A TWO-DIMENSIONAL DATA DISTRIBUTION METHOD FOR PARALLEL SPARSE MATRIX-VECTOR MULTIPLICATION

BRENDAN VASTENHOUW* AND ROB H. BISSELING†

Abstract. A new method is presented for distributing data in sparse matrix-vector multiplication. The method is two-dimensional, tries to minimise the true communication volume, and also tries to spread the computation and communication work evenly over the processors. The method starts with a recursive bipartitioning of the sparse matrix, each time splitting a rectangular matrix into two parts with a nearly equal number of nonzeros. The communication volume caused by the split is minimised. After the matrix partitioning, the input and output vectors are partitioned with the objective of minimising the maximum communication volume per processor. Experimental results of our implementation, Mondriaan, for a set of sparse test matrices show a reduction in communication compared to one-dimensional methods, and in general a good balance in the communication work.

Key words. matrix partitioning, matrix-vector multiplication, parallel computing, recursive bipartitioning, sparse matrix

AMS subject classifications. 05C65, 65F10, 65F50, 65Y05

1. Introduction. Sparse matrix-vector multiplication lies at the heart of many iterative solvers for linear systems and eigensystems. In these solvers, a multiplication $\mathbf{u} := A\mathbf{v}$ has to be carried out repeatedly for the same $m \times n$ sparse matrix A, but each time for a different input vector $\mathbf{v}$. On a distributed-memory parallel computer, efficient multiplication requires a suitable distribution of the data and the associated work. In particular, this requires distributing the sparse matrix and the input and output vectors over the p processors of the parallel computer such that each processor has about the same number of nonzeros and such that the amount of communication is minimal.

The natural parallel algorithm for sparse matrix-vector multiplication with an arbitrary distribution of matrix and vectors consists of the following four phases:

1. Each processor sends its components v_j to those processors that possess a nonzero a_{ij} in column j.
2. Each processor computes the products $a_{ij}v_j$ for its nonzeros a_{ij}, and adds the results for the same row index i. This yields a set of contributions u_{is}, where s is the processor identifier, $0 \leq s < p$.
3. Each processor sends its nonzero contributions u_{is} to the processor that possesses u_i.
4. Each processor adds the contributions received for its components u_i, giving $u_i = \sum_{t=0}^{p-1} u_{it}$.

Processors are assumed to synchronise globally between the phases.

In this paper, we propose a new general scheme for distributing the matrix and the vectors over the processors that enables us to obtain a good load balance and minimise the communication cost in the algorithm above. A good distribution scheme has the following characteristics:

- It tries to spread the matrix nonzeros evenly over the processors, to minimise the maximum amount of work of a processor in phase 2.

*Image Sciences Institute, University Hospital Utrecht, PO Box 85500, 3508 GA Utrecht, The Netherlands (brendan@isi.uu.nl)

†Mathematical Institute, Utrecht University, PO Box 80010, 3508 TA Utrecht, The Netherlands (Rob.Bisseling@math.uu.nl)

- It tries to minimise the true total number of communications, the *communication volume*, and not a different metric. (If the same vector component v_j is needed twice by a processor, for instance because of nonzeros a_{ij} and $a_{i'j}$, it is sent only once by the algorithm, and the cost function of the distribution scheme should reflect this.)
- It tries to spread the communications evenly over the processors, both with respect to sending and receiving, to minimise the maximum number of communication operations of a processor in phases 1 and 3.
- It tries to partition the matrix in both dimensions, e.g. by splitting it into rectangular blocks. As a result, the elements of a column need to be distributed over only $\sqrt{p}$ processors (assuming p is square) instead of all the p processors of the parallel computer. This limits the number of destination processors and hence communications of a vector component v_j in phase 1 to $\sqrt{p}-1$, provided v_j resides on one of the processors that needs it. In the same way, this also limits the number of communications in phase 3. Although a one-dimensional distribution has the advantage that it removes one or two phases, e.g. phases 3 and 4 in the case of a row distribution, the price to be paid is high: the rows (or columns) must be distributed over a larger number of processors, and the number of communications for a vector component can reach $p-1$.

In recent years, much work has been done in this area. Commonly, the matrix partitioning problem has been formulated as a graph partitioning problem, where (in the row-oriented version) a vertex i represents matrix row i together with the vector components u_i and v_i, and where an edge (i,j) represents a nonzero a_{ij}, and the aim is to minimise the number of cut edges. An edge (i,j) is *cut* if vertices i and j are assigned to different processors. This one-dimensional method is the basis of the partitioning algorithms implemented in software such as Chaco [24] and Metis [28], which has found widespread use. The success of these partitioning programs can be attributed to their incorporated efficient multilevel bipartitioning algorithms. Multilevel methods, first proposed by Bui and Jones [9], coarsen a graph by merging vertices at several successive levels until the remaining graph is sufficiently small, then partition the result and finally uncoarsen it, projecting back the partitioning and refining it at every level. The partitioning itself is done sequentially; a parallel version of Metis, ParMetis [29], has recently been developed.

