c, it is interesting to investigate the quality of the algorithm originally proposed in [3] to achieve this initial partitioning; secondly, we will try to develop a fully iterative method that employs the medium-grain model, where a full multi-level partitioning is performed at each iteration and computation time is traded for solution quality.

Both these research directions share an important part: we just need to develop efficient methods to compute from the given matrix A the matrices A_r or A_c required for the medium grain model, either from scratch or starting from an already existing partitioning (later in the work we will talk, respectively, about *partition-oblivious* and *partition-aware* algorithms).

To this extent, Chapters 2 and 3 describe several of these different methods, whereas in Chapter 4 we discuss their implementation and the experimental results for the two mentioned research directions.

Chapter 2

Strategies for splitting a matrix A into A_r and A_c

The main goal of this thesis is to find efficient ways of splitting our original matrix A into A_r and A_c, in order to use the medium-grain model.

We are interested in both improving the initial partitioning of A, and a fully iterative method; therefore, we will make the distinction between methods that don't need an initial partitioning (*partition-oblivious* methods), and are therefore suitable for the first case, and methods that do require an initial partitioning (*partition-aware* methods), to be used in a fully iterative scheme. Most of the time the same algorithm can be used for both purposes, albeit with slight modifications. Before we proceed and analyze the details of the examined heuristics, we can make a few observations, to better understand the general principles behind these algorithms.

If we are interested in an initial partitioning into A_r and A_c that will yield a good communication volume, we already have some information about their quality before the actual partitioning is performed. We can indeed compute an upper bound on the communication cost: if a complete row of A is assigned to A_r (or a full column is assigned to A_c), we are sure that those nonzeros will be assigned to the same processor, and we already discussed in Section 1.1 how this results in no communication for that row (or column). This can give us the idea of trying to keep, as much as possible, full rows and columns together, although it is impossible to do it all the time (because a given nonzero cannot be assigned to both A_r and A_c).

If our purpose is to compute A_r and A_c to improve an existing partitioning, we can follow a few principles to guide us in the choice of what information we should keep, and what we should discard for the next iteration. First of all, it makes sense to have confidence in the existing partitioning: if some nonzeros (for example, a full row or column) are assigned to the same processor, it means that at some point in the previous iteration it was decided that it was convenient to put those nonzeros together, and therefore we should have a preference for them to be together also in the new partitioning. However, this must only serve as an indication and not as a rigid rule, leaving some space for new choices to be made, in order to effectively improve the existing partitioning. Furthermore, we should try to keep, as much as possible, rows and columns together, as noted in the previous paragraph.

2.1 Individual assignment of nonzeros

A simple heuristic that can be used to produce A_r and A_c is a simplification of the algorithm proposed by Pelt and Bisseling along with the medium-grain model [13, Alg. 1], taking as a *score* function the length (i.e. the number of nonzeros) of the given row or column.

The main idea is to assign each nonzero a_{ij} to A_r if row i is shorter than column j (so it has a higher probability of being uncut in a good partitioning), and to A_c otherwise. Ties are broken, similarly as the original algorithm, in a consistent manner: if the matrix is rectangular we give preference to the shorter dimension, otherwise we perform a random choice.

The partition-oblivious version of this heuristic is given in Algorithm 2.1, and it is exactly the same as Algorithm 1 originally proposed. With $nz_r(i)$ we denote the number of nonzeros in row i and with $nz_c(j)$ the number of nonzeros in column j.

Input: sparse matrix A
Output: A_r,A_c

```
if m < n then
    w ← r
else if n < m then
    w ← c
else
    w ← random value c or r
end if
A_r := A_c := ∅
for all a_ij ∈ A do
    if nz_r(i) < nz_c(j) then
        assign a_ij to A_r
    else if nz(j) < nz(i) then
        assign a_ij to A_c
    else
        assign a_ij to A_w
    end if
end for
```

Algorithm 2.1: Partition-oblivious individual assignment of the nonzeros, based on row/column length.

This algorithm can be easily adapted to compute A_r and A_c from a given partitioning of A. Previously we claimed that it is convenient that uncut rows and columns have precedence over cut rows and columns: now, whenever we analyze a nonzero a_{ij} we first look at whether i and j are cut or uncut. If only one of them is cut, we assign the nonzero to the uncut one, otherwise (i.e. both are cut, or both are uncut) we do similarly as before and assign it to the shorter one.

The partition aware variant of this heuristic is given explicitly in Algorithm 2.2.

In Chapter 4, when we are going to perform numerical experiments, we will denote with `po_localview` and `pa_localview`, respectively, the partition-oblivious and the partition-aware variant of this heuristic.

2.2 Assignment of blocks of nonzeros

Instead of assigning nonzeros individually as in Section 2.1, we can take a more coarse-grained approach and try to assign at the same time a greater amount of nonzeros to either A_r or A_c. In particular, we will discuss how to exploit the Separated Block Diagonal (SBD) form of the partitioned matrix A and introduce a further iteration of this concept, discussing the Separated Block Diagonal of order 2 (SBD2) form of the matrix. Moreover, the heuristics described in this section are all partition-aware, and take as input a partitioned matrix.

As throughout this section the permutations of matrices will be fundamental, we adopt a simplified notation: given a vector I with row indices and a vector J with column indices, we denote as $A(I, J)$ the submatrix of A with only the rows in I and only the columns in J (following the order in which they appear in the vectors). With this notation, for example, $A([1, \dots, m], [1 \dots n]) = A$. Furthermore, if I_1 and I_2 are both vectors of indices, with (I_1, I_2) we denote the simple concatenation of these vectors.

```
Input: partitioned sparse matrix A
Output: A_r,A_c

  if m < n then
      w ← r
  else if n < m then
      w ← c
  else
      w ← random value c or r
  end if
  A_r := A_c := ∅
  for all a_ij ∈ A do
      if row i is uncut and column j is cut then
          assign a_ij to A_r
      else if row i is cut and column j is uncut then
          assign a_ij to A_c
      else
          if nz_r(i) < nz_c(j) then
              assign a_ij to A_r
          else if nz(j) < nz(i) then
              assign a_ij to A_c
          else
              assign a_ij to A_w
          end if
      end if
  end for
```

Algorithm 2.2: Partition-aware individual assignment of the nonzeros, based on row/column length.

2.2.1 Using the Separated Block Diagonal form of A

The SBD form of a bipartitioned matrix [23] is defined as follows: given a matrix A whose nonzeros are either assigned to processor 0 or 1, we compute the vectors R_0 and R_2 of the indices of the rows fully assigned, respectively, to processor 0 and processor 1, and the vector R_1 of the indices of the rows partially assigned to both of the processors; similarly, we compute C_0, C_2 and C_1 for the columns. Note that, when creating these vectors, their inner ordering is not important; usually, the ascending order is kept.

Then, we obtain the final index vector for the rows as $I = (R_0, R_1, R_2)$ and for the columns as $J = (C_0, C_1, C_2)$. With these quantities, we can finally compute the SBD form of the matrix A as $A(I, J)$.

An example of the procedure for obtaining this form is shown in Figure 2.1.

More explicitly, if we denote as $m_i := |R_i|$, $n_i := |C_i|$, with $i = 0, 1, 2$, the SBD form is the resulting block matrix:

$$\dot{A} := A(I, J) = \begin{bmatrix} \dot{A}_{00} & \dot{A}_{01} & \\ \dot{A}_{10} & \dot{A}_{11} & \dot{A}_{12} \\ & \dot{A}_{21} & \dot{A}_{22} \end{bmatrix}, \tag{2.1}$$

where

- $\dot{A}_{00}$ of size $m_0 \times n_0$, has nonzeros with uncut rows and uncut columns for processor 0;
- $\dot{A}_{22}$ of size $m_2 \times n_2$, has nonzeros with uncut rows and uncut columns for processor 1;
- $\dot{A}_{01}$ of size $m_0 \times n_1$, has nonzeros with uncut rows for processor 0 and cut columns;

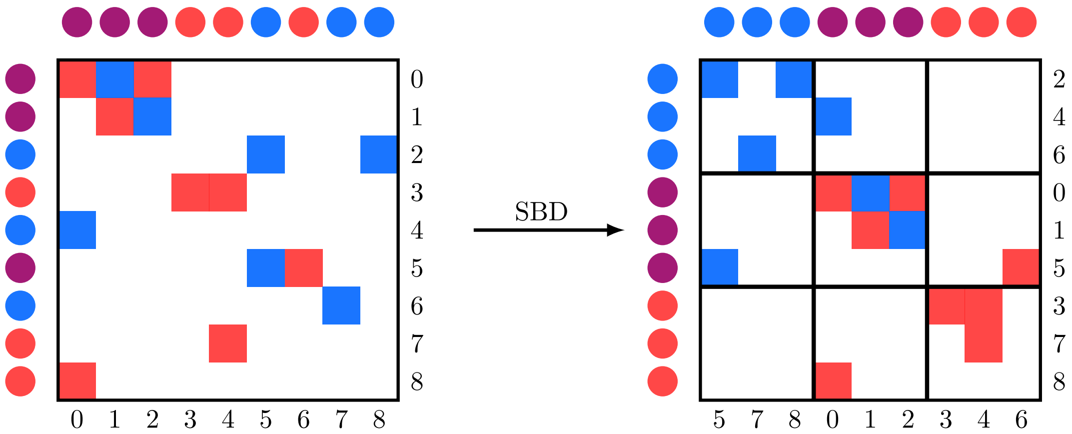


Figure 2.1: Example process to obtain the SBD form of a partitioned matrix. On the left the original matrix is shown, whereas on the right the permuted SBD form. On the top/left sides of the matrices the color of the circle denotes whether that row/column is completely red or blue or it is mixed (purple), whereas on the bottom/right sides the indices of the columns/rows are explicitly given.

- $\dot{A}_{21}$ of size $m_2 \times n_1$, has nonzeros with uncut rows for processor 1 and cut columns;
- $\dot{A}_{10}$ of size $m_1 \times n_0$, has nonzeros with cut rows and uncut columns for processor 0;
- $\dot{A}_{12}$ of size $m_1 \times n_2$, has nonzeros with cut rows and uncut columns for processor 1;
- $\dot{A}_{11}$ of size $m_1 \times n_1$, has nonzeros with cut rows and columns.

