Chapter 12

Two-Dimensional Approaches to Sparse Matrix Partitioning

Rob H. Bisseling, Bas O. Fagginger Auer, A. N. Yzelman
Utrecht University

Tristan van Leeuwen
University of British Columbia

Ümit V. Çatalyürek
Ohio State University

12.1 Introduction

Parallel computers are appearing on our desktops, as the multicore revolution is driving technology towards parallelism. In our applications, this causes the need for partitioning of the computational work and the corresponding data. In scientific computing, these data are often represented by a sparse

matrix, which has many zero elements but stores the interesting information in its nonzero elements.

To partition well, we need to balance the amount of work among the processors and minimize the amount of communication caused by having nonzeros in different processors. In this chapter, we focus on an important sparse matrix computation: the multiplication of a sparse matrix A by a (dense) input vector $\mathbf{v}$, which yields as output a (dense) vector $\mathbf{u} = A\mathbf{v}$. Here, A is of size $m \times n$, $\mathbf{v}$ is of length n, and $\mathbf{u}$ is of length m. This multiplication is the most time-consuming operation of most iterative solvers for linear systems [1] and eigensystems [2].

The load balance of the computation is commonly expressed by the following constraint:

$$\mathrm{nz}(A_s) \leq (1+\epsilon)\frac{\mathrm{nz}(A)}{p}, \quad \text{for } s = 0, 1, \ldots, p-1. \tag{12.1}$$

Here, $\mathrm{nz}(A)$ denotes the number of nonzeros of the matrix A and $\mathrm{nz}(A_s)$ the number of nonzeros of matrix part A_s. The parts are numbered from 0 to $p-1$, where p is the number of processors (or processor cores) among which A needs to be distributed; to avoid confusion, we will always call the basic processing element a *processor*. The allowed load imbalance $\epsilon > 0$ is a user-specified parameter. The proper choice of ϵ depends on the characteristics of the parallel computer: for a machine with faster communication, the load imbalance becomes more important, and hence ϵ should be reduced. In our exposition, we assume that the parallel computer has a distributed memory, so that the partitioning implies an actual physical distribution of the data. Still, in case of a shared memory, a good partitioning is also needed, primarily to divide the work, but also to reduce memory conflicts between processors trying to access the same location in the shared memory.

Communication arises because the nonzeros in a row or column may not all reside at the same processor, having been assigned to different parts by the partitioning. Since the aim of a matrix–vector multiplication is to compute $\mathbf{u} = A\mathbf{v}$, i.e.,

$$u_i = \sum_{j=0}^{n-1} a_{ij} v_j, \quad \text{for } i = 0, 1, \ldots, m-1, \tag{12.2}$$

nonzeros of a row assigned to different parts cause communication, as products $a_{ij}v_j$ or sums of such products have to be sent to the owner of vector component u_i. In the worst case, all p processors have nonzeros in row i, and $p-1$ processors need to send a contribution to u_i. In general, the number of processors having part of row i equals p_i with $p_i \leq p$, and the number of data words to be sent for computing u_i will be at most $p_i - 1$. (The owner of u_i is equal to one of the owners of a nonzero in row i, thus preventing an unnecessary communication.) For columns, similar considerations hold: the vector component v_j must be made known to all q_j processors that have a nonzero in column j, giving rise to $q_j - 1$ communications of a data word.

The aim of matrix partitioning is to find an assignment of the nonzeros to parts such that Equation (12.1) is satisfied and the *communication volume*, the total number of data words communicated, is minimized. The volume is defined by

$$V = \sum_{i=0}^{m-1} (p_i - 1) + \sum_{j=0}^{n-1} (q_j - 1), \tag{12.3}$$

where we assume that matrix rows and columns are not empty.

One-dimensional (1D) partitioning has been common practice for decades, usually assigning complete rows to a processor; of course assigning complete columns is also possible. Sometimes this is natural because a row has a physical meaning, like the connections of a grid point in a regular computational grid. Often, all the information concerning a grid point is kept together, and assigning the corresponding matrix row to one processor is natural. In other situations, there is not much meaning beyond the matrix entries themselves and the nonzeros of a row could be assigned to different processors. An example is web matrices, where rows/columns represent webpages, and nonzeros represent links between them. An advantage of 1D partitioning is that the parallel matrix–vector multiplication becomes slightly simpler; e.g., in the case of row partitioning (assigning complete rows to parts) we do not need to send contributions to the output vector, saving one phase of the computation (and, depending on the type of parallel computing model used, saving one global synchronization). The disadvantage is that the remaining phase may become more expensive. As a solution to an optimization problem, minimizing the communication in a 1D fashion is less general, and hence could be suboptimal.

Two-dimensional (2D) partitioning of sparse matrices is more recent, and it has been shown to be highly effective in distributing the work and minimizing communication for parallel computations in a wide range of applications [4, 5, 6, 7, 8].

Two types of 2D methods are particularly promising: (i) the *fine-grain approach* by Çatalyürek and Aykanat [4, 5] which partitions the nonzero matrix elements looking at individual elements, thereby allowing for the most general solution; (ii) the *Mondriaan approach* by Vastenhouw and Bisseling [8] which recursively bipartitions the sparse matrix into two (not necessarily contiguous) submatrices, trying splits in both the row and column directions and each time choosing the best. This orthogonal recursive bisection method is of the coarse-grain type; each split of a submatrix keeps either the nonzeros of the rows together as an atomic unit, or those of the columns. Both approaches use a multilevel hypergraph partitioner as their splitting engine, which minimizes the communication volume in the exact metric given by Equation (12.3).

A natural question to ask is whether combining the two approaches in a hybrid method would lead to savings in communication volume. This chapter explains the different approaches and also presents a new hybrid fine-grain/Mondriaan approach which tries to split the current subset of nonzeros

in three possible ways: by rows, by columns, and by individual nonzeros, and then chooses the best split. Thus, the partitioning algorithm automatically detects the best method for the type of matrix involved, and adjusts this decision to the current submatrix as the partitioning progresses, at the cost of an additional splitting attempt for each bipartitioning.

As a final note, partitioning methods for parallelizing the sparse matrix–vector multiplication can easily be combined with methods for accelerating sequential sparse matrix–vector multiplication, e.g., autotuning by using OSKI [9], or matrix reordering to improve cache use [3], as these can be applied directly to the local part of the matrix–vector multiplication.

12.2 Sparse Matrices and Hypergraphs

We will now reformulate the matrix partitioning problem of the previous section as a hypergraph partitioning problem. A *hypergraph* $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ consists of a set of vertices $\mathcal{V}$ and a set of *hyperedges* or *nets* $\mathcal{N}$, which are subsets of $\mathcal{V}$. A hypergraph is a generalization of an *undirected graph*, which represents the special case where the subsets are pairs of vertices. A *bipartitioning* of a hypergraph is a splitting of the vertices into two disjoint nonempty sets $\mathcal{V}_0$ and $\mathcal{V}_1$ with $\mathcal{V}_0 \cup \mathcal{V}_1 = \mathcal{V}$.

A sparse matrix can be converted to a hypergraph in various ways. Since the numerical value of the matrix nonzeros is irrelevant for minimizing the communication volume given by Equation (12.3), we only consider the sparsity pattern of the nonzeros. The first way of converting is by the *column-net model* [10], where each row i corresponds to a vertex v_i and each column j to a net $n_j = \{v_i : 0 \leq i < m \wedge a_{ij} \neq 0\}$. This hypergraph is denoted by $\mathcal{H}_\mathrm{c}$; it contains all the information on the sparsity pattern of the matrix. An example is given by Figure 12.1. A different way is by the *row-net model*, where the roles of rows and columns are reversed: rows correspond to nets, and columns to vertices. In this case, the hypergraph is denoted by $\mathcal{H}_\mathrm{r}$. Both ways are 1D in nature. A different, two-dimensional way of conversion will be discussed in Section 12.5.

12.3 Parallel Sparse Matrix–Vector Multiplication

A sparse matrix can be multiplied in parallel by a vector using either a 1D or a 2D partitioning. As the algorithm for a 1D partitioning is simpler, we will explain this algorithm first.

