

Contents

1	Introduction	1
1.1	Parallel sparse matrix-vector multiplication	1
1.2	Hypergraph model	5
1.3	Earlier work	7
1.4	Medium-grain model	8
2	Methods for the computation of A_r and A_c	11
2.1	Individual assignment of nonzeros	11
2.2	Minimization of the empty rows of A_r^T	11
2.3	Assignment of rectangular clusters of nonzeros	11
2.4	Partial assignment of rows and columns	11
3	Independent set formulation	12
3.1	Graph construction	12
3.2	Computation of the independent set	12
3.2.1	Hopcroft-Karp algorithm	12
4	Experimental results	13
4.1	Improving initial partitioning	13
4.2	Fully iterative partitioning	13
	Bibliography	14

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 very large size and involve **sparse** matrices, because the variables at hand are usually connected to a limited number of others (for example, a very 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 $n \times 1$ dense vector, and u the resulting $m \times 1$ vector.

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 $\mathcal{O}(n^2) = \mathcal{O}(mn)$ operations; this is not very efficient if we have 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 we have that $N \ll mn$.

As mentioned, the systems considered are very 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 a very 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 very 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, \dots, 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, and usually it is not, the same).

Figure 1.1 shows an example of such distribution, in which a 5×5 matrix A with 13 nonzeros and the two vectors u and v are split up among two processors, 0 and 1 (denoted, respectively, by the gray and the black color).

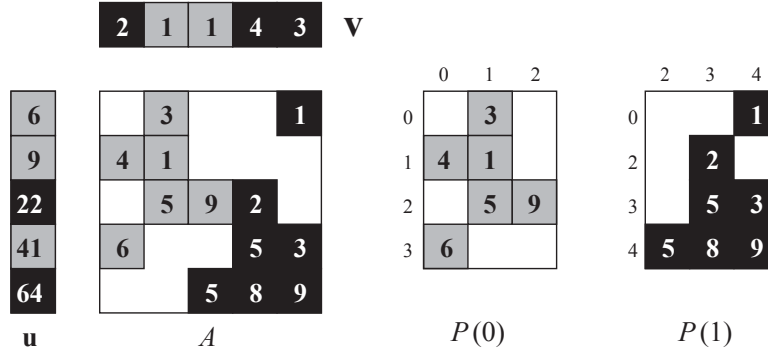


Figure 1.1: Example of a possible distribution among 2 processors of a 5×5 matrix, and the input and output vectors, as taken from [2, Fig. 4.3].

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 $s = 0, \dots, p - 1$, 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 very small sum with negligible computational cost and for this reason it has been omitted from the algorithm.

Figure 1.2 shows an example of the communication involved in supersteps (0) and (2): the fan-out is represented by the vertical arrows, while the fan-in is represented by the horizontal arrows.

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 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):

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

It is easy to understand that, in order to have efficient parallelization, the computation load has to be