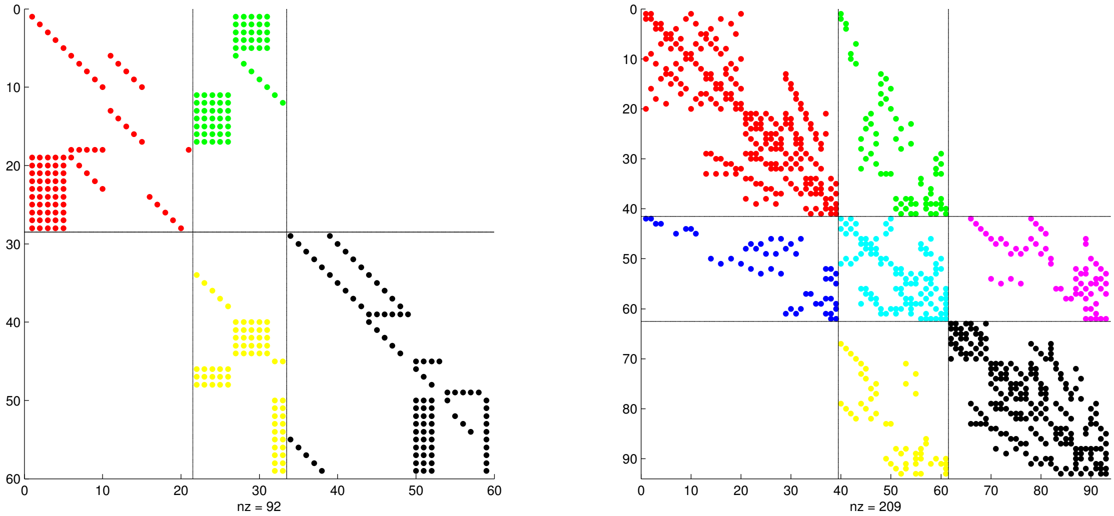
Note that the size of each part along with the number of contained nonzeros can greatly vary, also from matrix to matrix: for example, if the sparsity pattern of the matrix allows a "perfect" partitioning such that there is no communication, all blocks are empty except $\dot{A}_{00}$ and $\dot{A}_{22}$; conversely, if the matrix has a dense (or complicated) pattern and/or the partitioning is far from optimal, such blocks might be almost empty and $\dot{A}_{11}$ will have the majority of nonzeros. An example of the difference of the block sizes of $\dot{A}$ is shown in Figure 2.2.

By computing the Separated Block Diagonal form of a matrix, we are able to explicitly see the underlying structure of the partitioning of a matrix, and the properties of each block can be used to adapt the assignment of its nonzeros. More specifically, the blocks $\dot{A}_{00}$ and $\dot{A}_{22}$ have nonzeros with uncut rows and columns and therefore are more suited to be assigned together; of course, we still have to decide between A_r and A_c and, as mentioned earlier, it is impossible to do both: it is convenient to base our choice on the sizes of such blocks. For example, if $m_0 < n_0$, in the block $\dot{A}_{00}$ the columns are (on average) sparser than the rows: if we assign the nonzeros of this block to A_c we are, in principle, making sure that more rows/columns will stay uncut.

For the blocks with uncut rows and cut columns (namely, $\dot{A}_{01}$ and $\dot{A}_{21}$), the choice is easy: we assign them to A_r and keep their rows uncut. Similarly, we assign the nonzeros of $\dot{A}_{10}$ and $\dot{A}_{12}$ to A_c, keeping their columns uncut.

For the middle block $\dot{A}_{11}$, whose nonzeros have cut rows and cut columns, we cannot exploit any underlying structure: a possible way is to employ one of the other heuristics described in this chapter only considering this submatrix. Our choice is to go with Algorithm 2.1 presented in Section 2.1 (note that we cannot exploit the partition-aware variant of it, because all of the nonzeros in the block considered have cut rows and columns).

The heuristic that employs the SBD structure of a matrix can be explictly visualized in (2.2).



(a) `impcol_b`

(b) `cage6`

Figure 2.2: Example of SBD forms of partitioning of the matrices `impcol_b` and `cage6` [24]. Each part of $\dot{A}$ has been colored differently. In the first matrix there are no cut rows, which means that $\dot{A}_{10} = \dot{A}_{11} = \dot{A}_{12} = \varnothing$.

$$\text{Assignment of } \dot{A} : \begin{bmatrix} R/C & R & \\ C & M & C \\ & R & R/C \end{bmatrix} \tag{2.2}$$

In this matrix, whose structure is the same of (2.1), the letter R in a block denotes that we assign that block to A_r, and similarly for C and A_c. Moreover, R/C stands that the choice between A_r and A_c depends on the block size, whereas M indicates that the block is assigned in a mixed manner, according to Algorithm 2.1.

Note that, as mentioned in Chapter 1, the matrix is usually split by means of recursive bipartitionings: it is then sufficient to keep track of the order of these recursions to have an implicit ordering which can be easily used to compute the SBD form of a matrix [23], instead of computing this form from scratch using the algorithm described in [25, Appendix A].

In Chapter 4, we will refer to the algorithm described in this section as `pa_sbdview`.

2.2.2 Using the Separated Block Diagonal form of order 2 of A

The proposed SBD2 form of a partitioned matrix A is an extension of the SBD form: given a partitioned matrix A, we compute the Separate Block Diagonal form of A of order 2 by separating, in $\dot{A}_{10}$ and $\dot{A}_{12}$ the empty and non-empty columns, and in $\dot{A}_{01}$ and $\dot{A}_{21}$ the empty and non-empty rows. Then all the other blocks, except the central one, are permuted and split up accordingly. This procedure is better shown in Algorithm 2.3.

The resulting final matrix is a block tridiagonal matrix $\ddot{A}$:

Input: partitioned matrix A
Output: $\ddot{A}$
compute $\dot{A}$ as the SBD form of A and obtain also $R_0, R_1, R_2, C_0, C_1, C_2$;
split R_0 in R_{00} and R_{01}, such that $A(R_{00}, C_1) = \varnothing$;
split R_2 in R_{20} and R_{21}, such that $A(R_{21}, C_1) = \varnothing$;
split C_0 in C_{00} and C_{01}, such that $A(R_1, C_{00}) = \varnothing$;
split C_2 in C_{20} and C_{21}, such that $A(R_1, C_{21}) = \varnothing$;
$I := (R_{00}, R_{01}, R_1, R_{20}, R_{21})$;
$J := (C_{00}, C_{01}, C_1, C_{20}, C_{21})$;
$\ddot{A} := A(I, J)$.

Algorithm 2.3: Algorithm to obtain SBD2 form of a matrix A.

$$\ddot{A} := \begin{bmatrix} \ddot{A}_{00} & \ddot{A}_{01} & & & \\ \ddot{A}_{10} & \ddot{A}_{11} & \ddot{A}_{12} & & \\ & \ddot{A}_{21} & \ddot{A}_{22} & \ddot{A}_{23} & \\ & & \ddot{A}_{32} & \ddot{A}_{33} & \ddot{A}_{34} \\ & & & \ddot{A}_{43} & \ddot{A}_{44} \end{bmatrix}, \tag{2.3}$$

where each submatrix $\ddot{A}_{pq}$ is of size $m_p \times n_q$.

Figure 2.3 shows the process of obtaining this matrix $\ddot{A}$ starting from the SBD matrix $\dot{A}$ obtained in Figure 2.1.

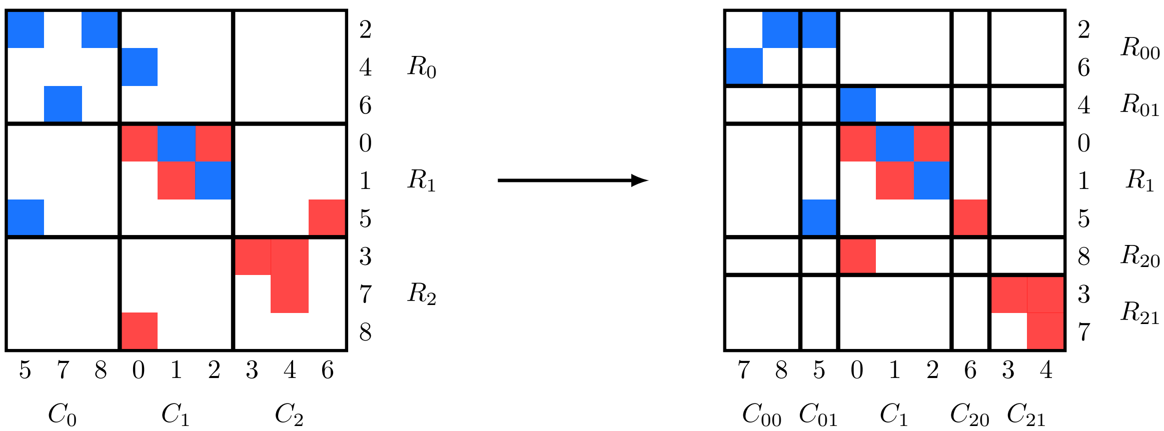


Figure 2.3: SBD2 form obtained starting from the SBD form of Figure 2.1.

To better understand the interesting properties of the newly created parts of the matrix, let us introduce the concept of *neighbor*: given the nonzero a_{ij} we say that a_{kl} is a neighbor if $k = i \vee l = j$; in other words, neighbors of a given nonzero are the ones that lie in the same row or in the same column.

Now, let us consider, for sake of brevity, just the top-left corner of $\ddot{A}$: nonzeros in $\ddot{A}_{00}$ are uncut in the rows and columns and whose neighbors are uncut also in the other, non-shared, dimension. Similarly, nonzeros in $\ddot{A}_{01}$ do not have any neighbor (w.r.t. their row) with cut columns but have neighbors (w.r.t their column) with cut rows. And similarly, with the roles of rows and columns reversed, for $\ddot{A}_{10}$. This exact same reasoning applies also for the bottom-right corner, with the appropriate adaptation of indices.

The size of these parts, and more generally of all of the blocks of $\ddot{A}$, is again highly dependent on the structure of the matrix, as shown in Figure 2.4.

Other than the corner blocks, for which we already argued that the matrix partitioning problem is easy, this structure enables us to assign more specifically nonzeros to either A_r or A_c: it is convenient to assign

$\ddot{A}_{01}$ and $\ddot{A}_{43}$ to A_r, as these nonzeros can be fully assigned to one processor without having the columns cut, and similarly we can assign $\ddot{A}_{10}$ and $\ddot{A}_{34}$ to A_c; for the other blocks, we can repeat the reasoning of the last section.

This heuristic that exploits the SBD2 form of the matrix A is given explicitly as follows:

$$\text{Assignment of } \ddot{A}: \begin{bmatrix} R & R & & & \\ C & R/C & R & & \\ & C & M & C & \\ & & R & R/C & C \\ & & & R & C \end{bmatrix} \tag{2.4}$$

R denotes, as previously, that the block as been assigned to A_r and similarly for C and A_c. R/C denotes that the assignment depends on the block size and M that a mixed assignment is performed with Algorithm 2.1.

Note that, in this case, the SBD2 form has to be computed from scratch from the SBD form, because it uses further information that is not employed during the normal partitioning.

In Chapter 4, we will refer to the algorithm described in this section as `pa_sbd2view`.

2.3 Maximizing empty rows of B

In this section, instead of describing a generating scheme that takes as input the matrix A and produces as output A_r and A_c, we will introduce an improvement scheme, which operates on already existing A_r and A_c and tries to refine them such that the upper bound on the communication volume is lowered.