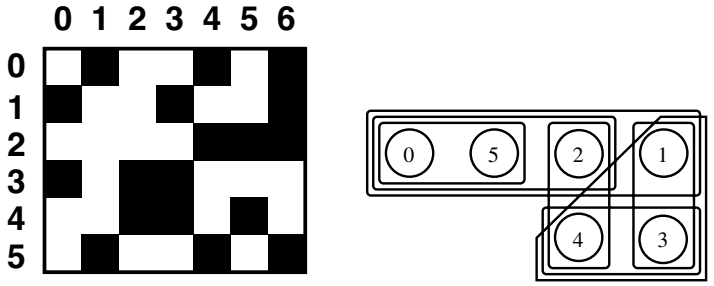

FIGURE 12.1: Example sparse matrix of size 6×7 with 18 nonzeros. Left: the matrix, with nonzeros depicted in black. Right: the column-net hypergraph created from the matrix, with vertices (circles) representing the rows of the matrix, and nets (sets containing the vertices) representing the columns. For example, column 6 of the matrix has nonzeros $a_{06}, a_{16}, a_{26}, a_{56}$, thus creating the net $\{0, 1, 2, 5\}$ of the hypergraph.

12.3.1 Using a One-Dimensional Partitioning

The natural parallel algorithm for sparse matrix–vector multiplication using a 1D row partitioning can be formulated as follows. Let $P(x)$ denote the processor that owns variable x. We write a_{i*} for row i, and we assume that $P(a_{i*}) = P(u_i)$. Furthermore, we assume that the vector component v_j is assigned to one of the processors with nonzeros in column j. Let I_s be the set of matrix rows of processor s and let J_s be the set of nonempty matrix columns of processor s, i.e.,

$$J_s = \{j : 0 \leq j < n \wedge (\exists i : 0 \leq i < m \wedge a_{ij} \neq 0 \wedge P(a_{ij}) = s)\}. \quad (12.4)$$

Algorithm 4 presents the natural row-based parallel 1D sparse matrix–vector multiplication algorithm. This algorithm consists of two phases, also called *supersteps* in bulk-synchronous parallel (BSP) language [11, 12]. Superstep (0) communicates vector components v_j and superstep (1) computes the vector components u_i. In superstep (1), the nonzeros a_{ij} are multiplied with the corresponding v_j and added into the output component u_i. The total communication volume of the algorithm equals the number of vector components v_j communicated in superstep (0).

The communication volume of parallel sparse matrix–vector multiplication based on a 1D matrix partitioning can be modeled by using a hypergraph where the vertices are partitioned. Assume we have a row partitioning and a corresponding partitioning of the vertices. If λ_j is the number of parts of the partitioning that have vertices in net j, the *cost* of the hypergraph partitioning is defined as

$$C = \sum_{j=0}^{n-1} c_j(\lambda_j - 1), \quad (12.5)$$

Algorithm 4 1D sparse matrix–vector multiplication for processor s.

```
(0)   { Communication of v }
      for all j ∈ J_s do
            get v_j from P(v_j);

(1)   { Local sparse matrix–vector multiplication }
      for all i ∈ I_s do
            u_i := 0;
            for all j : 0 ≤ j < n ∧ a_ij ≠ 0 do
                  u_i := u_i + a_ij v_j;
```

where c_j is a scaling constant reflecting the importance of net j. This cost metric is commonly called the $(\lambda - 1)$-*metric*.

Let $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ be a hypergraph with vertex weights $W : \mathcal{V} \to \mathbf{R}_{\geq 0}$ and net costs $c : \mathcal{N} \to \mathbf{R}_{\geq 0}$. Let p be the desired number of parts and $\epsilon > 0$ the allowed imbalance. Then the *hypergraph partitioning problem* is defined as finding a partitioning $\mathcal{V} = \mathcal{V}_0 \cup \cdots \cup \mathcal{V}_{p-1}$ of the vertices into pairwise disjoint nonempty subsets, such that the *load balance constraint*

$$W(\mathcal{V}_s) \leq (1 + \epsilon)\frac{W(\mathcal{V})}{p}, \quad \text{for } s = 0, 1, \ldots, p-1, \tag{12.6}$$

is satisfied, and the cost, given by Equation (12.5), is minimal.

By the translation of a matrix into a hypergraph for the column-net model, we have that $\lambda_j = q_j$, the number of different processors present in matrix column j. If we now define $c_j = 1$, the cost C becomes exactly equal to the communication volume V of the parallel sparse matrix–vector multiplication, defined by Equation (12.3). If we define vertex weight W_i as the number of nonzeros in matrix row i, the balance constraint becomes equal to the constraint for matrix partitioning, Equation (12.1). This is illustrated by Figure 12.2.

The total communication volume may not be the only important metric. On parallel architectures with high latencies, where there is a large cost for starting up a message, and a much smaller cost for each word of the message, it may be important to keep the number of messages small as well. In particular this holds for smaller problem sizes, and for a large number of processors p, since then the number of send–receive pairs can become $p(p-1)$. In this chapter, however, we will be exclusively concerned with the metric of communication volume.

The translation from a matrix partitioning problem to a hypergraph partitioning problem has made it possible to use existing software such as MLPart [13] and hMETIS [14], developed for the area of VLSI design where hypergraph partitioning is needed for obtaining good circuit layouts. But even

more, the importance of hypergraph partitioning for parallel matrix computations has driven the development of new hypergraph partitioners, such as PaToH [15], Zoltan [16], Par*k*way [17], and also the native hypergraph partitioner of Mondriaan [8]. Of these, the Zoltan hypergraph partitioner (see Chapter 13) and Par*k*way are themselves parallel, which is important for larger problems that cannot be stored on a single processor in the case of distributed-memory parallel architectures. All these partitioners are based on the multilevel idea [18]: first the hypergraph is coarsened by merging vertices, doing this repeatedly at several levels, then an initial partitioning is obtained of the coarse hypergraph, and finally this partitioning is further refined during an uncoarsening phase. A detailed explanation of the multilevel approach, within the context of graph partitioning, can be found in Chapter 14.

The hypergraph bipartitioning problem is NP-hard [19], so all partitioning approaches for large problems will have to be based on heuristics. Even finding good approximate solutions with a guaranteed quality bound is NP-hard, as was shown for graphs by Bui and Jones [20].

Graph partitioning is used with much success in the area of parallel mesh computations, which are often based on finite elements. Software such as Chaco [21], Metis [22], ParMetis [23], Jostle [24], Scotch [25], PT-Scotch [26], can be employed to partition meshes and, more in general, graphs, typically aiming at reducing the total number of cut edges. This is not the same as reducing the total communication volume in a parallel mesh computation, as two cut edges (i, j) and (i, j') with $j \neq j'$ may cause only one communication leaving the source processor $P(i)$ owning vertex i, since it might happen that $P(j) = P(j')$, i.e., the destination processors are the same. Therefore, minimizing the edge cut amounts to minimizing an upper bound on the communication, not the actual communication, and hence is an approximation. Since graph partitioning is in general faster than hypergraph partitioning, it may still be the method of choice for problems where the approximation is good. The packages ParMetis [23] and PT-Scotch [26] are parallel, and Jostle [24] also has a parallel version. Since a symmetric sparse matrix can be viewed as the adjacency matrix of a graph, with $a_{ij} \neq 0$ if and only if (i, j) is an edge of the graph, graph partitioners can also be used to partition such sparse matrices. Historically, graph partitioners were even the first to be used, before the hypergraph partitioners came to the field [27]. For more on graph partitioning, see Chapter 14. An overview of existing software packages ordered by problem type and by the possibility to run the package in parallel is given in Table 12.1.

12.3.2 Using a Two-Dimensional Partitioning

The natural parallel algorithm for sparse matrix–vector multiplication with the most general (2D) matrix distribution can be formulated as follows. We assume that all data elements, i.e., the matrix nonzeros a_{ij} and the vector components u_i and v_j, have been assigned to arbitrary processors. Let I_s now

TABLE 12.1: Overview of available software for partitioning graphs and hypergraphs.