Input: A_s , the local part of the vector v

Output: The local part of the vector u

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

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

```
(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.

distributed evenly. Usually, however, it is not possible to achieve a perfect load balance (e.g. when dividing up an odd number of computations among an even number of processors) and we have to reason in term of 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]:

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

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, after all, 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 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 $p = 2$: the difference between a “checkered” distribution (i.e. the nonzeros are assigned alternately to the processors, looking at both the rows and the columns) and a distribution in which every nonzero is assigned to processor 0 (forgetting for one moment about the load balance), is very large. In the first case we have the maximum possible communication, regardless of the vector distribution, whereas in the second case there is no communication at all.

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 nonzeros, and the partitioning of the vector can be executed according to this: because of the structure of the communication supersteps in Algorithm 1.1, we have that communication is required if and only if the rows/columns of the matrices are *cut*, i.e.

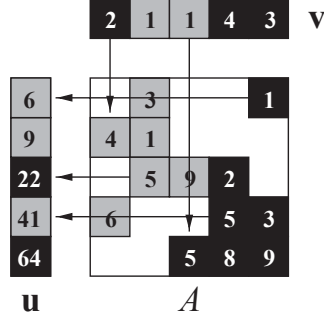


Figure 1.2: Communication involved following Algorithm 1.1 for a matrix distributed as in Figure 1.1. Vertical arrows represent step (0) while horizontal ones represent step (2).

assigned to more than one processor.

If a full column of our matrix A is assigned to the same processor, we can freely assign the first 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 extended for the rows. This simplification is possible because imposing a hard constraint similar to (1.3) also to the vector distribution is not very 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, \dots, 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 .

Then the total time costs for the communication steps in our Algorithm 1.1 are:

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

These costs are quite straightforward: it is reasonable to assume that the owner of the appropriate vector component is between the ones that have a nonzero in that row/column, and therefore one communication is not necessary.

Adding up 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 nonzero 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 always been used to model the load balancing in parallel computing [5]: 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 [6], 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):

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

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 *cutsizes*, 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 [6][7], that this cut-edge metric is not an accurate representation of the communication volume. Additional criticism [8] comes from the fact that the graph partitioning approach can only handle square symmetric matrices. It was also shown [9] 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.

In [6] it is shown that a more correct way of modeling the matrix partitioning problem is through the concept of hypergraph partitioning.

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

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 [10], as well as urban transportation design and study of propositional logic [11].

Because of this extensive literature, 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 1.3 shows an example of such hypergraph. Each colored set represents a different hyperedge; we can see that we can have hyperedges which contain only one node.

The definition of hypergraph partitioning problems is identical to the case of simple graph, with the difference that now we don't have cut edges, but cut hyperedges: given the hyperedge $e = \{e_1, \dots, e_k\}$, we say that e is cut if there are i, j such that $e_i \in V_r, e_j \in V_s$, 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

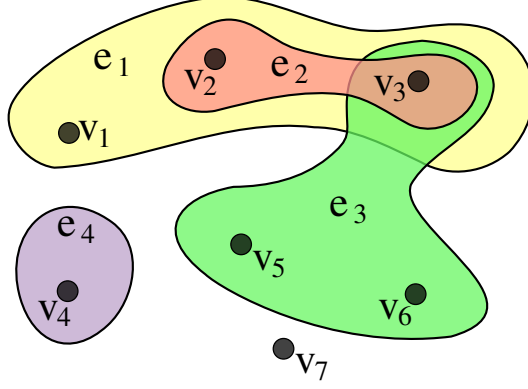


Figure 1.3: Hypergraph example. Image courtesy of Wikimedia Commons: <https://commons.wikimedia.org/wiki/File:Hypergraph-wikipedia.svg>.

assigned to, we have that the total cost of the partition (V_0, \dots, V_p) is:

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

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 partitioning hypergraph model, along with the simple graph partitioning problem, are known to be NP-hard [12, 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 ($V = \{v_1, \dots, v_m\}$), while columns are hyperedges ($E = \{e_1, \dots, e_n\}$). 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, we have that the size of the hyperedge e_j is exactly the number of nonzeros in that column, whereas the node v_i belongs exactly to as many hyperedges as there are nonzeros in that row.

As already said, performing a partitioning on the hypergraph consists in assigning each vertex to one of the sets V_0, \dots, 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 $v_j \in V$, we define its weight c_j as

$$c_j := |\{i : a_{ij} \neq 0\}|,$$

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 $N = \sum_{v_j \in V} c_j$.

Our modified balance constraint is as follows:

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

The **row-net** model is very similar the one just described (as can be guessed from the name): it is exactly the symmetrical of the column-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 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 [13].

In this model, we have that the N nonzeros are the vertices ($V = \{v_1, \dots, v_N\}$) and the m rows and n columns are hyperedges ($E = E_r \cup E_c = \{e_1, \dots, e_m\} \cup \{e_{m+1}, \dots, e_{m+n}\}$). 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 to (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

The hypergraph partitioning problem has a wide variety of applications and therefore there is a somewhat extended literature available; for this reason, it is convenient to translate a instance of our matrix partitioning problem to it in the best possible way.

Among the models which are used to perform this translation, we already mentioned row-net and column-net [6], proposed in 1999, and a more recent fine-grained approach [13], proposed in 2001. New models are relatively rare, and only recently Pelt and Bisseling proposed a interesting new one [14], the **medium-grain** model. As this model is at the very 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 complicated methods, which often comprise several stages.