At the beginning of this chapter, we mentioned how it is convenient to have full rows assigned to A_r and full columns assigned to A_c, in order to avoid communication; a good strategy to produce good A_r and A_c, could then be to maximize such full assignments. The proposed heuristic does essentially this, by trying to swap the assignment of nonzeros from A_r to A_c and viceversa, trying to obtain that full rows are assigned to A_r and full columns are assigned to A_c. In order to achieve both of these goals with a unique algorithm, it is convenient to reason in terms of the matrix B as in (1.9). If we maximize the number of empty rows of B, we are effectively emptying rows of A_r^T (i.e. emptying columns of A_r, therefore fully assigning nonzeros in them to A_c) and of A_c, thus assigning full rows to A_r.

This improvement heuristic falls into the category of *local search* algorithms: we start from a configuration (an assignment of nonzeros to A_r and A_c) and perform a search on the neighborhood, defined as the set of configurations which differ only by the assignment of a single nonzero. By performing this small swap, we can easily fall in a local optimum situation: a few nonzeros (depending on the structure of the matrix) are continuously swapped between A_r and A_c.

We can add a little hill-climbing capability to our heuristic by adding a little buffer: we pre-determine l_{max}, the maximum amount of worsening allowed, and, after this threshold is reached, we start considering only strictly improving solution. In order to have a meaningful threshold, it might be convenient to have it relative to the amount of rows/columns of B, or to its nonzeros. The higher this threshold is, the more capability we have of escaping local optima, but at the cost of slowing down considerably the improvement (even potentially arresting it) of our solution.

For the choice of the neighbor configuration to consider, it is convenient to consider the row of B with a diagonal element (which corresponds to a split row/column of A) with the minimum number of nonzeros: our immediate goal, which in reality spans over a few moves of our local search, is to fully assign the nonzeros of this row of B; we consider the minimum because each time we swap we might slightly worsen the solution.

A more explicit overview on this local search improvement scheme is described in Algorithm 2.4:

As this is a scheme that relies on existing A_r and A_c and aims at improving them, we still need to

Input: A_r, A_c, l_{max}, $iter_{max}$
Output: A'_r, A'_c

```
l := 0
Compute B following the medium-grain model
for it = 1, ..., iter_max do
    i := argmin_{k∈{1,...,m+n} s.t. B(k,k)≠0} nz_r(k)
    for all j ≠ i such that B(i,j) ≠ 0 do
        if B(j,j) ≠ 0 then
            B(i,j) = 0
            B(j,i) = 1
        else
            if l < l_max then
                B(i,j) = 0
                B(j,i) = 1
                l = l + 1
            end if
        end if
    end for
    if nz(B(i)) = 1 then
        B(i,i) = 0
        l = l − 1
    end if
end for
A'_r := B([1, ..., n], [n + 1, ..., m + n])^T
A'_c := B([n + 1, ..., m + n], [1, ..., n])
```

Algorithm 2.4: Local search refinement of A_r and A_c

choose how to generate these parts in the first place. If we can rely on an existing partitioning, a simple choice could be to take as A_r and A_c the subsets of nonzeros assigned, respectively, to processor 0 and 1; otherwise, if our goal is to produce A_r and A_c for the initial partitioning, the simplest choice is to randomly assign each nonzero to either A_r or A_c. These initial solutions, however, are fast to generate but not particularly efficient, and are therefore meaningful only if our improvement scheme is fast enough; otherwise, we can always rely on one of the other heuristics described in this chapter.

2.4 Partial assignment of rows and columns

In Section 2.1 we discussed how to assign each nonzero independently, whereas in Section 2.2 we examined the possibility of exploiting a little the structure of the matrix, in order to assign more nonzeros at once. Keeping this direction, there is some other structure of A that can lead to a better assignment: partial assignment of rows and columns.

The main idea behind this heuristic is that, every time we assign a nonzero to A_r, we know that it is convenient that also all the other nonzeros in the same row are assigned to it; conversely, if a nonzero is assigned to A_c, all the nonzeros in its column should stick with it. Therefore, we ideally want to keep together rows and columns as much as possible; but, as already discussed, there is always the problem that a nonzero cannot be assigned to both A_r and A_c, we can only reason in term of partial assigment of the row/column.

Throughout this section we will stop distinguishing between rows and columns of a matrix and reason in term of **indices** in the set $\{0, \ldots, m+n-1\}$: following the natural ordering, the m rows are mapped to $0, \ldots, m-1$ and the n columns to $m, \ldots, m+n-1$.

This simplification of terms is due to the fact that the core of this heuristic lies in the computation of a **priority vector** v, which is none other than a permutation of the indices $0, \ldots, m+n-1$, where they appear in order of decreasing priority: in this sense, the priority is to be intended as the probability of the nonzeros of that index to be together in a good partitioning.

The assignment of nonzeros is done by "painting" them with an imaginary color, which corresponds either to A_r or A_c: we iterate through our priority vector backwards (i.e. starting from the index with the lowest priority) and assign all of its nonzeros to go together: if the index corresponds to a row, then we assign all of its nonzeros to A_r, otherwise we assign them to A_c. Because each nonzero has both a row and a column, it is represented twice in our priority vector; the second time it is considered, we re-assign it by "painting it over" (hence the name of the algorithm).

A more explicit formulation of this procedure is given in Algorithm 2.5.

```
Input: Priority vector v, matrix A
Output: A_r, A_c
  A_r := A_c := ∅
  for i = m + n − 1, ..., 0 do
    if v_i < m then
      Add the nonzeros of row i to A_r
    else
      Add the nonzeros of column i to A_c
    end if
  end for
```

Algorithm 2.5: Overpainting algorithm

It is possible also to give an alternative formulation for this algorithm in which we iterate forward through v, as described in Algorithm 2.6.

```
Input: Priority vector v, matrix A
Output: A_r, A_c
  A_r := A_c := ∅
  for i = 0, ..., m + n − 1 do
    if v_i < m then
      Add the unmarked nonzeros of the row i of A to A_r
    else
      Add the unmarked nonzeros of the column i of A to A_c
    end if
    Mark nonzeros of index i as "evaluated"
  end for
```

Algorithm 2.6: Alternative formulation of Algorithm 2.5.

In this formulation, every assignment to A_r and A_c is final, but with the added complexity of checking which nonzeros of the considered index are still to be assigned, and only work with them.

Lastly, another different formulation is possible: we consider individually each nonzero a_{ij} and see whether in v $i < j$ (where the $<$ symbol is to be intended as "i precedes j" and not as the comparison of the values) or the other way around; in the first case, the row has more priority and we assign a_{ij} to A_r, otherwise we assign it to A_c. Note that, since we have to perform N lookups on the vector v, this is a more expensive formulation of the same algorithm.

An important point to observe is that this overpainting algorithm is completely deterministic: A_r and A_c are uniquely determined by the ordering of the indices in v. Therefore, the heuristic part of this algorithm lies entirely in the choice of this priority vector, and, for this reason, we will focus on it in the next subsection.

2.4.1 Computation of the priority vector v

Because, with the overpainting algorithm, the quality of A_r and A_c depends entirely on the choice of v, it is important to take a structured approach and explore a wide variety of possibilities for this priority vector.

In this section, we proceed and define several *generating schemes*, and their input determines whether the overpainting algorithm is used to obtain a better initial partitioning or a fully iterative scheme. In general, we try to come up with schemes that can be used for either purpose, with slight modifications.

Each one of the generating schemes can be summarized in three main steps:

1. usage of previous partitioning;
2. sorting;
3. internal ordering of indices.

Now, we give a more detailed explanation of each of those steps.

- **usage of previous partitioning**: if we are considering *partition aware generating schemes*, we separate the set of uncut indices (i.e. indices which correspond to uncut rows or columns) from the cut indices. We consider the simple concatenation of uncut indices and cut indices, in this order, and the next steps are performed on each of these parts. If, instead, we are considering a *partition oblivious scheme*, the subsequent operations are performed on the set $\{0, \ldots, m + n - 1\}$.
- **sorting**: we can either keep the set from the previous step untouched (therefore preserving the natural order of indices) or perform a sorting with respect to the number of nonzeros. The sorting is done in ascending order, as a short row/column is more likely to fit completely in a good partitioning because it does not yield many cut columns/rows.

  In addition, we can refine a bit our sorting: we could move the indices which have only one nonzero to the back, because no matter our assignment of such nonzero, that index will not be cut and it is best to try to keep also the other dimension uncut.
- **internal ordering**: as the last step, we want to finalize our vector v by deciding more precisely the position of each index. The strategies considered, which often depend internally on an additional parameter, are the following:
  - **concatenation**: we put either all the rows before all the columns, or all the columns before all the rows;
  - **mixing**: we can mix rows and columns in two main ways: alternation and spread. Suppose there are twice as many columns as rows: in the first case we get

    $$(c, r, c, r, c, r, \ldots, c, r, c, c, c, \ldots, c, c, c),$$

    whereas with the second one we get

    $$(c, c, r, c, c, r, \ldots, c, c, r),$$

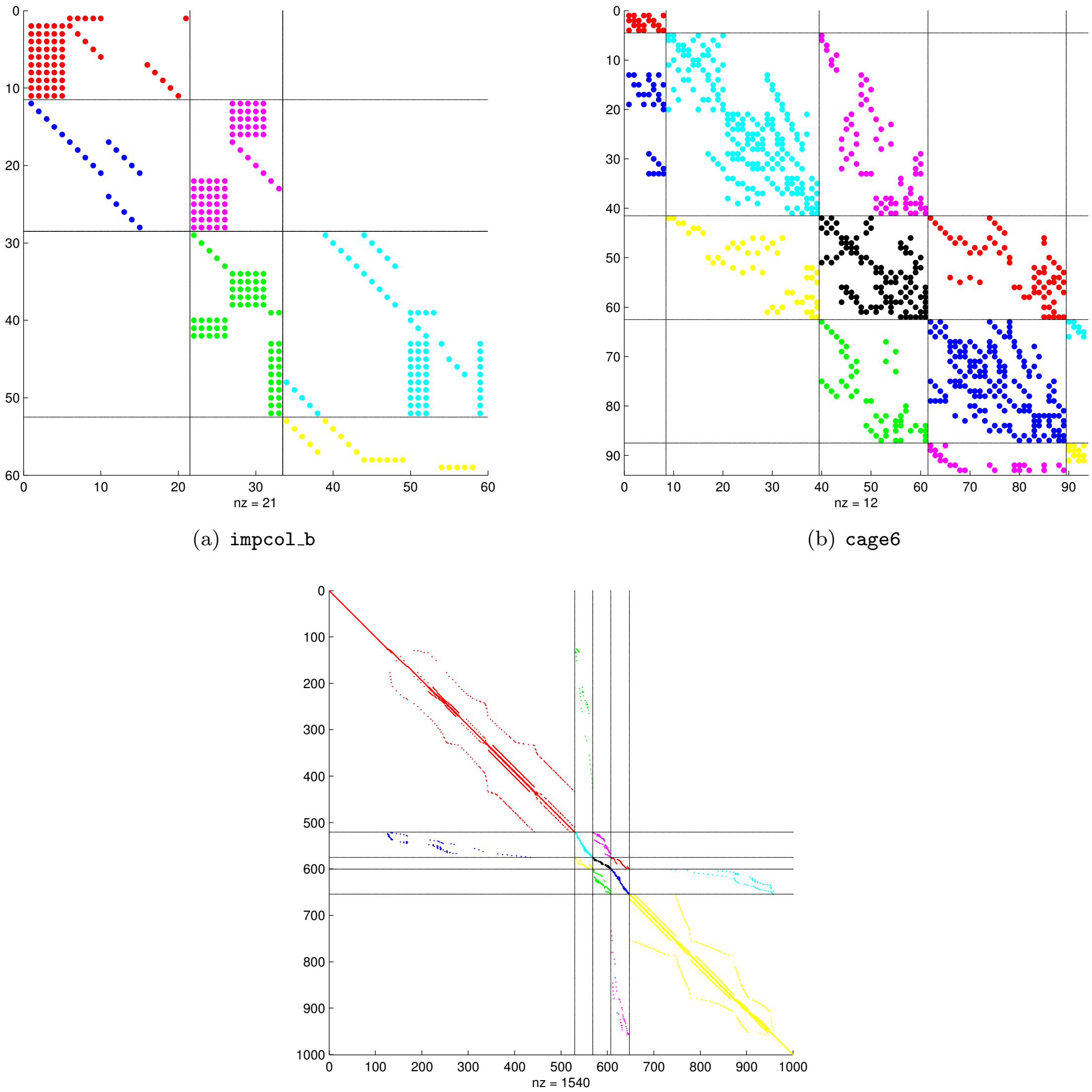
    where with c we denote a generic column and with r a generic row. To obtain a more even distribution, we always start with the greater dimension.
  - **random** (only in case of no sorting): we randomize the ordering of the indices;
  - **simple** (only in case of sorting): we let the sorting decide completely the ordering, and the vector is left as is.