Name	Graph/ hypergraph	Sequential/ parallel	Reference
Chaco	graph	sequential	[21]
Metis	graph	sequential	[22]
Scotch	graph	sequential	[25]
Jostle	graph	parallel	[24]
ParMetis	graph	parallel	[23]
PT-Scotch	graph	parallel	[26]
hMETIS	hypergraph	sequential	[14]
ML-Part	hypergraph	sequential	[13]
Mondriaan	hypergraph	sequential	[8]
PaToH	hypergraph	sequential	[15]
Par$k$way	hypergraph	parallel	[17]
Zoltan	hypergraph	parallel	[16]

be the set of nonempty matrix rows of processor s, i.e.,

$$I_s = \{i : 0 \leq i < m \wedge (\exists j : 0 \leq j < n \wedge a_{ij} \neq 0 \wedge P(a_{ij}) = s)\} \qquad (12.7)$$

and, as before, let J_s be the set of nonempty matrix columns of processor s, see Equation (12.4). Algorithm 5, adapted from [11], presents the natural parallel 2D sparse matrix–vector multiplication algorithm. This algorithm consists of four supersteps. These supersteps alternate between communication and computation: first we communicate, then compute, then communicate, and finally compute again. Here, the central superstep is (1), where nonzeros a_{ij} are multiplied with the corresponding v_j and added to the local contribution u_{is} of processor s to the output component u_i. The total communication volume of the algorithm is the sum of the number of vector components v_j communicated in superstep (0) and the number of contributions u_{is} communicated in superstep (2). For convenience, we take as the total computation cost the cost of superstep (1) measured in multiplications $a_{ij} \cdot v_j$. This cost is defined as the maximum over the processors of the number of local nonzeros. We ignore the cost of summing the contributions in superstep (3); this cost is much less than that of sending them in the preceding superstep.

The communication requirements of parallel sparse matrix–vector multiplication based on a 2D matrix partitioning is illustrated by Figure 12.3. Using both dimensions gives more flexibility, for instance to better use an allowed imbalance to reduce communication, as happens in the example.

In this chapter, we only discuss matrix partitioning, and not the vector partitioning, aiming at balancing the computational work load and minimizing the total communication volume. We assume that the vector is partitioned after the matrix partitioning in such a way that this does not cause additional

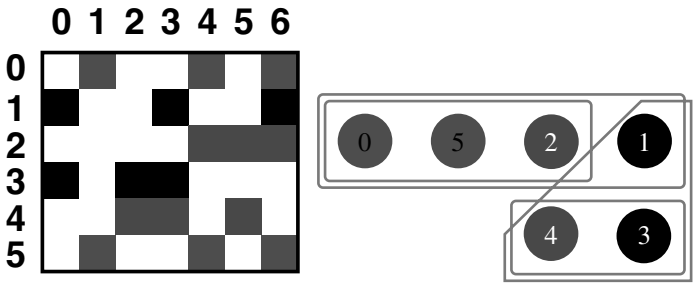

FIGURE 12.2: (See color insert.) Sparse matrix from Figure 12.1 partitioned into 3 parts by a 1D row partitioning. Rows 0 and 5 are assigned to processor 0 (red), rows 1 and 3 to processor 1 (black), and rows 2 and 4 to processor 2 (blue). Left: the matrix, with nonzeros painted by the color of their processor. Right: the column-net hypergraph created from the matrix, with only the cut nets shown (in green). The vertices are painted in the same color as their corresponding row. Column 6 has $\lambda_6 = 3$ processors, causing $\lambda_6 - 1 = 2$ communications during sparse matrix–vector multiplication, and the corresponding net $\{0, 1, 2, 5\}$ has three parts. This means that vector component v_6 has to become known to all three processors, and hence it has to be sent by the source processor to the other two processors. The other cut columns are columns 2, 3, 4; they each have two parts and cause one communication. Columns 0, 1, and 5 do not cause communication. The load balance happens to be perfect. The communication volume as defined by Equation (12.3) equals $V = 2 + 1 + 1 + 1 = 5$, which is the sum of the number of communications for the separate columns.

communication, meaning that $P(u_i)$ is one of the processors with nonzeros in row i, and $P(v_j)$ is one of the processors with nonzeros in column j. This leaves some freedom that can be used to achieve a secondary objective, e.g., balancing the communication load [28], reducing the number of messages [29], or balancing the number of vector components. The latter objective may be important in an iterative solver with many vector operations such as GMRES [30].

12.4 Coarse-Grain Partitioning

Coarse-grain approaches to partitioning keep naturally occurring sets of nonzeros such as rows or columns (or parts of rows or columns) together during the splits of the partitioning process; this is in contrast with the fine-grain approach, where nonzeros are handled individually. In the present section, we

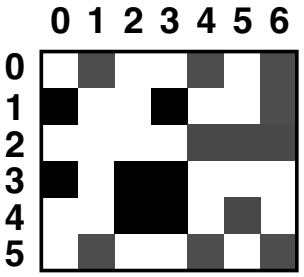

FIGURE 12.3: (See color insert.) Sparse matrix from Figure 12.1 partitioned into 3 parts by a 2D partitioning, using the Mondriaan package with the hybrid approach of this chapter. Rows 1 and 4, and columns 4 and 6 each cause one communication. The load imbalance is one nonzero, i.e., $\epsilon = 1/6$. The total communication volume is 4.

review various types of coarse-grain partitioning. In the next section, we will discuss the fine-grain approach.

12.4.1 Cartesian Partitioning and Its Variants

Cartesian partitioning, also known as *checkerboard partitioning* and *rectilinear partitioning*, is one of the frequently used matrix partitioning methods for dense matrices or sparse matrices with structured patterns that are difficult to exploit [31, 32, 33, 34, 35, 36]. A Cartesian partitioning is obtained by partitioning the rows of the matrix into P subsets, the columns into Q subsets, with $p = PQ$, and then assigning the Cartesian product of a row subset and a column subset to a processor, for each of the p possible combinations of row subset and column subset. This is a desirable partitioning due to its advantage of naturally limiting the number of processors communicating in each communication superstep of parallel 2D matrix–vector multiplication.

For sparse matrices, one can also achieve Cartesian partitioning by intelligently combining row-net and column-net hypergraph models with multi-constraint partitioning [4, 6, 7]. In this two-step method, for a $P \times Q$ processor mesh with $p = PQ$, the matrix is first partitioned into P horizontal strips (blocks of rows, which may have been permuted) using the column-net hypergraph model. Then, in the second step, the same matrix is partitioned into Q vertical strips using the row-net hypergraph model with multi-constraint partitioning, such that each vertex (representing a matrix column) will have P weights corresponding to the number of nonzeros in the P different blocks of the first partitioning. Here, for each of the P blocks of rows, a constraint of the form (12.1) must be satisfied, with Q parts instead of p. It is, of course, also possible to start the process in the vertical direction.

Despite the obvious communication advantages, Cartesian partitioning is

Algorithm 5 2D sparse matrix–vector multiplication for processor s.

```
(0)   { Communication of v }
      for all j ∈ J_s do
            get v_j from P(v_j);

(1)   { Local sparse matrix–vector multiplication }
      for all i ∈ I_s do
            u_is := 0;
            for all j : 0 ≤ j < n ∧ a_ij ≠ 0 ∧ P(a_ij) = s do
                  u_is := u_is + a_ij v_j;

(2)   { Communication of contributions to u }
      for all i ∈ I_s do
            put u_is in P(u_i);

(3)   { Summation of contributions }
      for all i : 0 ≤ i < n ∧ P(u_i) = s do
            u_i := 0;
            for all t : 0 ≤ t < p ∧ u_it ≠ 0 do
                  u_i := u_i + u_it;
```

very restrictive and for sparse matrices with very irregular patterns, it may yield a higher communication volume. Another partitioning method, *jagged partitioning* [35, 37], is a variant of Cartesian partitioning. In the jagged partitioning for a $P \times Q$ processor mesh, the matrix is first partitioned into P horizontal strips, just as for Cartesian partitioning, but with contiguous rows in each strip. In the second step, however, every horizontal strip is independently partitioned into Q submatrices, each containing a set of contiguous columns. This yields a partitioning that is not Cartesian, since splits span the entire matrix in one dimension but they are jagged in the other dimension. The resulting submatrices are contiguous blocks in the original matrix. Although efficient algorithms exist for producing jagged partitions with optimal load balance [38, 39, 40], they do not consider the minimization of communication volume explicitly. The *jagged-like partitioning* method [4, 7] utilizes 1D row-net and column-net hypergraph models to achieve communication-aware jagged matrix partitionings. The reason this method is called jagged-like instead of jagged is that it no longer enforces the rows and columns of the submatrices to be contiguous.