Hendrickson [21] criticises the graph partitioning approach because it can handle only square symmetric matrices, imposes the same partitioning for the input and output vectors, and because it does not necessarily try to minimise the communication volume, nor the number of messages, nor the maximum communication load of a processor. Hendrickson and Kolda [22] show that these disadvantages hold for all applications of graph partitioning in parallel computing, and not only for sparse matrix-vector multiplication. They note that in many applications we have been fortunate, because the effect of these disadvantages has been limited. This is because many applications originate in differential equations discretised on a grid, where the number of neighbours of a grid point is limited, so that the number of cut edges may not be too far from the true communication volume. In more complex applications, we may not be so lucky. Bilderback [5] shows for five different graph partitioning packages that the number of cut edges varies significantly between the processors, pointing to potential for improvement of the communication load balance. Hendrickson and Kolda [23] present an alternative, the bipartite graph model, where the rows

of an $m \times n$ matrix correspond to a set of m row vertices, the columns to a set of n column vertices, and a nonzero element a_{ij} corresponds to an edge (i, j) between row vertex i and column vertex j. The row and column vertices are each partitioned into p sets. This determines the distribution of the input and output vectors. The matrix distribution is a one-dimensional row distribution that conforms to the partitioning of the row vertices. The vertices are partitioned by a multilevel algorithm that tries to minimise the number of cut edges while keeping the difference in work between processors less than the work of a single row or column in the matrix. This model can handle nonsymmetric square matrices and rectangular matrices and it does not impose the same distribution for the input and output vectors.

Çatalyürek and Aykanat [10] present a multilevel partitioning algorithm that models the communication volume exactly by using a hypergraph formulation. A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ consists of a set of vertices $\mathcal{V} = \{0, \ldots, n-1\}$ and a set of *hyperedges* $\mathcal{N} = \{n_0, \ldots, n_{m-1}\}$, also called *nets*, which are subsets of $\mathcal{V}$. In their row-net model, each row of an $m \times n$ matrix corresponds to an hyperedge and each column to a vertex. (They also present a similar column-net model.) Çatalyürek and Aykanat assume that $m = n$, and that the vector distribution is determined by the matrix distribution: for the row-net model, components u_j, v_j are assigned to the same processor as matrix column j. The problem they solve is how to partition the vertices into sets $\mathcal{V}_0, \ldots, \mathcal{V}_{p-1}$ such that the computation load is balanced and the total cost of the cut hyperedges is minimal. A *cut hyperedge* n_i intersects at least two sets $\mathcal{V}_s$. The cost of a cut hyperedge is the number of sets it intersects, minus one. This is exactly the number of processors that has to send a nonzero contribution to u_i in phase 3. (For the row-net model, phase 1 vanishes.) The advantage of this approach is that it tries to minimise the true communication volume and not an approximation of the volume. This indeed leads to less communication: experimental results for the PaToH program (Partitioning Tool for Hypergraphs) show a 35 per cent reduction in volume for a set of test matrices from the Rutherford-Boeing collection [13, 14] and some linear programming matrices, compared to the Metis implementation of graph partitioning. Çatalyürek and Aykanat [10] also tested hMetis, a hypergraph-based version of Metis, and found that PaToH and hMetis produce partitionings of equal quality but that PaToH partitions about three times faster than hMetis.

Hu, Maguire, and Blake [27] present a similar algorithm for the purpose of reordering a nonsymmetric matrix by row and column permutations into bordered block-diagonal form, implemented in MONET (Matrix Ordering for minimal NET-cut). This form facilitates subsequent parallel numerical factorisation. The algorithm tries to assign matrix rows to processors in such a way that the number of cut columns is minimal.

Both the standard graph partitioning approach and the hypergraph approach produce one-dimensional matrix partitionings that can be used together with a two-phase matrix-vector multiplication. Two-dimensional matrix partitionings have also been proposed, but these are typically less optimised, and are often used with variants of the four-phase matrix-vector multiplication that exploit sparsity only for computation but not for communication. Such methods rely mainly on the strength of two-dimensional partitioning as a means of reducing communication. Fox et al. [17, Chapt. 21] present a four-phase algorithm for dense matrix-vector multiplication that uses a square block distribution of the matrix. In work on the NAS parallel conjugate gradient benchmark, Lewis and van de Geijn [32] and Hendrickson, Leland, and Plimpton [25] describe algorithms that are suitable for dense matrices or

relatively dense irregular sparse matrices. These algorithms exploit the sparsity for computation, but not for communication. Lewis and van de Geijn compare their two-dimensional algorithms with a one-dimensional algorithm, and find gains of a factor of 2.5 on an Intel iPSC/860 hypercube. Ogielski and Aiello [33] partition the rows and columns of a matrix A by permuting them randomly into a matrix PAQ and then splitting the rows into blocks of rows and the columns into blocks of columns. This gives a two-dimensional partitioning with an expected good load balance. Pinar and Aykanat [34] split the matrix first into blocks of rows, and then split each block independently into blocks of subcolumns, taking only computation load balance into account. The rows and columns are not permuted. This gives a two-dimensional row-wise jagged partitioning.