As the complete description of a generating scheme is somewhat lengthy, we use a simplified notation and adopt the following abbreviations:

- `PO`: partition oblivious
- `PA`: partition aware
- `sorted` and `unsorted`: sorting w.r.t. the number of nonzeros is performed or not
- `w` and `nw`: all the indices with only 1 nonzero are moved to the back or not
- `simple`: the sorted vector is left as-is
- `concat`: rows and columns are concatenated
- `row`: the concatenation is done rows-columns
- `col`: the concatenation is done columns-rows
- `mix`: mixing of the rows and columns is enforced
- `alt`: rows and columns are alternated
- `spr`: rows and columns are spread
- `random`: the order of the indices is randomized

With this notation, the name `po_sorted_nw_mix_spr` stands for "partition oblivious generating scheme, with indices sorted by number of nonzeros, without moving the indices with 1 nonzero to the back, with forced mixing of rows and columns, in a spread fashion". This is just one of the many possibilities, which are convenient to visualize using a directed graph, as shown in Figure 2.5; a generating scheme is simply a path from START to END.

Other than this family of heuristics, we can also formulate the problem of partial assignment of rows/columns, always following this framework, in another more mathematical way, to which we dedicate Chapter 3.



(a) `impcol_b` (b) `cage6`

(c) `sherman1`

Figure 2.4: Example of SBD2 forms of three different matrices. Similarly as in Figure 2.2, each part of $\ddot{A}$ has been given a color (note that since there are more parts than coloris used, some colors are repeated even though the parts are not related in any way). We can see in 2.4(a) that the second and fourth columns are empty, and therefore not shown in the image. We can also see the difference in structure between 2.4(b) and 2.4(c): the former one comes from a DNA electrophoresis problem [24], while the latter is an oil reservoir simulation challenge matrix [26]. We can see that with the `sherman1` matrix, the corner parts are predominant because it is a finite element matrix, with a strongly diagonal pattern: it makes sense that most of these nonzeros are "independent" from each other.

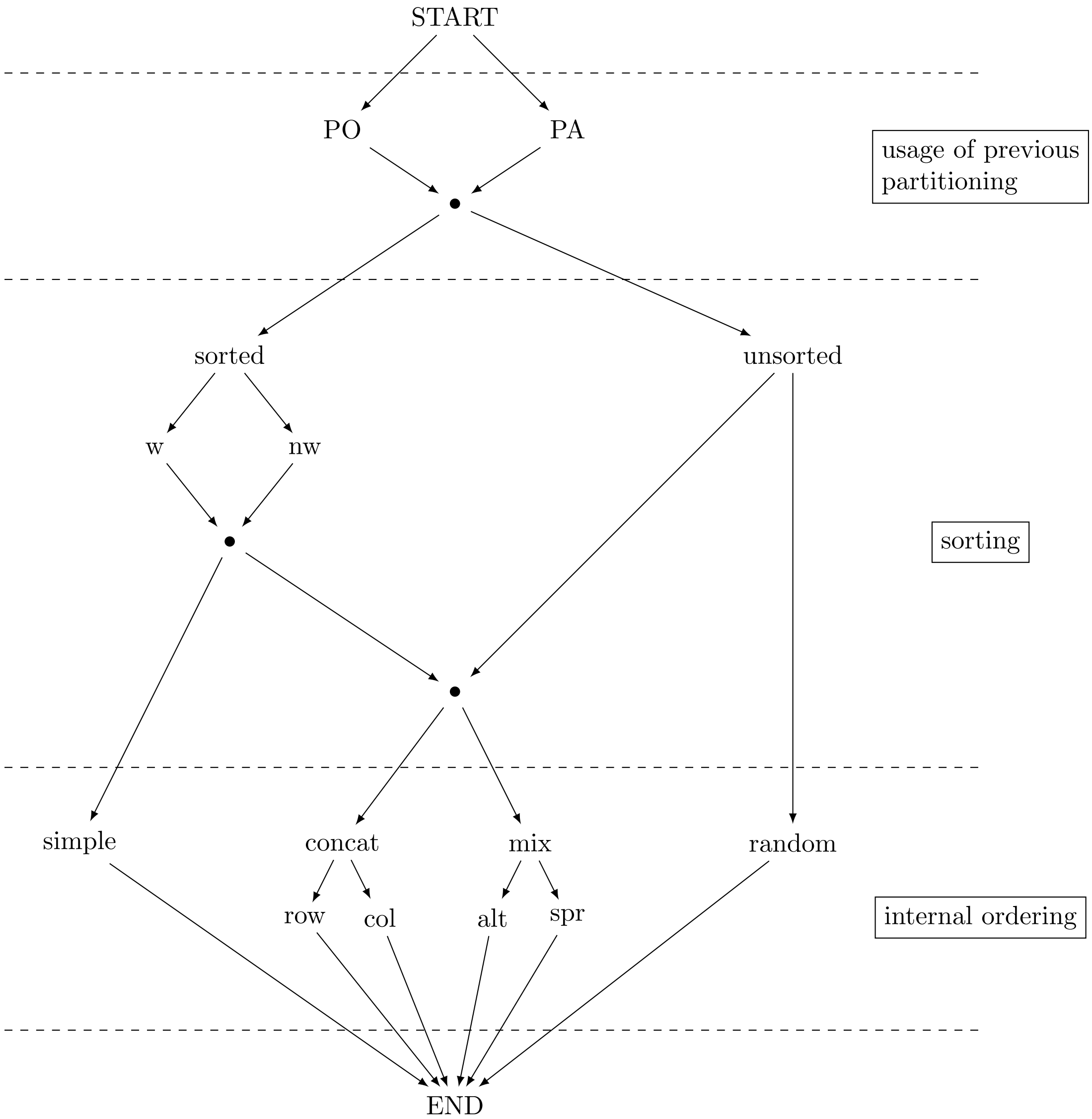


Figure 2.5: Directed graph that represents the family of heuristics used (any path from START to END). Dummy nodes (the ones without any label) were added in order to reduce the number of edges and ease legibility.

Chapter 3

Maximum independent set formulation of the partial row/column assignment problem

With the framework introduced in Section 2.4, we basically translated the problem of the assignment of nonzeros to A_r and A_c (which is already another formulation of the matrix partitioning problem with the medium grain model) to the problem of an efficient computation of a permutation of the indices $\{0, \ldots, m+n-1\}$. In this chapter, we will propose a method for this vector computation problem which relies on concepts of the field of graph theory.

The main idea is somewhat similar to the principle that led us to the development of the Separated Block Diagonal form of order 2 in Section 2.2.2. In that particular form of a partitioned matrix, the blocks $\ddot{A}_{00}$ and $\ddot{A}_{44}$ are interesting, as they contain "independent" nonzeros. More specifically, those rows and columns are fully assigned to a processor, and whose nonzeros do not have any neighbor (a nonzero in the same row or column) which has a cut column/row. The analogue of this concept of independendence, is now to be defined carefully: we want find a subset of the indices $\{0, \ldots, m+n-1\}$ which does not cause any communication, whenever we fully assign its rows to A_r and its columns to A_c. With this definition, our goal is clear: we want to assign as many nonzeros as possible in this way, obtaining a low upper bound on the communication volume, which can be computed during the creation of A_r and A_c.

To do so, we can employ a well studied object in graph theory: the **maximum independent set**. However, this requires a correct translation of our sparse matrix into a graph, described in Section 3.1. In Section 3.2, we delve a little more into the graph theory required and describe the actual algorithm used to compute a maximum independent set in such a graph. In Section 3.3, finally, we give a few different possibilities for computing the priority vector v using the concepts and algorithms just introduced.

3.1 Graph construction

We need to construct the graph correctly from our sparse matrix, in order to retrieve the desired information. In our case, we can simply consider the graph whose adjacency matrix is none other than the sparsity pattern of our matrix A. This exact same formulation has already been studied, for example, by Hendrickson and Kolda [27], who used their *bipartite graph model* to discuss different algorithms for bipartite graph partitioning.

More explicitly, in this graph formulation, rows and columns are vertices, and we have the edge (i, j) if $a_{ij} \neq 0$. It is fairly clear that the resulting graph is bipartite, because an edge connects only a row with a column.

An example of such translation from matrix to graph is shown in Figure 3.1, where we start from the matrix given in Figure 1.1.