12.4.2 Mondriaan Partitioning by Orthogonal Recursive Bisection

The Mondriaan approach [8] to partitioning consists of a sequence of splits, each time trying both the row and column direction, and then taking the best of the two (i.e., the one with the lowest communication volume). Each split is thus 1D, and the overall result is a 2D matrix partitioning. The motivation for this approach is that by keeping the nonzeros of a row or column together, communication in one direction is prevented by construction, while we still benefit from the flexibility of using both directions in the overall process. The 1D splits are a hypergraph-based bipartitioning of the current largest part in a prescribed ratio of numbers of nonzeros, such as 1:2 for the first split in the case of $p = 3$. An advantage of 1D splittings is that they are relatively fast, since they consider hypergraphs with a limited number of vertices, namely m or n for the first split. For $p = 2$, the Mondriaan method takes the best of two 1D partitionings, and hence the result is a 1D partitioning. Only for $p > 2$, it potentially exploits both dimensions.

A nice property of the sparse matrix–vector multiplication algorithm, Algorithm 5, is that the total communication volume for a p-way matrix partitioning obtained by successive bipartitionings is just the sum of the communication volumes of the separate splits, where each split can be seen as acting on a sparse matrix of the same size as A but containing only those nonzeros of the subset to be split. Thus, split k acts on an $m \times n$ matrix $A^{(k)}$, for $k = 0, \ldots, p-2$, where $A^{(0)} = A$. This enables looking at every split in isolation. This property was shown for a 1D partitioning in [10] and for an arbitrary partitioning in [8, Theorem 2.2].

The result of the Mondriaan splitting is not necessarily Cartesian, as the parts created by a bipartitioning are split independently later on. Still, the partitioning has internal structure: the nonzeros a_{ij} of processor s have indices in the set $I_s \times J_s$, cf. Equations (12.4) and (12.7), which represents a (scattered) submatrix of A; all these submatrices are disjoint, and by construction they fit in the original matrix, as the colored rectangles in the paintings by the Dutch painter Piet Mondriaan (1872–1944). This is illustrated by Figure 12.4. Such structure will generally not be present in an arbitrary partitioning of the nonzeros, where the submatrices enclosing the nonzeros of a processor may overlap.

12.5 Fine-Grain Partitioning

The *fine-grain approach* [4, 5] to partitioning translates the matrix to a hypergraph $\mathcal{H}_{\mathrm{f}}$, as follows. Every vertex represents a matrix nonzero. There are two types of nets: row nets and column nets. A *row net* i contains all

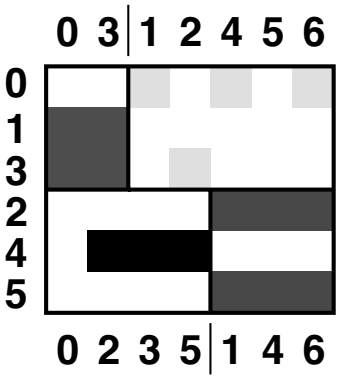

FIGURE 12.4: (See color insert.) Sparse matrix from Figure 12.1 partitioned into 4 parts by a Mondriaan partitioning. The first split is in the row direction, whereas the second and third splits are in the column direction. The matrix is shown in the local view, displaying the local submatrices of the processors. The column indices are different for the top and the bottom part. The submatrix $I_0 \times J_0$ in the upper left corner contains the nonzeros assigned to processor 0 (red), where $I_0 = \{0, 1, 3\}$ and $J_0 = \{0, 3\}$.

the vertices corresponding to nonzeros in matrix row i, and a *column net* j contains all the vertices corresponding to nonzeros in matrix column j. The total number of vertices of $\mathcal{H}_\mathrm{f}$ is $\mathrm{nz}(A)$ and the number of nets is $m + n$, assuming that no row or column is empty.

The hypergraph can then be partitioned into p parts, and the corresponding cost given by Equation (12.5) with $c_j = 1$ is exactly equal to the communication volume of the sparse matrix–vector multiplication, as every column net j that is cut into λ_j parts corresponds to a column that is cut into the same number of parts, so that $q_j = \lambda_j$, and similarly for cut row nets. Choosing vertex weights of 1 makes the load balance constraint of the hypergraph identical to that of the matrix, given by Equation 12.1. As a result, any hypergraph partitioner can be used, either splitting the hypergraph directly into p parts, or using recursive bipartitioning.

A different view of the fine-grain representation of a sparse matrix can be obtained by first translating the matrix A into its corresponding fine-grain hypergraph $\mathcal{H}_\mathrm{f} = \mathcal{H}_\mathrm{f}(A)$, and then translating it back into a matrix, but not by the same model, using either the row-net model, yielding the matrix $F = F(A)$ shown in Figure 12.5, or the column-net model, yielding the matrix F^T. The 2D partitioning of the matrix A then corresponds to a 1D column partitioning of the matrix F. Note that the matrix F, which is of size $(m + n) \times \mathrm{nz}(A)$, has a special form: for instance, in the figure every column has exactly two

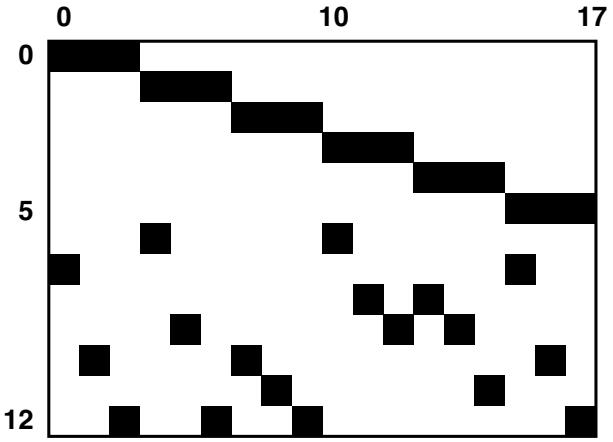

FIGURE 12.5: Fine-grain incidence matrix $F = F(A)$ of size $(6 + 7) \times 18$ created from the 6×7 matrix A with 18 nonzeros of Figure 12.1 by translating the fine-grain hypergraph $\mathcal{H}_{\mathrm{f}} = \mathcal{H}_{\mathrm{f}}(A)$ back into a matrix by the row-net model. The kth nonzero a_{ij} in A corresponds to column k in F, which has 2 nonzeros, f_{ik} and $f_{(m+j),k}$, where $m = 6$ is the number of row nets.

nonzeros, one corresponding to a row net from $\mathcal{H}_{\mathrm{f}}$ and the other to a column net.

12.6 The Hybrid Partitioning Algorithm

The Mondriaan method tries to bipartition the hypergraphs $\mathcal{H}_{\mathrm{r}}$ and $\mathcal{H}_{\mathrm{c}}$. It is a straightforward extension of this method to bipartition the fine-grain hypergraph $\mathcal{H}_{\mathrm{f}}$ as well. This can be done within the same framework of the Mondriaan p-way matrix partitioner, and using the same hypergraph splitter. We call the resulting method the *hybrid fine-grain/Mondriaan method* for sparse matrix partitioning, or in short the *hybrid method*.

This hybrid method will cost an additional hypergraph bipartitioning run for every split, which may be expensive because the fine-grain bipartitioning, which is 2D, needs more computation time and memory than the row and column bipartitionings, which are 1D. Therefore, an important implementation issue in making the hybrid method work in practice is exploitation of the special structure that every hypergraph $\mathcal{H}_{\mathrm{f}}$ possesses.

Algorithm 6 presents a recursive formulation of the hybrid method. The algorithm splits the current subset A of nonzeros into two parts, with a fraction

p_0/p of the nonzeros assigned to part 0 and a fraction $1 - p_0/p = p_1/p$ to part 1. The values of p_0 and p_1 are chosen as close as possible to $p/2$, to minimize the number of split levels. Following the adaptive strategy proposed in [8], the imbalance allowed in the first split is based on the assumption that each subsequent split in a path down the binary splitting tree adds the same imbalance. For part r, with $r = 0, 1$, the first split thus allows an imbalance of ϵ/q_r, with $q_r = \lceil \log_2 p_r \rceil + 1$. The imbalances are recomputed when calling the function recursively, based on the actual current division of nonzeros and on the corresponding number of parts.

The function split$(A, \text{method}, \delta_0, \delta_1, f)$ is not given in detail here; an outline of how it works follows. The aim is to split A into two parts, with part 0 having at most a fraction $(1 + \delta_0)f$ of the nonzeros and part 1 at most a fraction $(1+\delta_1)(1-f)$. The matrix is first translated into a hypergraph for the chosen splitting method (row, column, or fine-grain), and then this hypergraph is split into two parts using multilevel partitioning, see Chapter 14. This is done by coarsening the hypergraph repeatedly, merging similar vertices until the hypergraph is small enough, and then running a heuristic Kernighan–Lin/Fiduccia–Mattheyses (KLFM) algorithm [41, 42] which assigns merged vertices to the two parts trying to minimize communication while balancing the nonzeros within the specification. Then the hypergraph is uncoarsened while a simpler version of KLFM is run to refine the partitioning at each level.