Bisseling [6] presents a two-dimensional algorithm aimed at a square mesh of transputers that exploits sparsity both for computation and communication. The matrix is distributed by the square cyclic distribution. Vector components are distributed over all the processors; communications are done within chains of processors of minimal length. For instance, v_j is broadcast to a set of processors $P(s,t)$, $s_{\min} \leq s \leq s_{\max}$ in processor column t, where the range is chosen as small as possible. Bisseling and McColl [7] improve this algorithm so that only truly needed communications are performed; they achieve this by transferring the algorithm from the restricted model of a square mesh with store-and-forward routing to the more general bulk synchronous parallel model. They analyse the communication of various distributions using the maximum number of sends or receives per processor as cost function. The matrix distribution is *Cartesian*, i.e., defined by partitioning the matrix rows into M sets I_s, the columns into N sets J_t and assigning the $p = MN$ Cartesian products $I_s \times J_t$ to the processors. The vector distribution is the same as that of the matrix diagonal. Experiments for several classes of matrices show that tailoring the distribution to the matrix at hand yields better distributions than matrix-independent schemes. This work makes no attempt, however, to find the best data distribution for an arbitrary sparse matrix, as is done by general-purpose multilevel partitioning algorithms.

In recent work, Çatalyürek and Aykanat [11] extend their previous one-dimensional hypergraph-based partitioning method for square matrices to two dimensions. They produce a Cartesian matrix distribution by first partitioning the rows into M sets with an approximately equal number of nonzeros, and then partitioning the columns trying to spread the nonzeros in all the row sets simultaneously by solving a multi-constraint partitioning problem. The distribution of the vectors $\mathbf{u}$ and $\mathbf{v}$ is identical, and equal to the distribution of the matrix diagonal. For the choice $M = N = \sqrt{p}$, the maximum number of messages per processor decreases to $2(\sqrt{p}-1)$, compared to the $p-1$ messages of a one-dimensional distribution. This is an advantage on a computer with a high startup cost for messages, in particular for relatively small matrices. In their experiments, the number of messages indeed decreases significantly and the communication volume stays about the same, both compared to a one-dimensional distribution.

Berger and Bokhari [3] present a recursive bisection-based strategy for partitioning nonuniform two-dimensional grids. The partitioning divides the grid alternatingly in horizontal and vertical directions, with the aim of achieving a good balance in the computational work. Recursive bisection is a well-known optimisation technique, which has been used for instance in parallel circuit simulation, see Fox et al. [17, Chapt. 22]. This technique can also be used to partition matrices, as has been done by Romero and Zapata [35] to achieve good load balance in sparse-matrix vector

multiplication.

In the present work, we bring the techniques discussed above together, hoping to obtain a more efficient sparse matrix-vector multiplication. Our primary focus is the general case of a sparse rectangular matrix with input and output vectors that can be distributed independently. The original motivation of our work is the design of a parallel web-search engine based on latent semantic indexing; see [4] for a recent review of such information retrieval methods. The indexing is done by computing a singular value decomposition using Lanczos bidiagonalisation [20], which requires the repeated multiplication of a rectangular sparse matrix and a vector. We view our distribution problem exclusively as a partitioning problem and do not take the mapping of the parts to the processors of a particular parallel machine with a particular communication network into account. Tailoring the distribution to a machine would harm portability. More generic approaches are possible (see e.g. Walshaw and Cross [39]), but adopting such an approach would make our algorithm more complicated.

The remainder of this paper is organised as follows. Section 2 presents a two-dimensional method for partitioning the sparse matrix that attempts to minimise the communication volume. Section 3 presents a method for partitioning the input and output vectors that attempts to balance the communication volume between the processors. Section 4 discusses possible adaptation of our methods to special cases such as square matrices or square symmetric matrices. Section 5 presents experimental results of our program Mondriaan for a set of test matrices. Finally, Section 6 draws conclusions and outlines possible future work.

2. Matrix partitioning. We make the following assumptions. The matrix A has size $m \times n$, with $m, n \geq 1$. The matrix is *sparse*, i.e., many of its elements are zero. Since for the purpose of partitioning we are only interested in the sparsity pattern of the matrix (and not in the numerical values), we assume that elements a_{ij}, with $0 \leq i < m$ and $0 \leq j < n$, are either 0 or 1. The input vector $\mathbf{v}$ is a dense vector of length n and the output vector $\mathbf{u}$ is a dense vector of length m. We do not exploit possible sparsity in the vectors. The parallel computer has $p = 2^q$ processors, where $q \geq 0$, each with its own local memory.