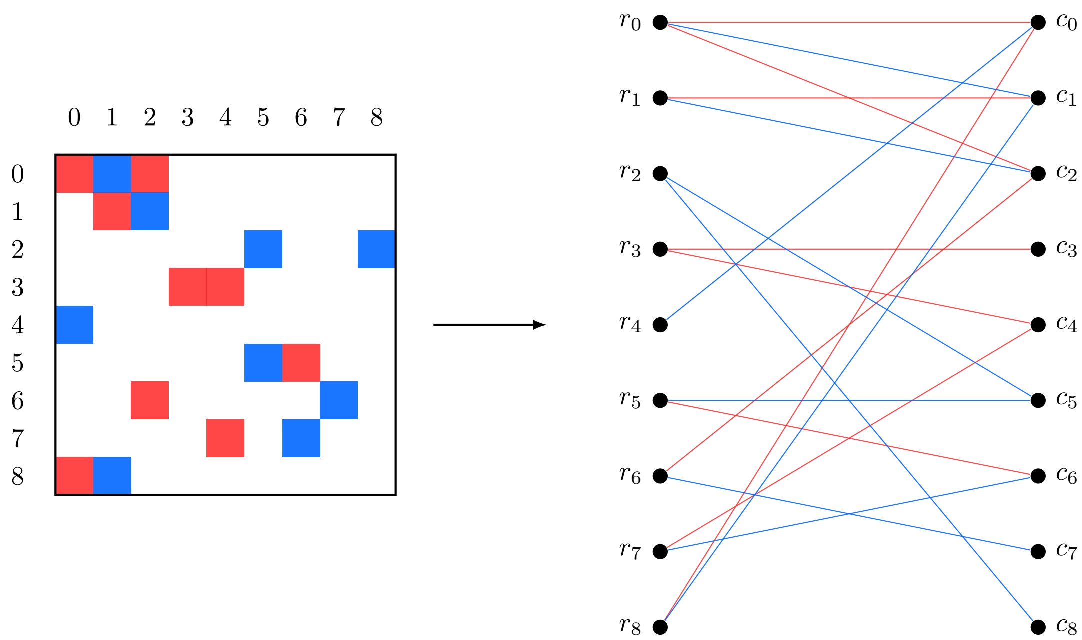


Figure 3.1: Graph constructed using the sparsity pattern of the matrix of Figure 1.1 as adjacency matrix (rows and columns are vertices, nonzeros are edges). The edge color has been kept the same as the corresponding nonzero, but only to facilitate the understanding. The fact that the matrix is partitioned does not play any role in the resulting graph. In the bipartite graph, with r_i we denote row i, whereas with c_j we denote column j.

3.2 The maximum independent set and its computation

In this section, we will give an overview of the maximum independent set problem, discuss its complexity and the relation with other famous problems in graph theory, and, lastly, give an efficient algorithm that can be used with a bipartite graph.

3.2.1 Maximum independent set

The concepts of *independent set* and *vertex cover* are closely related [28]: let $G = (V, E)$ be an undirected graph.

Definition 3.1 (Independent set)**.** *An* independent set *is a subset $V' \subseteq V$ such that $\forall u, v \in V'$, $(u, v) \notin E$. A* maximum *independent set is an independent set of G with maximum cardinality.*

Definition 3.2 (Vertex cover)**.** *A* vertex cover *is a subset $V' \subseteq V$ such that $\forall (u, v) \in E$ we have $u \in V' \vee v \in V'$, i.e. at least one of the endpoints of any edge is in the cover. A* minimum *vertex cover is a vertex cover of G with minimum cardinality.*

A graphical depiction of two independent sets for an example graph is shown in Figure 3.2.

The following lemma explicitly gives us the relation between a vertex cover and an independent set.

Lemma 3.1. *Given a graph G, V' is a vertex cover set if and only if $V \setminus V'$ is an independent set.*

Proof. Let V' be a vertex cover, i.e. $\forall (u, v) \in E$, $u \in V'$ or $v \in V'$. This is equivalent to say that $\forall u, v \in V \setminus V'$, $(u, v) \notin E$, which is the definition of independent set. $\Diamond$

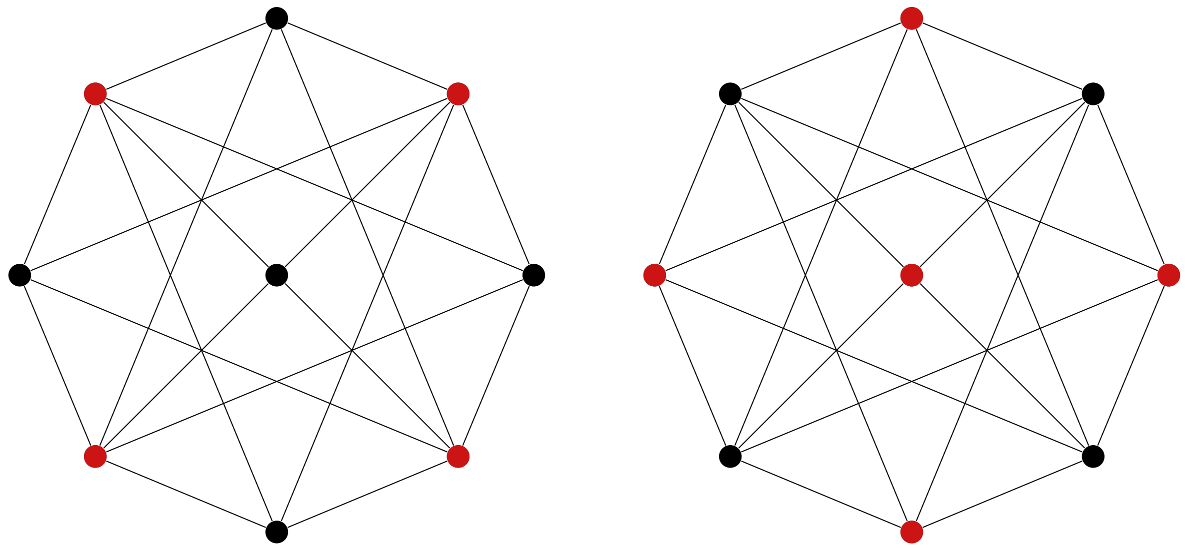


Figure 3.2: Two different independent sets (red vertices) on an example graph. The two independent sets have different cardinality, and the one on the right is a maximum independent set.

As the decision variant of the problem of finding a minimum vertex cover is NP-complete [28, Theorem 3.3], it follows from this lemma that also finding a maximum independent set in a graph is NP-complete; the main consequence of this result is that we cannot solve this problem directly for a generic graph, as it would be as hard as our original matrix partitioning problem. Luckily, we are dealing with a particular kind of graph, a bipartite graph, which simplifies greatly the computations of a maximum independent set.

Before exploiting the bipartiteness of our graph, we need to make an additional observation: Lemma 3.1 states that, in a generic graph, the vertex cover problem and independent set cover are complementary. Therefore, computing a maximum independent set is equivalent to computing a minimum vertex cover. This equivalence is particularly useful in our case, because another mathematical object can be related: the maximum matching.

Definition 3.3 (Matching)**.** *Let $G = (V, E)$ be a graph. A* matching *$M \subseteq E$ is a set of edges such that at most one edge is incident to each vertex $v \in V$. We say that a vertex $v \in V$ is matched by M if an edge in M is incident to v. A* maximum *matching is a matching of maximum cardinality.*

In particular, because we are in a bipartite graph, we can employ Kőnig's Theorem [29]:

Theorem 3.1 (Kőnig)**.** *In a bipartite graph, the size of a maximum matching is equal to the size of a minimum vertex cover.*

We will algorithmically prove the theorem, showing that from a maximum matching we can obtain a minimum vertex cover and their size is equal. First, however, we need two more definitions that are useful when dealing with (maximum) matchings.

Definition 3.4 (Simple path)**.** *Let $G = (V, E)$ be a graph. A path $P = (v_1, \dots, v_k)$ is said to be* simple *if $v_i \neq v_j$, $\forall i \neq j$, i.e. all the vertices are distinct and there are no self-edges or sub-cycles.*

Definition 3.5 (Augmenting path)**.** *Let M be a matching on the graph $G = (V, E)$. The simple path P is said to be* augmenting *if it starts and ends on unmatched (or exposed) vertices, and its edges alternate between $E \setminus M$ and M, in this order.*

It is easy to see that, if we have a matching M and an augmenting path P, if P contains k edges in M, then it has exactly $k+1$ edges in $E \setminus M$, and, if the graph is bipartite, the two endpoints of P belong to the two different sets of vertices. Moreover, $M \oplus P$ is a matching of size $|M| + 1$, where $M \oplus P := (M \setminus P) \cup (P \setminus M)$ denotes the *symmetric difference* between M and P.

Now, in Algorithm 3.1, we give a scheme that constructs a bipartite directed graph, starting from a bipartite undirected graph $G = (L \cup R, E)$ and a matching M.

Input: Bipartite undirected graph $G = (L \cup R, E)$, matching M.
Output: Bipartite directed graph $G' = (L \cup R, D)$.

```
D ← ∅
for (i, j) ∈ E such that i ∈ L and j ∈ R do
    if (i, j) ∈ M then
        D ← D ∪ (j, i)
    else
        D ← D ∪ (i, j)
    end if
end for
```

Algorithm 3.1: Construction of a bipartite directed graph starting from an undirected bipartite graph and a matching.

In other words, the edges in E are given a direction: the ones in the matching go from R to L, and the others from L to R.

In Algorithm 3.2, we give an explicit scheme to compute a minimum vertex cover starting from a maximum matching.

Input: Bipartite graph $G = (L \cup R, E)$, maximum matching M.
Output: Minimum vertex cover C.

```
Construct the modified graph G' = (L ∪ R, D) as in Algorithm 3.1
T ← ∅
for all v ∈ L such that v is not matched do
    Add v to T
    for all u ∈ L ∪ R reachable from v using the directed edges in D do
        Add u to T
    end for
end for
C ← (L \ T) ∪ (R ∩ T)
```

Algorithm 3.2: Construction of the minimum vertex cover in a bipartite graph, starting from the maximum matching.

In Figure 3.3, we can visualize an example of Algorithm 3.2, starting from the graph of Figure 3.1, with a maximum matching. In the final image, the relationships between maximum independent set, minimum vertex cover and maximum matching can be easily recognized.

With the following lemma, we will prove the correctness of Algorithm 3.2 and Kőnig's Theorem.

Lemma 3.2. *Let $G = (L \cup R, E)$ be a bipartite graph and let M be a maximum matching. The subset of vertices C obtained following Algorithm 3.2 is the minimum vertex cover. In addition, we have that $|C| = |M|$.*

Proof. First of all, we will prove that C is a vertex cover: assume it is not, which means that there is an edge $e = (i, j) \in E$ with $i, j \notin C$, which implies that $i \in L \cap T$ and $j \in R \setminus T$, because the graph is bipartite.

We now have two possibilities: either $e \notin M$ or $e \in M$. In the first case, because $i \in T$ and e can be traversed (it goes from L to R in the directed graph), j can be reached and thus $j \in T$, a contradiction. In the second case, we have that i is matched and belongs to T: this means that it was reached by traversing the matched edge, which implies that $j \in T$, again a contradiction. Since in both cases we get to a contradiction, C is indeed a vertex cover of G.

Now, we will prove that $|C| \leq |M|$, by showing that every vertex in C is matched. It is clear that every vertex in $L \setminus T$ is matched, by definition of T. Suppose there is a unmatched vertex $v \in R \cap T$: since

$v \in T$, it means that it was reached from a unmatched vertex u in L, thus the path from u to v is augmenting, which would imply that the matching M is not of maximum cardinality, a contradiction. Moreover, note that there are no edges in the matching between $L \setminus T$ and $R \cap T$ (otherwise, as shown previously, the endpoint in L would also be in T).