In Algorithm 6, all three hypergraph-based splitting methods are tried, i.e., by rows, by columns, and by the fine-grain method, and the best is chosen. This is motivated by the idea that often taking rows or columns as atomic units is beneficial, as this completely prevents communication in one direction and requires the solution of a smaller hypergraph partitioning problem (in terms of number of vertices), but that sometimes a true 2D bipartitioning is required. Since it cannot be easily predicted when this situation occurs, this should be automatically detected by the algorithm. Figure 12.6 illustrates the hybrid method.

12.7 Time Complexity

The time complexity of the complete hybrid p-way partitioning procedure of an $m \times n$ matrix is

$$T = \mathcal{O}((m + n) \cdot \text{nz}^{\text{r}}_{\text{max}} \cdot \text{nz}^{\text{c}}_{\text{max}} \cdot \log_2 p), \tag{12.8}$$

where $\text{nz}^{\text{r}}_{\text{max}}$ is the maximum number of nonzeros in a matrix row and $\text{nz}^{\text{c}}_{\text{max}}$ the maximum in a column, which is briefly explained as follows. Consider a 1D column partitioning. The first merging step of the coarsening computes inner products between pairs of matrix columns, which can be done in at most $\mathcal{O}(n \cdot \text{nz}^{\text{r}}_{\text{max}} \cdot \text{nz}^{\text{c}}_{\text{max}})$ operations, since for each of the n columns at most

Algorithm 6 Recursive adaptive algorithm for the computation of a p-way partitioning of a sparse matrix A with an allowed load imbalance of ϵ. The number of parts p can be any number ≥ 1, not necessarily a power of two. The output is a sequence $(A_0, A_1, \ldots, A_{p-1})$ of mutually disjoint subsets of nonzeros, each containing the nonzeros of one processor, with $\cup_{s=0}^{p-1} A_s = A$.

function MatrixPartition(A, p, ϵ)
  if $p > 1$ **then**
    $p_0 := \lfloor p/2 \rfloor$;
    $p_1 := \lceil p/2 \rceil$;
    $q_0 := \lceil \log_2 p_0 \rceil + 1$;
    $q_1 := \lceil \log_2 p_1 \rceil + 1$;
    $(A_0^{\mathrm{r}}, A_1^{\mathrm{r}}) := \mathrm{split}(A, \mathrm{row}, \epsilon/q_0, \epsilon/q_1, p_0/p)$;
    $(A_0^{\mathrm{c}}, A_1^{\mathrm{c}}) := \mathrm{split}(A, \mathrm{col}, \epsilon/q_0, \epsilon/q_1, p_0/p)$;
    $(A_0^{\mathrm{f}}, A_1^{\mathrm{f}}) := \mathrm{split}(A, \mathrm{fine}, \epsilon/q_0, \epsilon/q_1, p_0/p)$;
    $(A_0, A_1) :=$ best of $(A_0^{\mathrm{r}}, A_1^{\mathrm{r}})$, $(A_0^{\mathrm{c}}, A_1^{\mathrm{c}})$, $(A_0^{\mathrm{f}}, A_1^{\mathrm{f}})$;

    $\mathrm{nz}_{\max} := \frac{\mathrm{nz}(A)}{p}(1+\epsilon)$;
    $\epsilon_0 := \frac{\mathrm{nz}_{\max}}{\mathrm{nz}(A_0)} \cdot p_0 - 1$;
    $\epsilon_1 := \frac{\mathrm{nz}_{\max}}{\mathrm{nz}(A_1)} \cdot p_1 - 1$;
    MatrixPartition(A_0, p_0, ϵ_0);
    MatrixPartition(A_1, p_1, ϵ_1);
  else output A;

$\mathrm{nz}_{\max}^{\mathrm{c}}$ nonzeros are handled, each requiring access to a row of at most $\mathrm{nz}_{\max}^{\mathrm{r}}$ nonzeros. The next step has only half the number of columns. If the first merge was successful, meaning that columns merge mainly with columns that have a very similar sparsity pattern, then the number of nonzeros per column hardly increases, while the number of nonzeros per row halves. Thus the next coarsening step requires less than half the work of the first step, and so on. The complete coarsening thus takes at most twice the time of its first step. The KLFM algorithm used in the uncoarsening is somewhat cheaper, but still of the same order. Overall, $\log_2 p$ stages of splitting are needed. The first stage is the most expensive, as the complete matrix is handled. The second stage works on two submatrices with smaller $m, n, \mathrm{nz}_{\max}^{\mathrm{r}}, \mathrm{nz}_{\max}^{\mathrm{c}}$ and because of the form of the complexity formula each submatrix costs less than half the work of the first stage. The overall cost of the second stage is thus at most that of the first stage. For other methods than 1D column partitioning, the complexity formula stays the same.

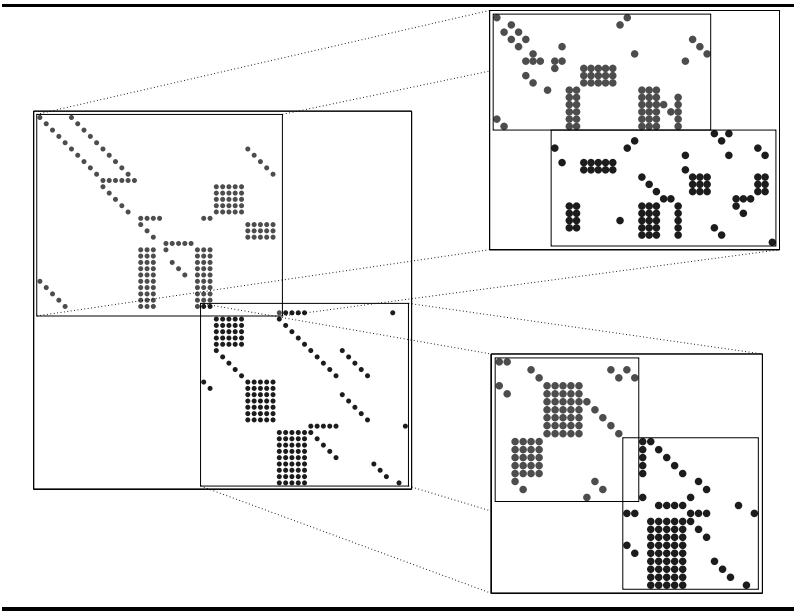

FIGURE 12.6: (See color insert.) Split of the 59×59 matrix `impcol_b` with 312 nonzeros into four parts. The first split (left) is by the fine-grain method; the second split (top right) is by rows; and the third split (bottom right) is again by the fine-grain method. The matrix is permuted correspondingly by moving, after each split, the newly cut rows and columns to the middle and the uncut ones to the left or right, or top or bottom. The matrix structure seen on the left is the (doubly) separated block-diagonal (SBD) structure, see [3].

12.8 Experimental Results

To compare the different 1D and 2D approaches to partitioning, we performed numerical experiments on a set of 10 test matrices, originating in various fields, see Table 12.2. All experiments were performed on an Intel Core i7 860 2.8 GHz processor, under the Ubuntu Linux 10.10 operating system (kernel 2.6.35, x86_64), with 8 Gbyte of RAM.

Table 12.3 presents a comparison of 1D and 2D methods, showing in the 1D case both a row-wise and a column-wise partitioning, and in the 2D case three methods, namely the Mondriaan approach based on orthogonal recursive bisection, the fine-grain approach, and the hybrid approach. For this table, the native hypergraph bipartitioner of Mondriaan was used. Table 12.4 presents a similar comparison, but now with PaToH [15] as a bipartitioner. The bottom line of the tables shows an average performance, obtained by dividing the

TABLE 12.2: Test set of sparse matrices.