We sometimes view a matrix as just a set of index pairs, writing

$$A = \{(i,j) \ : \ 0 \leq i < m \ \wedge \ 0 \leq j < n\}. \tag{2.1}$$

The number of nonzeros in A is

$$nz(A) = |\{(i,j) \in A \ : \ a_{ij} = 1\}|. \tag{2.2}$$

A subset $B \subset A$ is a subset of index pairs. A *k-way partitioning* of A is a set $\{A_0, \ldots, A_{k-1}\}$ of nonempty, mutually disjoint subsets of A that satisfy $\bigcup_{r=0}^{k-1} A_r = A$.

The communication volume of the natural parallel algorithm for sparse matrix-vector multiplication is the total number of data elements that are sent in phases 1 and 3. This volume depends on the data distribution chosen for the matrix and the vectors. From now on, we assume that vector component v_j is assigned to one of the processors that owns a nonzero a_{ij} in matrix column j, if such a nonzero exists; otherwise, column j is empty and does not cause communication. Such an assignment is always better than assignment to one of the other processors, because this would cause an extra communication. We also assume that u_i is assigned to one of the processors that owns a nonzero a_{ij} in matrix row i. Under these two assumptions, the communication volume is independent of the vector distributions. This motivates the following matrix-based definition.

DEFINITION 2.1. *Let A be an $m \times n$ sparse matrix and let $A_0, \ldots, A_{k-1}$ be mutually disjoint subsets of A, where $k \geq 1$. Define*

$$\begin{aligned} p_i &= p_i(A_0, \ldots, A_{k-1}) \\ &= |\{r : 0 \leq r < k \wedge (\exists j : 0 \leq j < n \wedge a_{ij} = 1 \wedge (i,j) \in A_r)\}|, \end{aligned} \tag{2.3}$$

i.e. the number of subsets that has a nonzero in row i of A, for $0 \leq i < m$, and

$$\begin{aligned} q_j &= q_j(A_0, \ldots, A_{k-1}) \\ &= |\{r : 0 \leq r < k \wedge (\exists i : 0 \leq i < m \wedge a_{ij} = 1 \wedge (i,j) \in A_r)\}|, \end{aligned} \tag{2.4}$$

i.e. the number of subsets that has a nonzero in column j of A, for $0 \leq j < n$. Define $p_i' = \max(p_i - 1, 0)$ and $q_j' = \max(q_j - 1, 0)$. Then the communication volume for the subsets $A_0, \ldots, A_{k-1}$ is defined as

$$V(A_0, \ldots, A_{k-1}) = \sum_{i=0}^{m-1} p_i' + \sum_{j=0}^{n-1} q_j'.$$

Note that the volume function V is also defined when the k mutually disjoint subsets do not form a k-way partitioning. If $k = p$ and the subsets form a p-way partitioning, and if we assign each subset to a processor, then $V(A_0, \ldots, A_{p-1})$ is exactly the communication volume in the parallel algorithm above. This is because every v_j is sent from its owner to all the other q_j' processors that possess a nonempty part of column j and every u_i is the sum of a local contribution by its owner and contributions received from the other p_i' processors. An important property of the volume function is the following.

THEOREM 2.2. *Let A be an $m \times n$ matrix and let $A_0, \ldots, A_{k-1}$ be mutually disjoint subsets of A, where $k \geq 2$. Then*

$$V(A_0, \ldots, A_{k-1}) = V(A_0, \ldots, A_{k-3}, A_{k-2} \cup A_{k-1}) + V(A_{k-2}, A_{k-1}). \tag{2.5}$$

Proof. It is sufficient to prove (2.5) with V replaced by p_i', for $0 \leq i < m$, and by q_j', for $0 \leq j < n$, from which the result follows by summing. We will only treat the case of the p_i'; the case of the q_j' is similar. Let i be a row index. We have to prove that

$$p_i'(A_0, \ldots, A_{k-1}) = p_i'(A_0, \ldots, A_{k-3}, A_{k-2} \cup A_{k-1}) + p_i'(A_{k-2}, A_{k-1}). \tag{2.6}$$

If A_{k-2} or A_{k-1} has a nonzero in row i, we can substitute $p_i' = p_i - 1$ in the terms of the equation. The resulting equality is easy to prove, starting at the rhs, because

$$\begin{aligned} & p_i(A_0, \ldots, A_{k-3}, A_{k-2} \cup A_{k-1}) - 1 + p_i(A_{k-2}, A_{k-1}) - 1 \\ &= p_i(A_0, \ldots, A_{k-3}) + 1 - 1 + p_i(A_{k-2}, A_{k-1}) - 1 \\ &= p_i(A_0, \ldots, A_{k-3}, A_{k-2}, A_{k-1}) - 1, \end{aligned} \tag{2.7}$$

which is the lhs. If A_{k-2} and A_{k-1} do not have a nonzero in row i, the lhs and the rhs of (2.6) both equal $p_i'(A_0, \ldots, A_{k-3})$. □

This theorem is a generalisation to arbitrary subsets of a remark by Çatalyürek and Aykanat [10] on the case where each subset A_r consists of a set of complete

matrix columns. The theorem implies that to see how much extra communication is generated by splitting a subset of the matrix, we only have to look at that subset.