Therefore we have that all the vertices in C are matched and the edges of the matching are distinct, which implies that $|C| \leq |M|$. Furthermore, for any matching M' and vertex cover C' is true that $|M'| \leq |C'|$ as there is at least one endpoint in C' for every edge in M'. So we have that $|C| = |M|$, proving Theorem 3.1.

Lastly, these inequality also imply that C is the vertex cover of minimum cardinality: assume it is not, i.e. we have that C' is a vertex cover with $|C'| < |C|$. Now, if we consider the maximum matching M, the inequalities give us that $|M| \leq |C'| < |C| \leq |M|$, a contradiction. C is then the minimum vertex cover.

◊

We have shown that there are close relationships between the maximum matching, minimum vertex cover and maximum independent set. Now, the problem is shifted toward finding an efficient way of computing the maximum matching. In the next section we will describe the Hopcroft-Karp algorithm, which is a simple extension of Algorithm 3.2.

3.2.2 The Hopcroft-Karp algorithm for bipartite matching

The Hopcroft-Karp algorithm [30], devised in 1973, is an efficient scheme for finding a maximum independent set on bipartite graphs, with a running time of $\mathcal{O}\left(|E|\sqrt{|V|}\right)$. This is a considerable improvement over the famous Ford-Fulkerson algorithm of 1956, which, for bipartite graphs, has a running time of $\mathcal{O}(|V||E|)$. We can do a comparison between these two algorithms, even though the latter is technically meant for the maximum flow problems, because a bipartite graph can be modified in such a way that a maximum flows corresponds to a maximum matching in the original graph.

Both algorithms relies on the concepts of augmenting paths, introduced in the previous section in Definition 3.5.

The main idea of the Hopcroft-Karp algorithm is to find these augmenting paths to progressively increase the size of the matching, as outlined in Algorithm 3.3. The fact that in the main loop we augment the matching over several augmenting paths simultaneously gives us the $\sqrt{|V|}$ factor in the running time, instead of a simple $|V|$.

Input: Bipartite graph $G = (L \cup R, E)$
Output: Maximum matching M
  $M \leftarrow \varnothing$
  repeat
    $l_M \leftarrow$ length of the shortest augmenting path, using the matching M
    $P \leftarrow \{P_1, \ldots, P_k\}$, a maximal set of vertex-disjoint shortest augmenting paths of length l_M
    $M \leftarrow M \oplus (P_1 \cup \cdots \cup P_k)$
  until $P = \varnothing$

Algorithm 3.3: Basic outline of the Hopcroft-Karp algorithm

The core of this algorithm is substantially Algorithm 3.2: instead of starting from a maximum matching M to compute the set T (from which follows immediately a minimum vertex cover C), we construct the matching and the T progressively, as follows:

1. we construct the directed graph as in Algorithm 3.1;
2. we perform a breadth-first search (following the directed edges) starting from the unmatched vertices in L, which terminates when unmatched vertices in R are reached. l_M is the length of these

shortest augmenting paths;

3. the maximal set of vertex-disjoint shortest augmenting paths is computed: we start from an unmatched vertex in R reached in the previous step and perform a depth-first search. Whenever we reach a unmatched vertex in L it means that we found an augmenting path P, because we were following the directed edges constructed in the first step. We then add this path to P and resume with the next depth-first search.

If we use the Hopcroft-Karp algorithm in our sparse graph constructed as in Section 3.1, the running time can be even considerably better than the theoretical one: if there are no particularly dense rows and columns, the graph is far from being strongly connected, as each vertex in the graph has just a handful of edges, resulting in fast search phases.

3.3 Computation of the priority vector v with the maximum independent set

After having translated our matrix into a graph as in Section 3.1 and having computed the maximum independent set as described in Section 3.2, we still have to compute our priority vector v, to be used in the same framework of Section 2.4. Similarly as done for all the methods described in Chapter 2, we will distinguish between partition-oblivious heuristics and partition-aware ones.

Let $I \subseteq \{0, \dots, m+n-1\}$ be a set of indices. Instead of computing the graph starting from the full matrix A, we do it from the submatrix $A(I)$ (i.e. only taking rows and columns in I); next, we compute the maximum independent set on the resulting graph using the Hopcroft-Karp algorithm: if we denote by S_I the indices that correspond to this maximum independent set, we always give to this set a high priority, putting it before the remaining indices of $I \setminus S_I$.

With this in mind, the partition-oblivious version is quite straightforward: we take as $I = \{0, \dots, m+n-1\}$, and simply compute

$$v := (S_I, I \setminus S_I).$$

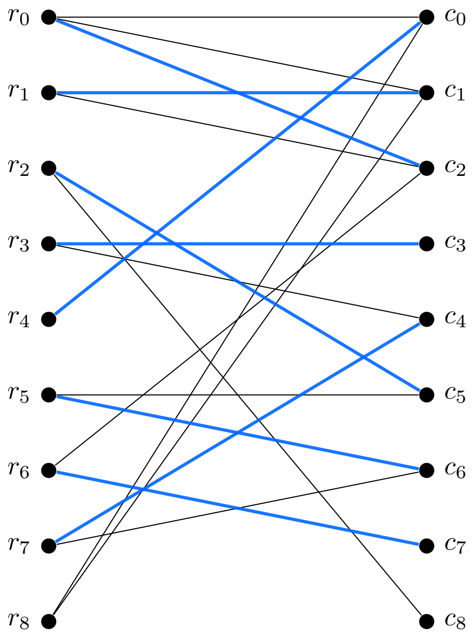
Now, for a partitioned matrix, let U denote the set of uncut indices, and C the set of cut indices. For the partition-aware version of this heuristic we have the following possibilities:

1. we compute S_U and have
$$v := (S_U, U \setminus S_U, C);$$

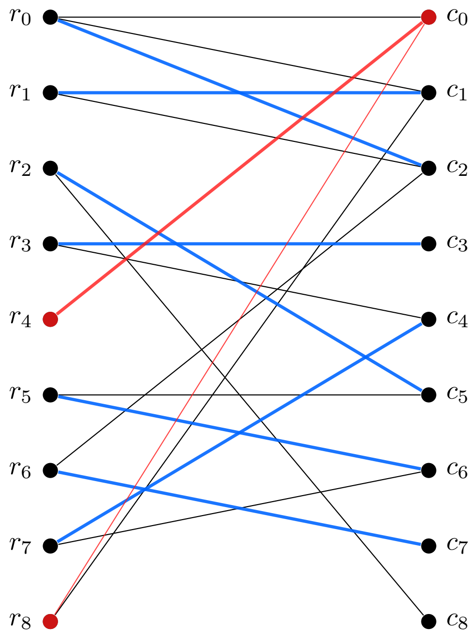
2. we compute S_U, S_C and have
$$v := (S_U, U \setminus S_U, S_C, C \setminus S_C);$$

3. we compute S_U, then we define $U' := U \setminus S_U$ and compute $S_{C \cup U'}$, having
$$v := (S_U, S_{C \cup U'}, (C \cup U') \setminus S_{C \cup U'}).$$

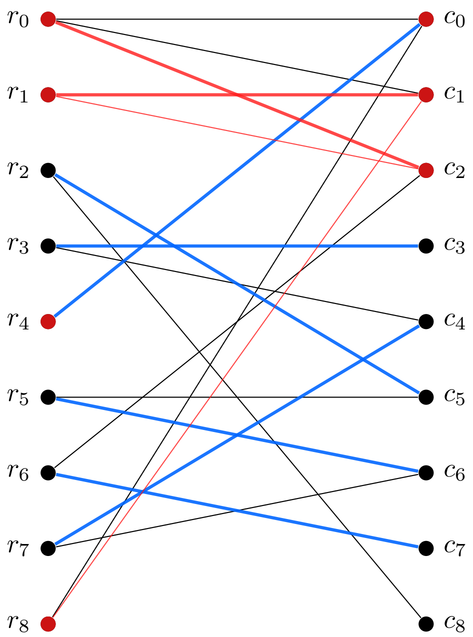
Note that, by construction, we do not expect these three strategies to be radically different in practice: if at the previous iteration the partitioning was done well, U will be quite big, resulting in similar priority vectors.



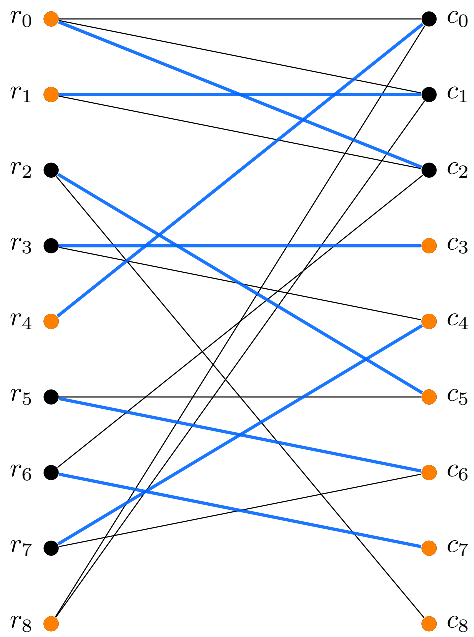
(a) Graph at the beginning of the algorithm. The edges in blue belong to the maximum matching M. $T = \varnothing$.



(b) We start with the unmatched vertex in r_8; we traverse the unmatched edge (r_8, c_0) and the matched edge (c_0, r_4). $T = \{r_8, c_0, r_4\}$.



(c) We traverse the unmatched edge (r_8, c_1), then the matched edge (c_1, r_1), then the unmatched (r_1, c_2) and lastly the matched (c_2, r_0). $T = \{r_8, c_0, r_4, c_1, r_1, c_2, r_0\}$.



(d) We take as $C = (L \cap T) \cup (R \setminus T)$ (depicted in orange). We can see from the final graph that it is indeed a minimum vertex cover.

Figure 3.3: Example of the actions performed in Algorithm 3.2 for the graph of Figure 3.1. The red vertices are in the set T. In the last image the organge vertices belong to a minimum vertex cover. Lemma 3.1 can quickly be checked, as the black vertices are indeed an independent set.

Chapter 4

Implementation and experimental results

In Chapter 2 and 3 we discussed different heuristics aimed at solving the the matrix partitioning problem; whether we try to improve the initial partitioning or perform a fully iterative procedure, we need to translate those ideas into practice, devising an efficient implementation.