Matrix	m	n	nz	Field
`dfl001`	6071	12230	35632	linear programming
`cre_b`	9648	77137	260785	linear programming
`tbdmatlab`	19859	5979	430171	information retrieval
`nug30`	52260	379350	1567800	linear programming
`c98a`	56243	56274	2075889	cryptography
`tbdlinux`	112757	20167	2157675	information retrieval
`stanford`	281903	281903	2312497	web search
`polyDFT`	46176	46176	3690048	polymer self-assembly
`cage13`	445315	445315	7479343	DNA electrophoresis
`stanford_berkeley`	683446	683446	7583376	web search

communication volume by the volume for the 2D Mondriaan method using its own hypergraph bipartitioner, as given in the '2D Mon' column of Table 12.3. The allowed load imbalance defined by Equation (12.1) was set to $\epsilon = 0.03$. Communication volumes in the table represent averages over 25 runs with different random number seeds, where the volume of a run is influenced to some extent by random choices made by the algorithm. An example of randomness appearing in the algorithm is in the initial partitioning, where eight random starting partitionings of the coarsened hypergraph are generated, and the best resulting partitioning is chosen. Furthermore, ties in the algorithm are usually broken by 'flipping a coin.' For debugging purposes, the random number seed can be fixed, but for comparing results of two different methods we retain the randomness and perform a large number of runs.

The 2D methods perform much better than the 1D methods. In Table 12.3, the best 1D method, which is 1D column partitioning, yields on average 21% more communication than the 2D Mondriaan method, and in 24 out of 30 *problem instances*, i.e., pairs (Matrix, p), it beats the 1D row method, and in three instances it performs equally. This preference for column partitioning is a bias of the test set, since partitioning A^T instead of A would have given the opposite conclusion. Taking for each problem instance the best of the 1D row and 1D column results still gives on average 23% more communication than with the 2D Mondriaan method. Similar differences between 1D and 2D results can be seen in Table 12.4.

The 2D Mondriaan method with its own bipartitioner is better than the fine-grain method for all matrices except `cage13`, see Table 12.3. The hybrid method beats both methods in 20 out of 30 problem instances, and the difference with the best of both is small in the other cases. We may conclude that in most cases the hybrid method succeeds in picking the best of both, exactly what it was designed for.

With the PaToH implementation of the hypergraph bipartitioner, we find

TABLE 12.3: Communication volume of p-way partitioning using the Mondriaan package (version 3.11) with its own hypergraph bipartitioner. Average over 25 runs.

Matrix	p	1D row	1D column	2D Mon	2D fine	2D hybrid
dfl001	4	2974	1411	1402	1646	1410
	16	6086	3650	3660	3939	3618
	64	9124	6175	6142	6439	6044
cre_b	4	33293	1402	1379	3498	1393
	16	59910	4142	4153	8889	4122
	64	86092	9353	9312	17876	9276
tbdmatlab	4	13784	15061	10373	13987	10340
	16	56117	41769	26569	44122	26553
	64	152760	93667	52603	96504	58058
nug30	4	284683	67065	65926	109410	65491
	16	355935	150653	148256	258382	145881
	64	376809	250778	246508	439015	244726
c98a	4	162188	117532	100232	140069	100250
	16	652151	351879	239427	379300	239257
	64	1349998	647273	452397	665934	449727
tbdlinux	4	41275	54438	30159	35999	33945
	16	167039	152469	69971	120256	87672
	64	486500	328067	143767	298505	172510
stanford	4	22074	1309	1313	2462	1246
	16	96563	4662	4774	7106	4389
	64	339045	13324	13112	19631	12391
polyDFT	4	8813	8744	8501	12670	8496
	16	39066	39135	37855	51094	37707
	64	83527	83237	75034	113611	74966
cage13	4	118701	118701	112532	97957	99504
	16	267701	267701	245730	204808	208823
	64	495286	495286	428818	357141	359062
stanford_berkeley	4	35276	1626	1673	2367	1169
	16	230519	7219	10108	15184	6307
	64	917105	20473	23789	43413	19373
Avg normalized value		7.80	1.21	1.00	1.53	0.97

TABLE 12.4: Communication volume of p-way partitioning using the Mondriaan package (version 3.11) with PaToH as hypergraph bipartitioner. Average over 25 runs. Note that the 'Mon' column refers to the Mondriaan partitioning method used, based on orthogonal recursive bisection, but the splits of this method are carried out using PaToH.

Matrix	p	1D row	1D column	2D Mon	2D fine	2D hybrid
dfl001	4	2953	1374	1391	1386	1358
	16	5888	3608	3599	3552	3522
	64	8406	6049	6039	6060	5872
cre_b	4	33097	1206	1195	1225	1151
	16	57110	3481	3483	3585	3412
	64	73974	7811	7831	8097	7576
tbdmatlab	4	14078	15052	10666	10841	10391
	16	54397	40554	26511	31329	26219
	64	134093	81681	49238	64680	50810
nug30	4	270049	56809	55810	62323	52675
	16	384826	117883	116083	133336	106562
	64	542912	197808	198609	231081	185720
c98a	4	161633	117691	100096	124396	96455
	16	643899	351613	228064	325374	226643
	64	1341322	646249	417233	578384	408051
tbdlinux	4	42773	52710	35180	24921	24967
	16	166555	147091	77038	81754	73274
	64	443921	308329	145350	186478	149748
stanford	4	8755	934	896	921	826
	16	50281	3330	3169	3345	3004
	64	217976	9801	9643	9066	8359
polyDFT	4	9448	9228	8785	8794	8615
	16	35427	35177	34009	37024	34381
	64	78560	78615	73531	82879	73290
cage13	4	120883	121737	116349	89953	90534
	16	273048	273487	250559	188854	190640
	64	496963	498482	437986	333481	334123
stanford_berkeley	4	3321	1432	1139	1406	1031
	16	134868	5287	4650	5046	4080
	64	579112	15489	14958	14083	12810
Avg normalized value		5.32	1.11	0.90	0.95	0.83

again that 2D Mondriaan performed better than the fine-grain method, although by a much smaller margin. Apparently, the advantage of the more general solution is less important than the positive effect, as a solution heuristic, of forcing the nonzeros of a row (or column) to stay together, which is built into the Mondriaan approach. Here too, the hybrid method usually (i.e., in 24 out of 30 instances) beats the best of both, with at most 3% difference between hybrid and best of both in the other instances.

Comparing Tables 12.3 and 12.4, we see that PaToH is significantly better optimized for the fine-grain method, achieving an average normalized volume of 0.95 vs. 1.53 for the native Mondriaan bipartitioner. For the Mondriaan method, it is also better, 0.90 vs. 1.00. Overall, the best results are obtained with the hybrid method using PaToH as the bipartitioner, with a normalized volume of 0.83. Note that all these results are obtained within the same Mondriaan software framework, so that they can be compared with each other. Using PaToH for the whole partitioning process, i.e., not only for the bipartitioning process, would give slightly different results; see [16] for results using only PaToH, and also for results using Zoltan.

Tables 12.5 and 12.6 present a comparison of the run time of the five methods, using both the native hypergraph bipartitioner of Mondriaan and PaToH. Our main observation here is that the time of the hybrid method is almost the time of the Mondriaan and fine-grain method combined. This can be expected since both methods are tried within the hybrid partitioner. Another observation from both tables is that the time of the Mondriaan method is about the time of the 1D row and 1D column method combined. This means that 1D methods are only useful in terms of computing time if there is a quick way of determining which of the two directions, row or column, has to be chosen, see e.g., the recipe given in [7]. Remarkable is the speed of the 2D Mondriaan method using the PaToH bipartitioner; it is almost three times as fast compared to using the native Mondriaan bipartitioner. It is also 2.4 times faster than the fine-grain method using PaToH. Examining the timings for $p = 4, 16, 64$ in Tables 12.5 and 12.6, we see that the scaling with p in practice is often better than the theoretical upper bound of $\mathcal{O}(\log_2 p)$ given by the time complexity, Equation (12.8).

Table 12.7 presents the actual run time of a parallel sparse matrix–vector multiplication for the largest two test matrices after partitioning using Mondriaan with PaToH as bipartitioner. The purpose of the table is to check whether lower communication volumes indeed lead to lower run times of the parallel program. For `cage13` and $p = 64$, the savings of one third in communication volume shown in Table 12.4 between 1D row and 2D hybrid leads to a decrease in run time of about 26% as shown in Table 12.7. The large savings for `stanford_berkeley` between 1D row and 1D column show up as a decrease of a factor of 2.5 in run time. The overall gain in speed is not directly proportional to the gain in communication volume because there are other factors involved as well, such as computation time and synchronization time; for smaller problems, the latter may actually be dominant. For larger p,

TABLE 12.5: Time (in s) needed to compute a p-way partitioning using the Mondriaan package (version 3.11) with its own hypergraph bipartitioner. Average over 25 runs.