For example, in [15] Uçar and Aykanat 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 [16]: 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, Vastenhout 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.

Beside this general purpose models and methods, it is also possible to take into account the structure of the matrix to be partitioned: the hypergraph-based approach was indeed initially devised for structurally symmetric matrices [6], and only later it was shown to be efficient also in case of rectangular matrices [17]. Moreover, Hu, Maguire and Blake present in [18] an algorithm for nonsymmetric matrices that performs row and column 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 model, it might be difficult to choose the best one, given a matrix to partition. Çatalyürek, Aykanat, and Uçar propose a partitioning recipe [19] 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) [20], hMetis [21] (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)[18]. Zoltan-PHG (Parallel Hypergraph Partitioner) [5] 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 very different results, with respect to both solution quality and execution time, despite having at the core the same method for finding good initial solutions. The method employed is the well-known Kernighan-Lin [22] method, with the optimizations of Fiduccia-Mattheyses [23]. 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 find improvement in the quality 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 operate a sequence of bisection of the partitions. This is a good simplification in the sense that it just suffices to find very 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 with direct splitting into the desired number of parts.

1.4 Medium-grain model

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

A new model has recently been proposed by Pelt and Bisseling [14], which can be described as a sort of middle ground between the 1-dimensional ones and fine-grain. 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 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

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

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

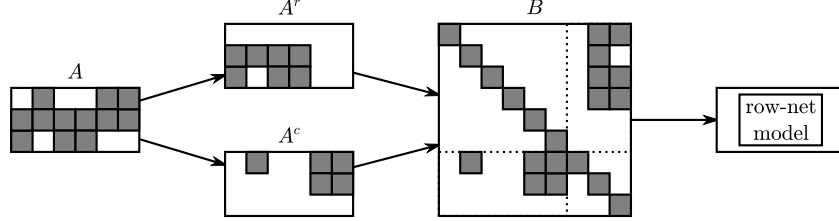


Figure 1.4: Construction of the hypergraph according to the medium-grain model.

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 1.5.

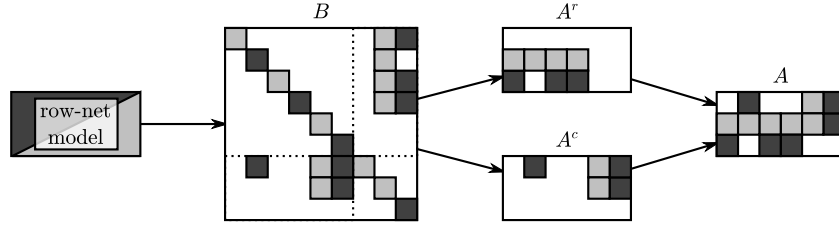


Figure 1.5: Process of obtaining a matrix partitioning starting from a partitioning of the hypergraph following the medium-grain model. In this case $p = 2$.

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 we have 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 k -th column of A ; the corresponding nonzeros can be found in the k -th column of A_c and in the k -th row of A_r^T . If both these parts are nonempty, i.e. the k -th 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 k -th column of A_c , that belongs to same row of B as the k -th row of A_r , has the purpose of ensuring a correct computation of the communication volume [14, 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 seems to confirm that this model has indeed some advantages with respect to column-net, row-net and fine-grain, both regarding partitioning time and solution quality.

Because of these good results, it is interesting 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 computational time is traded for solution quality.

Both these research directions share a very important part: we just need to develop efficient methods to compute A_r or A_c , 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 with respect the two mentioned research directions.

Chapter 2

Methods for the computation of A_r and A_c

As described in Section 1.4, our main goal is to find efficient ways of splitting our original matrix A into A_r and A_c , and then use these in the medium-grain model.

Since we are interested in both improving the initial partitioning of A and a fully iterative method, in the next sections we will discuss methods that can either take as input an unpartitioned matrix or a partitioned one.

2.1 Individual assignment of nonzeros

2.2 Minimization of the empty rows of A_r^T

2.3 Assignment of rectangular clusters of nonzeros

2.4 Partial assignment of rows and columns

Chapter 3

Independent set formulation

3.1 Graph construction

3.2 Computation of the independent set

3.2.1 Hopcroft-Karp algorithm

Chapter 4

Experimental results

4.1 Improving initial partitioning

4.2 Fully iterative partitioning

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, 6).
- [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, 7, 8, 10).
- [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] Karen D. Devine et al. “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 pages 5, 8).
- [6] 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, 7, 8).
- [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 5).
- [8] 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 5).
- [9] Bruce Hendrickson and Tamara G. Kolda. “Graph partitioning models for parallel computing”, in: *Parallel computing* 26.12 (2000), pp. 1519–1534 (cited on page 5).
- [10] G. Karypis et al. “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 5).
- [11] David A. Papa and Igor L. Markov. “Hypergraph Partitioning and Clustering”. In: *In Approximation Algorithms and Metaheuristics*. 2007 (cited on page 5).
- [12] 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 6).
- [13] Ümit V. Çatalyürek and Cevdet Aykanat. “A fine-grain hypergraph model for 2D decomposition of sparse matrices”. In: *in: Proceedings of the 15th International Parallel and Distributed Processing Symposium, 2001, p. 118. C. Aykanat*, pp. 609–625 (cited on page 7).
- [14] Daniël M. Pelt and Rob H. Bisseling. “A Medium-Grain Model for Fast 2D Bipartitioning of Sparse Matrices”, submitted. 2013 (cited on pages 7–9).
- [15] Bora Uçar and Cevdet Aykanat. “Revisiting hypergraph models for sparse matrix partitioning”, in: *SIAM review* 49.4 (2007), pp. 595–603 (cited on page 7).

- [16] 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 7).
- [17] Ali Pinar et al. “Decomposing linear programs for parallel solution”. In: *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*. Springer, 1996, pp. 473–482 (cited on page 8).
- [18] Y.F. Hu, K.C.F. Maguire, and R.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).
- [19] 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).
- [20] 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).
- [21] George Karypis and Vipin Kumar. “Multilevel k -way hypergraph partitioning”, in: *VLSI design* 11.3 (2000), pp. 285–300 (cited on page 8).
- [22] B.W. Kernighan and S. Lin. “An efficient heuristic procedure for partitioning graphs”, in: *Bell system technical journal* (1970) (cited on page 8).
- [23] 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).