In Section 1.3, we mentioned existing software partitioners: Mondriaan [3] is the package of our choice and we defer to it the actual computations of the partitionings, limiting ourselves to create the matrix B of the medium-grain model as in (1.9). The actual algorithm used to construct this matrix is given explicitly in Algorithm 4.1. Note how we consider only the sparsity patterns of A, neglecting completely the values of the nonzeros.

```
Input: A_r, A_c
Output: B
  B ← ∅
  for all a_ij ∈ A_c do                                    ▷ The part relative to A_c
      b_{i+n,j} = 1
  end for
  for all a_ij ∈ A_r do                                    ▷ The part relative to A_r
      b_{j,i+n} = 1
  end for
  for j = 1, ..., n do                                     ▷ Dummy nonzeros for cut columns
      if ∃ i s.t. a_ij ∈ A_r and ∃ i' s.t. a_i'j ∈ A_c then
          b_{i,i} = 1
      end if
  end for
  for i = 1, ..., m do                                     ▷ Dummy nonzeros for cut rows
      if ∃ j s.t. a_ij ∈ A_r and ∃ j s.t. a_ij' ∈ A_c then
          b_{n+i,n+i} = 1
      end if
  end for
```

Algorithm 4.1: Construction of B following the medium-grain model.

Now, having discussed the means of obtaining A_r and A_c and the matrix B, we can outline the general framework used to test the effectiveness of the proposed heuristics. The framework is given explicitly in Algorithm 4.2, and it takes as a parameter the maximum number of iterations allowed, $iter_{max}$.

In Chapter 2 and 3, we distinguished between partition-oblivious and partition-aware methods, and Algorithm 4.2 is suitable for both types of heuristics: even though the framework is naturally suited for developing a fully iterative scheme, if we are after a better initial partitioning for the medium-grain,

Input: Sparse matrix A
Output: Partitioning for the matrix A
  Partition A with Mondriaan using the default options and the medium-grain model
  for $i = 0, \dots, iter_{max}$ **do**
    Use any of the heuristics described previously to compute A_r and A_c
    compute B, using Algorithm 4.1, from A_r and A_c
    Partition B with Mondriaan using the default options and the row-net model
    Re-construct A with the new partitioning
  end for

Algorithm 4.2: General framework for the testing of our heuristics

we can simply neglect the partitioning done in the first step. This is precisely the scope of a partition-oblivious heuristic.

Regarding the actual implementation, we can see from Algorithm 4.2 that Mondriaan is used to perform the actual partitioning and this is the ideal case for its use as a software library. As a consequence, we used C as the main implementation language, even though MATLAB was used for faster prototyping: the flexibility added by managing objects at runtime is ideal when designing algorithms. In order to have C code and MATLAB code interact in the correct way, we took advantage of MEX files [31]. In general, unless preliminary tests showed that the considered heuristic had a remarkably bad quality, we translated back everything to the C language, in order to remove the MEX layer of complexity and get a more efficient implementation. For the Hopcroft-Karp algorithm described in Chapter 3, we used an implementation [32] written in the python programming language, which computes directly the matching on a bipartite graph and the maximum independent set.

The parameter $iter_{max}$ of Algorithm 4.2 is of vital importance for the running time of our iterative method: it is true that our research motivation is to trade computation time for solution quality, but efficiency in doing so is a fundamental goal. Therefore, an iterative scheme that improves the quality of the solution very slowly (thus needing a high $iter_{max}$ to improve as much as possible) is not desirable; fortunately, preliminary tests showed that our heuristics are indeed very fast in this aspect, and only one iteration is needed: additional ones either improve the solution quality by a very small amount (which likely depends on the probabilistic nature of the heuristic and the partitioner) or produce the opposite effect, resulting in a much worse solution. Table 4.1 shows an example of multiple iterations for some of the described heuristic.

Heuristic	**Iterations**										
	0	1	2	3	4	5	6	7	8	9	10
`po_localview`	599	575	576	578	573	583	578	574	575	578	573
`pa_localview`	588	577	580	577	577	574	578	580	576	579	575
`sbdview`	590	588	591	579	597	595	589	603	595	589	587
`po_unsorted_concat`	579	597	594	593	593	597	597	598	594	603	596

Table 4.1: Multiple iterations of four different heuristics for the test matrix `dfl001`. The numbers shown are the rounded arithmetic means of each iteration over 10 repeats of the experiment. Iteration 0 stands for the initial partitioning obtained with the medium-grain model.

In order to perform effective numerical experiments, we need to have a consistent way of testing. First of all, since we are after heuristics suitable for many matrices, it makes sense to have several matrices to test for as described in Section 4.1.

Secondly, since randomness is involved in the partitioner itself and, to a different extent, in some of the considered heuristics, in order to have insightful results we need to perform several measurements and compute an average. After generating an initial partitioning, one iteration of the heuristic was performed independently 5 times, and their result was averaged; this was repeated 20 times, obtaining an average of the 20 initial partitioning and an average of the 20 averages of final results.

Lastly, in order to have a meaningful result that can help us understand whether the given heuristic is globally effective, we will compute the geometric mean of all the initial partitioning and the geometric mean of all the final results. Then we will normalize w.r.t the first value and understand whether the considered heuristic performs better, on average, or not. This normalized geometric mean is denoted, in the next tables, with the symbol ρ.

4.1 Test matrices

In order to have insightful results, we mentioned that the matrices used in the numerical experiments should have different features: in particular we distinguish between (strongly) rectangular matrices and square matrices, and try to have a wide selection w.r.t. the number of nonzeros.

These matrices are mainly from the University of Florida Sparse Matrix Collection [24], and some can additionally be found on the Matrix Market collection [26] and Table 4.2 provides a more thorough description of the matrices used, along with an outline of their basic properties (number of rows m, number of columns n, number of nonzeros N) and their original purpose. Some of these matrices (namely the ones with the † symbol in the table) belong to the 10th Dimacs Implementation Challenge [33], which addressed the graph partitioning and graph clustering problem, and are therefore naturally suited for testing the quality solutions of our algorithms.

Name	m	n	N	**Source problem**
`lpi_ceria3d`	3576	4400	21178	Netlib Linear Programming
`dfl001`	12230	6071	35632	Netlib Linear Programming
`delaunay_n15` †	32768	32768	196548	Delaunay triangulations of random points in plane
`deltaX`	68600	21961	247424	High fillin with exact partial pivoting
`cre_b`	9648	77137	260785	Netlib Linear Programming
`tbdmatlab`	19859	5979	430171	Term-by-document matrix
`nug30`	52260	379350	1567800	Netlib Linear Programming
`coAuthorsCiteseer` †	227320	227320	1628268	Citation and coauthor network
`bcsstk32` †	44609	44609	2014701	Stiffness matrix for automobile chassis
`bcsstk30` †	28924	28924	2043492	Stiffness matrix for off-shore generator platform
`c98a`	56243	56274	2075889	Factorization of composite integers with 98 decimal digits
`wave` †	156317	156317	2118662	3D finite elements
`tbdlinux`	112757	20167	2157675	Term-by-document matrix
`stanford`	281903	281903	2312497	Links between pages in Stanford website
`rgg_n_2_18_s0` †	262144	262144	3094566	Random graph
`polyDFT`	46176	46176	3690048	Polymer self-assembly
`cage13`	445315	445315	7479343	DNA Electrophoresis
`stanford_berkeley`	683446	683446	7583376	Links between Stanford and Berkeley websites

Table 4.2: Matrices used in our experiments.

4.2 Preliminary selection of the best heuristics

Since in Chapter 2 and 3 we discussed many heuristics, which in turn depend on different parameters, it is best to perform a preliminary analysis to quickly figure out which produce the best solutions and should therefore be tested exenstively, and the ones that should not be furtherly considered.

For this reason, a small number of different matrices (with a relatively small amount of nonzeros) has been selected from our choice of Table 4.2: in particular the considered matrices, which have a different structure, are `dfl001`, `tbdlinux`, `nug30`, `rgg_n_2_18_s0`, `bcsstk30`.

The local search heuristic given in Section 2.3, quickly turned out to be far from effective, and therefore we decided to discard it altogether in the numerical experiments.

4.2.1 Partition-oblivious heuristics

Table 4.3 summarizes the results of this preliminary analysis of the partition-oblivious heuristics for the 5 chosen matrices. In the first line, the results of the medium-grain with the algorithm proposed by the authors are given. The value ρ represents the geometric mean of the results for a given heuristics, averaged over all matrices and normalized w.r.t. the default medium-grain.

Heuristic	**Matrix**					ρ
	`dfl001`	`nug_30`	`bcsstk30`	`tbdlinux`	`rgg_n_2_18_s0`	
medium-grain	590	36262	552	8135	910	1.0
`po_localview`	**571**	36665	**598**	8327	1160	1.07
`po_unsorted_concat_row`	1492	189689	653	15081	1098	2.04
`po_unsorted_concat_col`	589	38491	600	24024	**1066**	1.32
`po_unsorted_random`	1314	113070	1127	20154	1093	2.11
`po_unsorted_mix_alt`	1461	181216	715	27942	1104	2.32
`po_unsorted_mix_spr`	1322	81915	759	18584	1122	1.81
`pa_sorted_w_simple`	597	38383	785	8307	1093	1.13
`po_sorted_nw_simple`	606	38674	789	**8301**	1096	1.14
`po_sorted_w_concat_row`	1486	189681	642	15082	1078	2.01
`po_sorted_w_concat_col`	597	38655	621	24045	1068	1.33
`po_sorted_nw_concat_row`	1496	189683	614	15086	1090	2.01
`po_sorted_nw_concat_col`	593	38513	621	24005	1076	1.33
`po_sorted_w_mix_alt`	1317	162549	790	23683	1091	2.19
`po_sorted_w_mix_spr`	641	163013	782	23441	1093	1.88
`po_sorted_nw_mix_alt`	1457	62273	797	15015	1096	1.69
`po_sorted_nw_mix_spr`	719	62402	793	15072	1106	1.47
`po_is`	594	**30655**	615	13286	-	1.12

Table 4.3: Results of the devised partition-oblivious heuristics for the five chosen matrices. In each column, we use the boldface to highlight the best found partitioning for each matrix (not considering the medium-grain value).

In table there is no result for the heuristic `po_is` and the matrix `rgg_n_2_18_s0`, because the maximum number of levels of recursion was reached before the algorithm completed.

4.2.2 Partition-aware heuristics

In Table 4.4 we summarize the results of the partition-aware heuristics. As now we are considering a fully iterative framework, it is best to explicitly give the average of the 20 initial partitionings (iteration 0) alongside the average of the 20 final results (iteration 1). The value ρ represents, similarly as before, the geometric mean of the final results, normalized w.r.t. the average of the initial partitionings for that heuristic.

Similarly as before, the computation of the independent set was not possible in the case of the matrix `rgg_n_2_18_s0` because the maximum levels of recursion was reached before being able to finish the algorithm.

4.2.3 Observations on the preliminary tests

These preliminary tests proved themselves useful, as we could effectively narrow down the best heuristics among the ones proposed; moreover, as a side effect, we are able to make a few general observations regarding desirable and undesirable features of the heuristics.

First of all, we can see that in Table 4.3 the first heuristic, discussed in Section 2.1, is the one that produced the best results among the partition-oblivious ones. This is not surprising, as the algorithm is very similar to the one originally proposed by the authors in [13], even though without a few refinements, which result in our `po_localview` slightly worse than our reference values. The results with the partition-oblivious computation of the independent set was surprisingly good, especially with the matrix `nug_30`.