Matrix	p	1D row	1D column	2D Mon	2D fine	2D hybrid
dfl001	4	0.29	0.19	0.47	0.35	0.84
	16	0.38	0.36	0.76	0.64	1.38
	64	0.53	0.53	1.10	0.90	1.94
cre_b	4	1.63	1.58	3.36	2.76	6.17
	16	2.17	2.46	5.23	4.32	9.56
	64	4.04	3.21	6.78	5.69	12.38
tbdmatlab	4	2.84	2.45	5.41	10.09	16.59
	16	4.72	4.38	9.25	14.63	25.86
	64	5.58	5.75	11.24	17.59	30.15
nug30	4	17.45	14.71	40.68	23.71	63.07
	16	15.86	23.27	58.57	42.64	100.10
	64	21.59	29.36	68.28	54.21	116.30
c98a	4	25.30	82.15	104.40	391.60	493.00
	16	38.70	112.10	162.20	482.90	667.70
	64	53.10	123.60	193.90	517.10	745.50
tbdlinux	4	35.73	21.41	61.23	142.50	199.50
	16	52.24	32.30	94.13	190.70	290.30
	64	60.14	39.12	111.90	214.60	336.90
stanford	4	7.93	70.58	79.16	249.60	310.70
	16	13.19	105.20	118.60	390.50	479.90
	64	17.52	123.20	141.70	455.80	565.50
polyDFT	4	7.23	7.11	14.55	49.13	64.13
	16	15.42	15.45	29.15	87.43	117.10
	64	21.25	20.90	39.90	117.50	157.50
cage13	4	54.14	54.24	109.80	121.80	233.20
	16	95.52	95.72	169.00	195.70	372.20
	64	116.60	115.80	227.10	255.70	491.10
stanford_berkeley	4	16.96	288.20	300.50	1454.00	1746.00
	16	31.84	484.10	523.60	2788.00	3140.00
	64	51.03	574.60	632.60	3393.00	3797.00
Avg normalized value		0.39	0.57	1.00	2.19	3.17

TABLE 12.6: Time (in s) needed to compute a p-way partitioning using the Mondriaan package (version 3.11) with PaToH as hypergraph bipartitioner. Average over 25 runs.

Matrix	p	1D row	1D column	2D Mon	2D fine	2D hybrid
dfl001	4	0.06	0.07	0.14	0.11	0.24
	16	0.11	0.20	0.30	0.24	0.50
	64	0.17	0.26	0.44	0.35	0.72
cre_b	4	0.31	1.07	1.36	1.85	3.19
	16	0.50	1.60	2.11	2.75	4.87
	64	0.83	1.99	2.77	3.37	6.16
tbdmatlab	4	1.61	1.30	2.93	6.47	9.28
	16	2.50	2.26	5.07	9.67	15.12
	64	2.88	2.79	6.46	11.56	18.62
nug30	4	4.00	3.89	8.52	8.47	16.65
	16	5.99	6.76	15.05	14.78	29.48
	64	19.65	8.65	22.87	17.84	38.20
c98a	4	9.74	12.83	19.71	50.39	67.54
	16	13.97	25.35	38.83	84.48	112.60
	64	19.87	34.10	52.48	106.40	145.30
tbdlinux	4	16.83	8.96	24.43	75.48	99.05
	16	23.87	14.90	42.14	109.50	149.80
	64	27.66	17.67	53.64	125.30	179.20
stanford	4	3.27	5.77	8.68	28.26	36.64
	16	5.79	11.17	16.07	48.64	62.69
	64	8.27	15.86	22.73	66.05	86.15
polyDFT	4	6.04	6.05	12.16	57.23	68.41
	16	11.85	12.32	23.33	93.86	116.60
	64	15.87	16.41	33.32	120.60	152.10
cage13	4	12.54	12.43	25.19	43.27	67.04
	16	26.57	28.31	44.53	76.49	117.80
	64	36.58	41.24	66.67	102.10	161.70
stanford_berkeley	4	5.84	10.08	15.53	49.34	63.99
	16	12.57	22.37	31.34	102.20	123.80
	64	21.10	36.90	49.00	160.20	186.30
Avg normalized value		0.17	0.19	0.35	0.83	1.16

TABLE 12.7: Time (in ms) of parallel sparse matrix–vector multiplication using the educational program `bsp_mv` from BSPedupack [11] on the Huygens IBM Power6+ parallel computer in Amsterdam, which consists of 1664 dual-core processors running at 4.7 GHz. Here, p denotes the number of cores used. This program has not been optimized; for comparison, the highly optimized sequential program from [3] takes only 18.08 ms for `cage13` and 19.61 ms for `stanford_berkeley`. The matrices were partitioned by using Mondriaan with PaToH as bipartitioner for $\epsilon = 0.03$.

Matrix	p	1D row	1D column	2D Mon	2D fine	2D hybrid
`cage13`	1	372.2	372.2	372.2	372.2	372.2
	4	125.0	124.2	120.7	123.0	124.2
	16	39.5	38.8	37.1	36.6	35.9
	64	18.3	17.0	16.1	13.5	13.5
`stanford_berkeley`	1	552.6	552.6	552.6	552.6	552.6
	4	222.7	171.5	169.3	182.5	171.9
	16	74.4	49.3	71.2	52.9	59.1
	64	49.5	21.2	21.4	22.6	21.6

communication increases and computation decreases, making communication gains more pronounced. Note that partitioning with the 2D hybrid method takes the same time as about 8625 sparse matrix–vector multiplications would take for `stanford_berkeley` with $p = 64$, where we ignore the fact that we use different hardware in our timings. For the simpler 1D row method, the break-even point is already at 1030 iterations. Still, in most cases the extra time spent on partitioning by using a 2D method would pay off.

12.9 Conclusions and Outlook

The main conclusions of this chapter are

- 2D partitioning is superior to 1D partitioning, as it leads to significantly smaller communication volumes in sparse matrix–vector multiplication.
- Both the Mondriaan approach, based on orthogonal recursive bisection, and the fine-grain approach, based on assigning individual nonzeros to processors, have their strong points and will perform better on some classes of matrices. The hybrid approach combines the best of the two and almost always achieves the lowest communication volume.

- The hybrid approach takes longer to compute, about the same time as the Mondriaan and the fine-grain approach together.
- The fastest 2D partitioner is the 2D Mondriaan method with PaToH as a bipartitioner.

Open questions that deserve further investigation are

- In addition to reducing the communication volume, how can we achieve other objectives as well, such as reducing the total number of messages? Note that all data words to be sent from the same source processor to the same destination processor can be packaged into one message. The number of messages is important in certain types of parallel computation, e.g., where message-passing is employed as a model, in particular if the latency for sending a message is high. The aim would be to keep the number of messages far below the theoretical maximum of $2p(p-1)$ for the 2D case.
- Can we partition the matrix and the vectors simultaneously while modeling the computation and communication requirements exactly? In our approach, the vectors are partitioned after the matrix partitioning, but then it may already be too late to do this well. For instance, if balancing the communication is desired, current methods [28] in practice achieve the best solution that can be obtained after the matrix partitioning, but still there can be severe communication imbalance. Uçar and Aykanat [43] perform the two partitionings at the same time, but do not model the communication cost exactly.
- Can we find a recipe for choosing automatically when to use 2D Mondriaan or fine-grain partitioning, based on a few problem statistics? Such a recipe is given for the choice between 1D row, 1D column, 2D fine-grain, and jagged-like methods in [7], based on $m, n, \mathrm{nz}(A)$ and the average and median nonzero counts of the rows and columns. For instance, for rectangular matrices that are far from square, splitting the largest dimension is prescribed, i.e., 1D column partitioning if $m \ll n$. This choice can be made again at every split. Having such a recipe will save partitioning time, while still keeping the same quality as the hybrid partitioning.

Acknowledgments

This work was supported in part by the DOE grant De-FC02-06ER2775 and by the NSF grants CNS-0643969, OCI-0904809, and OCI-0904802.

We thank the Dutch National Computer Facilities foundation and SARA in Amsterdam for providing access to the supercomputer Huygens.

Bibliography

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

[2] Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., and van der Vorst, H., editors, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, SIAM, Philadelphia, PA, 2000.