We also define a function that gives the maximum amount of computational work of a processor in the local matrix-vector multiplication. For simplicity, we express the amount of work in multiplications (associated with matrix nonzeros); we ignore the associated additions.

DEFINITION 2.3. *Let A be an $m \times n$ matrix and let $A_0, \ldots, A_{k-1}$ be mutually disjoint subsets of A, where $k \geq 1$. Then the maximum amount of computational work for the subsets $A_0, \ldots, A_{k-1}$ is*

$$W(A_0, \ldots, A_{k-1}) = \max_{0 \leq r < k} nz(A_r).$$

The function V describes the cost of phases 1 and 3 of the parallel algorithm; the function W that of phase 2. The cost of phase 4 is ignored in our description. Usually this cost is much less than that of the other phases: the total number of additions by all the processors in phase 4 is bounded by V, because every contribution added has been received previously in phase 3, and addition is usually much cheaper than communication. Minimising V thus minimises an upper bound on the cost of phase 4. Balancing the communication load in phase 3 thus balances the computation load in phase 4.

Our aim in this section is to design an algorithm for finding a p-way partitioning of the matrix A that satisfies the load-balance criterion

$$W(A_0, \ldots, A_{p-1}) \leq (1 + \epsilon)\frac{W(A)}{p}, \tag{2.8}$$

and that has low communication volume $V(A_0, \ldots, A_{p-1})$. Here, $\epsilon > 0$ is the *load imbalance parameter*, a constant that expresses the relative amount of load imbalance that is permitted. A small value of ϵ means that the load balance must be close to perfect.

First, we examine the simplest possible partitioning problem, the case $p = 2$. One way to split the matrix is to assign complete columns to A_0 or A_1. This has the advantage that ${q_j}' = 0$ for all j, thus causing no communication of vector components v_j. (Splitting a column j by assigning nonzeros to different processors would automatically cause a communication.) If two columns j and j' have a nonzero in the same row i, i.e., $a_{ij} = a_{ij'} = 1$, then these columns should preferably be assigned to the same processor; otherwise ${p_i}' = 1$. The problem of assigning columns to two processors is exactly the 2-way hypergraph partitioning problem defined by the hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ with $\mathcal{V} = \{0, \ldots, n-1\}$ the set of vertices (representing the matrix columns) and $\mathcal{N} = \{n_0, \ldots, n_{m-1}\}$ the set of hyperedges where $n_i = \{j : 0 \leq j < n \wedge a_{ij} = 1\}$. The problem is to partition the vertices into two sets $\mathcal{V}_0$ and $\mathcal{V}_1$ such that the number of cut hyperedges is minimal and such that the load balance criterion (2.8) is satisfied. Here, a cut hyperedge n_i intersects both $\mathcal{V}_0$ and $\mathcal{V}_1$ and its cost is one. To calculate the work load, every vertex j is weighted by the number of nonzeros c_j of column j, giving

$$\sum_{j \in \mathcal{V}_r} c_j \leq (1 + \epsilon) \cdot \frac{1}{2} \cdot \sum_{j \in \mathcal{V}} c_j, \quad \text{for } r = 1, 2. \tag{2.9}$$

Methods developed for this problem [10] are directly applicable to our situation. Such methods are necessarily heuristic, since the general hypergraph partitioning problem

is NP-complete [31]. To capture these methods, we define a hypergraph splitting function h on a matrix subset A by

$$(A_0, A_1) \leftarrow h(A, \mathit{sign}, \epsilon).$$

The output is a pair of mutually disjoint subsets (A_0, A_1) with $A_0 \cup A_1 = A$ that satisfies $W(A_0, A_1) \leq (1 + \epsilon)W(A)/2$. If $\mathit{sign} = 1$, the columns of the subset are partitioned (i.e., elements of A from the same matrix column are assigned to the same processor); if $\mathit{sign} = -1$, the rows are partitioned. We do not specify the function h further, but just assume that such a function is available and that it works well, partitioning optimally or close to the optimum.

Splitting a matrix into two parts by assigning complete columns (or rows) has the advantages of simplicity and absence of communication in phase 1. Still, it may sometimes be beneficial to allow a column j to be split, for instance because its first half resembles a column j' and the other half resembles a column j''. Assigning the first half to the same processor as j' and the second half to the same as j'' can save more than one communication in phase 3. In this approach, individual elements are assigned to processors instead of complete columns (or rows). To keep our overall algorithm simple, we do not follow this approach.