Secondly, regarding the family of heuristics discussed in Section 2.4, it appears that mixing rows and columns in in general not advisable: for both the partition-oblivious and partition-aware schemes, the results are considerably worse than the reference values.

The same principle can be deducted also from the surprisingly good results of the `unsorted_concat` family of schemes, in which there is a clear separation of rows and columns. For these schemes, it appears that the matrix structure (and in particular, how far it is from being square) heavily influence the quality of the final partitioning. Furthermore, they are mutually exclusive, as expected: if one produces good results, the other one performs terribly. The consistency of their behavior suggests us that they could be merged into a `po_unsorted_concat` and `pa_unsorted_concat` schemes that automatically choose whether having columns-rows or rows-columns, depending on the size of the matrix to be partitioned.

Moreover, it appears that moving or not the indices with one nonzero to the back (denoted, respectively, by `w` and `nw`) does not yield a substantial difference: the results of the first case are usually better, but not as much as expected. In both cases, however, sorting w.r.t the number of nonzeros does not seem to yield a dramatic improvement over the unsorted heuristics.

4.3 Analysis of the performance of the best heuristics through the test matrices

Heuristic	i	**Matrix**					ρ
		dfl001	nug_30	bcsstk30	tbdlinux	rgg_n_2_18_s0	
pa_localview	0	582	36224	537	8051	914	
	1	**575**	36896	577	9934	2189	
pa_sbdview	0	590	36057	546	8112	899	
	1	1493	187241	699	19852	1074	
pa_sbd2view	0	583	36123	542	8018	906	
	1	1276	125009	1150	20757	1055	
pa_unsorted_concat_row	0	584	36512	597	7934	929	
	1	628	42581	641	7337	1088	
pa_unsorted_concat_col	0	589	35945	574	7999	898	
	1	589	38862	602	10087	1069	
pa_unsorted_random	0	591	36390	550	8044	909	
	1	622	36952	589	8383	1094	
pa_unsorted_mix_alt	0	592	36097	536	8019	896	
	1	633	40286	684	9683	1108	
pa_unsorted_mix_spr	0	589	36137	542	8024	905	
	1	642	39703	645	8361	1094	
pa_sorted_w_simple	0	594	36589	544	8012	876	
	1	599	38887	563	8002	1093	
pa_sorted_nw_simple	0	596	36115	542	7995	888	
	1	599	38793	572	8014	1098	
pa_sorted_w_concat_row	0	585	36305	537	8006	898	
	1	1494	190799	661	19036	1087	
pa_sorted_w_concat_col	0	590	36164	537	8032	903	
	1	954	115312	689	25157	1074	
pa_sorted_nw_concat_row	0	590	35955	545	7999	897	
	1	1489	190657	693	19152	1073	
pa_sorted_nw_concat_col	0	597	36436	548	8009	891	
	1	975	116063	707	25543	**1063**	
pa_sorted_w_mix_alt	0	586	36566	549	8006	912	
	1	692	41963	593	9476	1105	
pa_sorted_w_mix_spr	0	593	36085	537	8010	924	
	1	619	39697	571	9060	1078	
pa_sorted_nw_mix_alt	0	588	36509	546	8020	891	
	1	687	42542	**566**	9486	1085	
pa_sorted_nw_mix_spr	0	589	36197	553	8006	886	
	1	655	39201	602	9076	1094	
pa_is_1	0	592	36716	534	7997		
	1	588	**36118**	614	**7323**	-	
pa_is_2	0	596	35972	539	8011		
	1	592	36375	631	8681	-	
pa_is_3	0	590	36432	540	7997		
	1	589	36324	635	7410	-	

Table 4.4: Results of the devised partition-oblivious heuristics for the five chosen matrices. In each column, we use the boldface to highlight the best found partitioning for each matrix.

Bibliography

[1] Jan Rabaey. *Digital integrated circuits : a design perspective.* Prentice Hall, 1996. ISBN: 0131786091 (cited on page 2).

[2] Rob H. Bisseling. *Parallel Scientific Computation: A structured approach using BSP and MPI.* Oxford University Press, 2004 (cited on pages 2, 3, 7).

[3] Brendan Vastenhouw and Rob H. Bisseling. "*A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication*", in: *SIAM Review* 47.1 (2005), pp. 67–95 (cited on pages 2, 4, 8, 10, 31).

[4] Leslie G. Valiant. "*A bridging model for parallel computation*", in: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. ISSN: 0001-0782. DOI: `10.1145/79173.79181`. URL: `http://doi.acm.org/10.1145/79173.79181` (cited on page 2).

[5] Ümit V. Çatalyürek and Cevdet Aykanat. "*Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*", in: *Parallel and Distributed Systems, IEEE Transactions on* 10.7 (1999), pp. 673–693. ISSN: 1045-9219. DOI: `10.1109/71.780863` (cited on pages 5, 6, 8).

[6] Bruce Hendrickson. "Graph partitioning and parallel solvers: Has the emperor no clothes?" In: *Solving Irregularly Structured Problems in Parallel.* Springer, 1998, pp. 218–225 (cited on page 6).

[7] Sivasankaran Rajamanickam and Erik G. Boman. "Parallel partitioning with Zoltan: Is hypergraph partitioning worth it?" In: *Graph Partitioning and Graph Clustering.* 2012, pp. 37–52 (cited on page 6).

[8] Bruce Hendrickson and Tamara G. Kolda. "*Graph partitioning models for parallel computing*", in: *Parallel computing* 26.12 (2000), pp. 1519–1534 (cited on page 6).

[9] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. "*Multilevel hypergraph partitioning: applications in VLSI domain*", in: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 7.1 (1999), pp. 69–79. ISSN: 1063-8210. DOI: `10.1109/92.748202` (cited on page 6).

[10] David A. Papa and Igor L. Markov. "Hypergraph Partitioning and Clustering". In: *Approximation Algorithms and Metaheuristics.* 2007 (cited on page 6).

[11] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout.* New York, NY, USA: John Wiley & Sons, Inc., 1990. ISBN: 0-471-92838-0 (cited on page 7).

[12] Ümit V. Çatalyürek and Cevdet Aykanat. "A fine-grain hypergraph model for 2D decomposition of sparse matrices". In: *Proceedings of the 15th International Parallel and Distributed Processing Symposium.* 2001, pp. 609–625 (cited on pages 7, 8).

[13] Daniël M. Pelt and Rob H. Bisseling. "*A Medium-Grain Model for Fast 2D Bipartitioning of Sparse Matrices*", 2013 (cited on pages 8–11, 35).

[14] Bora Uçar and Cevdet Aykanat. "*Revisiting hypergraph models for sparse matrix partitioning*", in: *SIAM review* 49.4 (2007), pp. 595–603 (cited on page 8).

[15] Ümit V. Çatalyürek and Cevdet Aykanat. "A hypergraph-partitioning approach for coarse-grain decomposition". In: *Supercomputing, ACM/IEEE 2001 Conference.* IEEE. 2001, pp. 42–42 (cited on page 8).

[16] Yifan F. Hu, Kevin C.F. Maguire, and Richard J. Blake. "*A multilevel unsymmetric matrix ordering algorithm for parallel process simulation*", in: *Computers & Chemical Engineering* 23.11 (2000), pp. 1631–1647 (cited on page 8).

[17] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. "*On two-dimensional sparse matrix partitioning: Models, methods, and a recipe*", in: *SIAM Journal on Scientific Computing* 32.2 (2010), pp. 656–683 (cited on page 8).

[18] Ümit V. Çatalyürek and Cevdet Aykanat. "PaToH (Partitioning Tool for Hypergraphs)". In: *Encyclopedia of Parallel Computing.* Springer, 2011, pp. 1479–1487 (cited on page 8).

[19] George Karypis and Vipin Kumar. "*Multilevel k-way hypergraph partitioning*", in: *VLSI design* 11.3 (2000), pp. 285–300 (cited on page 8).

[20] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, and Ümit V. Çatalyürek. "Parallel hypergraph partitioning for scientific computing". In: *Proceedings of the 20th international conference on Parallel and distributed processing.* IPDPS'06. Rhodes Island, Greece: IEEE Computer Society, 2006, pp. 124–124. ISBN: 1-4244-0054-6. URL: `http://dl.acm.org/citation.cfm?id=1898953.1899056` (cited on page 8).

[21] Brian W. Kernighan and Shen Lin. "*An efficient heuristic procedure for partitioning graphs*", in: *Bell system technical journal* 49 (1970), pp. 291–307 (cited on page 8).

[22] Charles M. Fiduccia and Robert M. Mattheyses. "A linear-time heuristic for improving network partitions". In: *Design Automation, 1982. 19th Conference on.* IEEE. 1982, pp. 175–181 (cited on page 8).

[23] Albert-Jan N. Yzelman and Rob H. Bisseling. "*Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*", in: *SIAM Journal on Scientific Computing* 31.4 (2009), pp. 3128–3154 (cited on pages 13, 15).

[24] Tim A. Davis and Yifan Hu. "*The University of Florida Sparse Matrix Collection*", in: *ACM Transactions on Mathematical Software* 38 (2011), pp. 1–25. URL: `http://www.cise.ufl.edu/research/sparse/matrices` (cited on pages 15, 22, 33).

[25] B. O. Fagginger Auer. "GPU Acceleration of Graph Matching, Clustering, and Partitioning". Ph. D. thesis. Utrecht University, 2013 (cited on page 15).

[26] Ronald F. Boisvert, Roldan Pozo, Karin A. Remington, Richard F. Barrett, and Jack Dongarra. "Matrix Market: a web resource for test matrix collections." In: *Quality of Numerical Software.* 1996, pp. 125–137. URL: `http://math.nist.gov/MatrixMarket/` (cited on pages 22, 33).

[27] Bruce Hendrickson and Tamara G. Kolda. "*Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing*", in: *SIAM Journal on Scientific Computing* 21.6 (2000), pp. 2048–2072 (cited on page 24).

[28] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1990. ISBN: 0716710455 (cited on pages 25, 26).

[29] Dénes Kőnig. "*Gráfok és mátrixok*", in: *Matematikai és Fizikai Lapok* 38 (1931), pp. 116–119 (cited on page 26).

[30] John E. Hopcroft and Richard M. Karp. "*An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*", in: *SIAM Journal on computing* 2.4 (1973), pp. 225–231 (cited on page 28).

[31] MathWorks. *C/C++ Source MEX-Files.* 2013. URL: `http://www.mathworks.nl/help/matlab/matlab_external/c-c-source-mex-files.html` (cited on page 32).

[32] David Eppstein. *Hopcroft-Karp bipartite matching (Python recipe).* 2002. URL: `http://code.activestate.com/recipes/123641-hopcroft-karp-bipartite-matching/` (cited on page 32).

[33] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *10th Dimacs Implementation Challenge.* 2012. URL: `http://www.cc.gatech.edu/dimacs10/index.shtml` (cited on page 33).