[3] Yzelman, A. N. and Bisseling, R. H., "Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods," *SIAM Journal on Scientific Computing*, Vol. 31, No. 4, 2009, pp. 3128–3154.

[4] Çatalyürek, Ü. V., *Hypergraph Models for Sparse Matrix Partitioning and Reordering*, Ph.D. thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999, Available at http://www.cs.bilkent.edu.tr/tech-reports/1999/ABSTRACTS.1999.html.

[5] Çatalyürek, Ü. V. and Aykanat, C., "A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices," *Proceedings Eighth International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001)*, IEEE Press, Los Alamitos, CA, 2001, p. 118.

[6] Çatalyürek, Ü. V. and Aykanat, C., "A Hypergraph-Partitioning Approach for Coarse-Grain Decomposition," *Proceedings Supercomputing 2001*, ACM Press, New York, 2001, p. 42.

[7] Çatalyürek, Ü. V., Aykanat, C., and Uçar, B., "On Two-Dimensional Sparse Matrix Partitioning: Models, Methods, and a Recipe," *SIAM Journal on Scientific Computing*, Vol. 32, No. 2, 2010, pp. 656–683.

[8] Vastenhouw, B. and Bisseling, R. H., "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Review*, Vol. 47, No. 1, 2005, pp. 67–95, Available at http://www.staff.science.uu.nl/~bisse101/Mondriaan.

[9] Vuduc, R., Demmel, J. W., and Yelick, K. A., "OSKI: A library of automatically tuned sparse matrix kernels," *J. Phys. Conf. Series*, Vol. 16, 2005, pp. 521–530.

[10] Çatalyürek, Ü. V. and Aykanat, C., "Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 7, 1999, pp. 673–693.

[11] Bisseling, R. H., *Parallel Scientific Computation: A Structured Approach using BSP and MPI*, Oxford University Press, Oxford, UK, 2004.

[12] Valiant, L. G., "A Bridging Model for Parallel Computation," *Communications of the ACM*, Vol. 33, No. 8, 1990, pp. 103–111.

[13] Caldwell, A. E., Kahng, A. B., and Markov, I. L., "Improved Algorithms for Hypergraph Bipartitioning," *Proceedings Asia and South Pacific Design Automation Conference*, ACM Press, New York, 2000, pp. 661–666.

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

[15] Çatalyürek, Ü. V. and Aykanat, C., *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*, Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at http://bmi.osu.edu/~umit/software.htm, 1999.

[16] Devine, K. D., Boman, E. G., Heaphy, R., Bisseling, R. H., and Catalyurek, U. V., "Parallel Hypergraph Partitioning for Scientific Computing," *Proceedings IEEE International Parallel and Distributed Processing Symposium 2006*, IEEE Press, 2006.

[17] Trifunović, A. and Knottenbelt, W. J., "Parallel multilevel algorithms for hypergraph partitioning," *Journal of Parallel and Distributed Computing*, Vol. 68, No. 5, 2008, pp. 563–581.

[18] Bui, T. and Jones, C., "A heuristic for reducing fill-in in sparse matrix factorization," *Proceedings Sixth SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, Philadelphia, PA, 1993, pp. 445–452.

[19] Lengauer, T., *Combinatorial algorithms for integrated circuit layout*, John Wiley and Sons, Chichester, UK, 1990.

[20] Bui, T. N. and Jones, C., "Finding good approximate vertex and edge partitions is NP-hard," *Information Processing Letters*, Vol. 42, 1992, pp. 153–159.

[21] Hendrickson, B. and Leland, R., "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations," *SIAM Journal on Scientific Computing*, Vol. 16, No. 2, 1995, pp. 452–469.

[22] Karypis, G. and Kumar, V., "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, Vol. 20, No. 1, 1998, pp. 27–58, 359–392.

[23] Karypis, G. and Kumar, V., "Parallel Multilevel k-Way Partitioning Scheme for Irregular Graphs," *SIAM Review*, Vol. 41, No. 2, 1999, pp. 278–300.

[24] Walshaw, C. and Cross, M., "JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview," *Mesh Partitioning Techniques and Domain Decomposition Techniques*, edited by F. Magoules, Civil-Comp Ltd., 2007, pp. 27–58.

[25] Pellegrini, F. and Roman, J., "SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs," *Proceedings High Performance Computing and Networking Europe*, Vol. 1067 of *Lecture Notes in Computer Science*, Springer, Berlin, 1996, pp. 493–498.

[26] Chevalier, C. and Pellegrini, F., "PT-Scotch: a tool for efficient parallel graph ordering," *Parallel Computing*, Vol. 34, No. 6-8, 2008, pp. 318–331.

[27] Hendrickson, B. and Kolda, T. G., "Graph Partitioning Models for Parallel Computing," *Parallel Computing*, Vol. 26, No. 12, 2000, pp. 1519–1534.

[28] Bisseling, R. H. and Meesen, W., "Communication balancing in parallel sparse matrix-vector multiplication," *Electronic Transactions on Numerical Analysis*, Vol. 21, 2005, pp. 47–65, Special Issue on Combinatorial Scientific Computing.

[29] Uçar, B. and Aykanat, C., "Encapsulating Multiple Communication-Cost Metrics in Partitioning Sparse Rectangular Matrices for Parallel Matrix-Vector Multiplies," *SIAM Journal on Scientific Computing*, Vol. 25, No. 6, 2004, pp. 1837–1859.

[30] Saad, Y. and Schultz, M. H., "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems," *SIAM Journal on Scientific and Statistical Computing*, Vol. 7, No. 3, 1986, pp. 856–869.

[31] Bisseling, R. H., "Parallel Iterative Solution of Sparse Linear Systems on a Transputer Network," *Parallel Computation*, edited by A. E. Fincham and B. Ford, Vol. 46 of *The Institute of Mathematics and its Applications Conference Series. New Series*, Oxford University Press, Oxford, UK, 1993, pp. 253–271.

[32] Hendrickson, B., Leland, R., and Plimpton, S., "An efficient parallel algorithm for matrix-vector multiplication," *International Journal of High Speed Computing*, Vol. 7, No. 1, 1995, pp. 73–88.

[33] Lewis, J. G., Payne, D. G., and van de Geijn, R. A., "Matrix-Vector Multiplication and Conjugate Gradient Algorithms on Distributed Memory Computers," *Proceedings IEEE Scalable High Performance Computing Conference*, IEEE Press, 1994, pp. 542–550.

[34] Lewis, J. G. and van de Geijn, R. A., "Distributed Memory Matrix-Vector Multiplication and Conjugate Gradient Algorithms," *Proceedings Supercomputing'93*, ACM Press, New York, 1993, pp. 484–492.

[35] Nicol, D. M., "Rectilinear partitioning of irregular data parallel computations," *Journal of Parallel and Distributed Computing*, Vol. 23, No. 2, 1994, pp. 119–134.

[36] Ogielski, A. T. and Aiello, W., "Sparse matrix computations on parallel processor arrays," *SIAM Journal on Scientific Computing*, Vol. 14, No. 3, 1993, pp. 519–530.

[37] Saltz, J. H., Petition, S. G., Berryman, H., and Rifkin, A., "Performance Effects of Irregular Communication Patterns on Massively Parallel Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 13, No. 2, 1991, pp. 202–212.

[38] Kutluca, H., Kurc, T. M., and Aykanat, C., "Image-Space Decomposition Algorithms for Sort-First Parallel Volume Rendering of Unstructured Grids," *Journal of Supercomputing*, Vol. 15, 2000, pp. 51–93.

[39] Manne, F. and Sørevik, T., "Partitioning an Array onto a Mesh of Processors," *PARA '96: Proceedings Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization*, Springer-Verlag, London, UK, 1996, pp. 467–477.

[40] Pınar, A. and Aykanat, C., "Sparse Matrix Decomposition with Optimal Load Balancing," *Proceedings International Conference on High Performance Computing*, Dec 1997, pp. 224–229.

[41] Fiduccia, C. M. and Mattheyses, R. M., "A Linear-Time Heuristic for Improving Network Partitions," *Proceedings of the 19th IEEE Design Automation Conference*, IEEE Press, Los Alamitos, CA, 1982, pp. 175–181.

[42] Kernighan, B. W. and Lin, S., "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, Vol. 29, 1970, pp. 291–307.

[43] Uçar, B. and Aykanat, C., "Revisiting Hypergraph Models for Sparse Matrix Partitioning," *SIAM Review*, Vol. 49, No. 4, 2007, pp. 595–603.