Next, we consider the case $p = 4$. Aiming at a two-dimensional partitioning we could first partition the columns into sets J_1 and J_2 , and then the rows into sets I_0 and I_1. This will split the matrix into four submatrices, identified with the Cartesian products $I_0 \times J_0$, $I_0 \times J_1$, $I_1 \times J_0$, and $I_1 \times J_1$. This distribution, like most matrix distributions currently in use, is Cartesian. The four-processor case reveals a serious disadvantage of Cartesian distributions: the same partitioning of the rows must be applied to both sets of columns. A good row partitioning for the columns of J_0 may be bad for the columns of J_1, and vice versa. This will often lead to a compromise partioning of the rows. Dropping the Cartesian constraint enlarges the set of possible partitionings and hence gives better solutions. Therefore, we can partition both parts separately. Theorem 2.2 implies that this can even be done independently, because

$$V(A_0, A_1, A_2, A_3) = V(A_0 \cup A_1, A_2 \cup A_3) + V(A_0, A_1) + V(A_2, A_3), \tag{2.10}$$

where the parts are denoted by A_0, A_1, A_2, A_3 with $A_0 \cup A_1$ the first set of columns and $A_2 \cup A_3$ the second set. To partition $A_0 \cup A_1$ in the best way, we do not have to consider the partitioning of $A_2 \cup A_3$.

The advantage of independent partitioning is illustrated by Fig. 2.1. For ease of understanding, the matrix shown in the figure has been split by a simple scheme that is solely based on minimising the computational load imbalance and that partitions the matrix greedily into contiguous blocks. (In general, however, we also try to minimise communication and we allow partitioning into noncontiguous matrix parts.) Note that independent partitioning leads to a much better load balance, giving a maximum of 80 nonzeros per processor, or $\epsilon \approx 2.5\%$, compared to the 128 nonzeros, or $\epsilon \approx 64\%$, for the Cartesian case. The total communication volume is about the same, 66 vs. 63. It is clear that independent partitioning gives much better possibilities to improve the load balance or minimise the communication cost.

The method used to obtain a four-way partitioning from a two-way partitioning can be applied repeatedly, resulting in a recursive algorithm for sparse matrix partitioning, Algorithm 1. This algorithm is greedy because it tries to bipartition the current matrix in the best possible way, without taking subsequent bipartitionings into account. When $q = \log_2 p$ bipartitioning levels remain, we allow in principle a

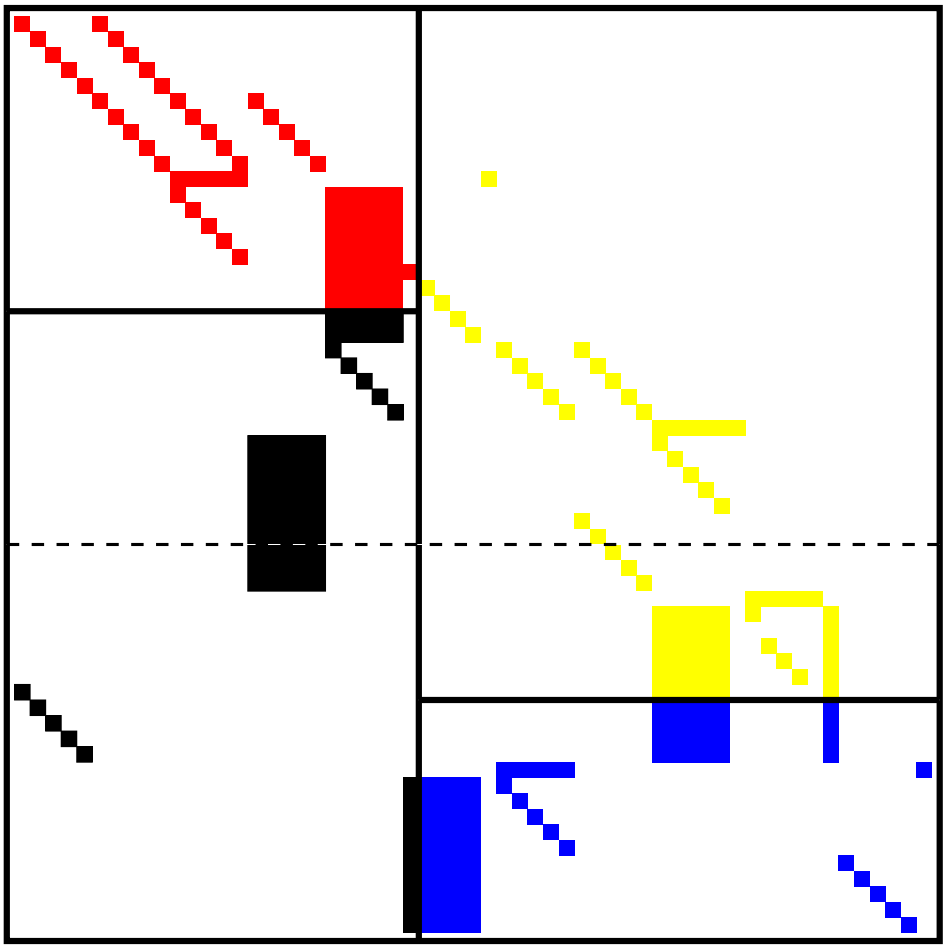

FIG. 2.1. *Block distribution of the* 59×59 *matrix* `impcol_b` *with 312 nonzeros from the Rutherford-Boeing collection [13, 14] over four processors, depicted by the colours red, yellow, blue, and black. The matrix is first partitioned into two blocks of columns, and then each block is partitioned independently into two blocks of rows, as shown by the bold lines. The resulting number of nonzeros of the processors is 76, 76, 80, and 80, respectively. Also shown is a Cartesian distribution, where both column blocks are split in the same way, as indicated by the dashed line. Now, the number of nonzeros is 126, 28, 128, and 30, respectively.*

load imbalance of ϵ/q for each bipartitioning. The value ϵ/q is used once, but then the value for the remaining levels is adapted to the outcome of the current bipartitioning. For instance, the part with the smallest amount of work will have a larger allowed imbalance than the other part. The corresponding value of ϵ is based on the maximum number of nonzeros *maxnz* allowed per processor. The partitioning direction is chosen alternatingly.

Many variations on this basic algorithm are possible, for instance regarding the load balance criterion and the partitioning direction. We could allow a value $\delta_p \epsilon$ with $\delta_p \leq 1$ as load imbalance parameter for the current bipartitioning, instead of $\epsilon / \log_2 p$. The partitioning direction need not be chosen alternatingly: it could be determined greedily, i.e., by trying both row and column partitionings and taking the best result. This doubles the cost of the partitioning. A cheaper variant would be to

MatrixPartition(A, *sign*, p, ϵ)
input: A is an $m \times n$ matrix.
sign is the sign of the first bipartitioning to be done.
p is the number of processors, $p = 2^q$ with $q \geq 0$.
ϵ: allowed load imbalance, $\epsilon > 0$.
output: p-way partitioning of A satisfying criterion (2.8).

if $p > 1$ **then**
$q := \log_2 p$;
$(A_0, A_1) := h(A, sign, \epsilon/q)$;
$maxnz := \frac{nz(A)}{p}(1 + \epsilon)$;
$\epsilon_0 := \frac{maxnz}{nz(A_0)} \cdot \frac{p}{2} - 1$;
$\epsilon_1 := \frac{maxnz}{nz(A_1)} \cdot \frac{p}{2} - 1$;
MatrixPartition($A_0, -sign, \epsilon_0, p/2$);
MatrixPartition($A_1, -sign, \epsilon_1, p/2$);
else output A;

Algorithm 1: Recursive bipartitioning algorithm

split the largest dimension, i.e., take $sign = 1$ if $m \leq n$ and $sign = -1$ otherwise. We expect this strategy to choose the best direction in most cases, but without trying both. The latter strategy tries to partition into square submatrices. In fact, this should be the objective of a true two-dimensional partitioning, rather than trying to maintain the original aspect ratio m/n by partitioning each dimension the same number of times, which is done by the alternating-direction strategy. The choice of splitting direction is particularly important for matrices with a large aspect ratio. (The strategy of splitting the largest dimension is motivated by the ideal case of dense matrices that are partitioned into $M \times N$ equal blocks of size $m/M \times n/N$, where $MN = p$, with a corresponding vector partitioning into m/p components u_i and n/p components v_j per processor. In this case, the communication volume equals $m(N-1) + n(M-1) \approx mN + nM$, which is minimal if $m/M = n/N$. For random sparse matrices with a high density the same reasoning holds, and for other sparse matrices we expect similar behaviour of the communication volume as a function of the aspect ratio.)

For the alternating-direction strategy, we can guarantee an upper bound on the number of processors q_j that holds a matrix column j. The bound is $q_j \leq \sqrt{p}$, for $0 \leq j < n$, if p is an even power of two. This is because each level of partitioning with $sign = -1$ causes at most a doubling of the maximum number of processors that holds a matrix column, whereas each level with $sign = 1$ does not affect this maximum. Similarly, $q_j \leq \sqrt{2p}$ if p is an odd power of two and the first bipartitioning has $sign = -1$; otherwise the bound is $q_j \leq \sqrt{p/2}$.

The result of the recursive bipartitioning algorithm is a p-way partitioning of the matrix A. Processor $P(s)$ obtains a subset $I_s \times J_s$ of the original matrix, where $I_s \subset \{0, \ldots m-1\}$ and $J_s \subset \{0, \ldots n-1\}$. This subset is itself a submatrix, but its rows and columns are not necessarily consecutive. Figures 2.2 and 2.3 show the result of such a partitioning from two different view points.

Figure 2.2 gives the *global view* of the partitioning, showing the original matrix

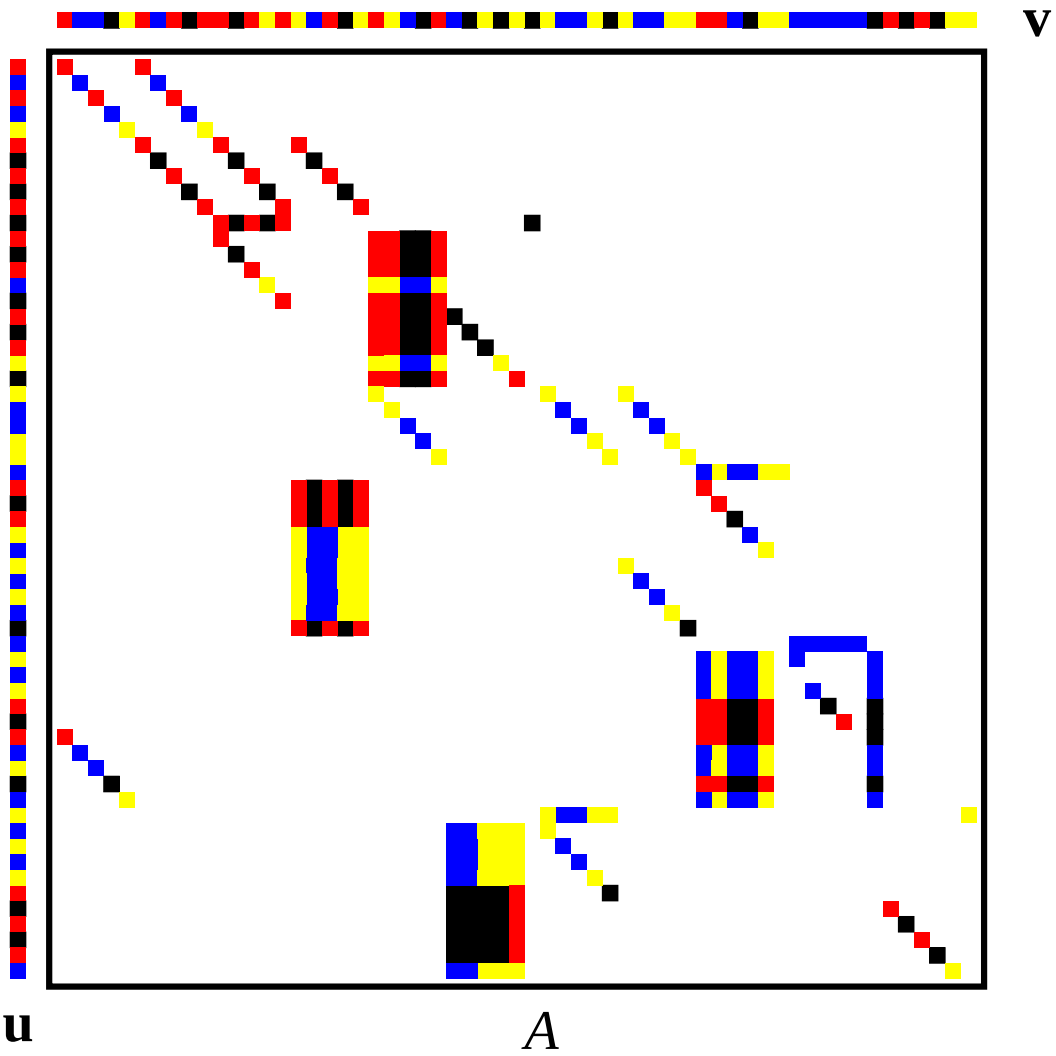

FIG. 2.2. *Global view of the distribution of the matrix* `impcol_b` *over four processors by the recursive bipartitioning algorithm with the greedy best-direction strategy. The processors are depicted by the colours red, yellow, blue, and black; they possess 79, 78, 79, and 76 nonzeros, respectively. Also given is a distribution of the input and output vectors that assigns a component v_j to one of the processors that owns a nonzero in matrix column j, and assigns u_i to one of the owners of a nonzero in row i. This distribution is obtained by the vector distribution algorithm given in Section 3.*

and vectors with the processor assignment for each element. This view reveals for instance that the four blocks of nonzeros from the original matrix `impcol_b` are each distributed over all four processors. The total communication volume is 76, which is slightly more than the volume of 66 of the simple block-based distribution method used in Figure 2.1. (Here, the block-based method is lucky because of the presence of the four large blocks. In general, the recursive bipartitioning method is much better.) In the global view, it is easy to see where communication takes place: every matrix column that has nonzeros in a different colour than the vector component above it causes communication, and similar for rows.

Figure 2.3 gives the *local view*, showing the submatrices stored in the processors; these submatrices fit exactly in the space of the original matrix. This view displays the structure of the local submatrices. Here, it is also easy to see where communication takes place: every hole in the vector distribution represents a vector component that is stored on a remote processor, thus causing communication. We have removed empty local rows and columns, thus reducing the size of $I(s) \times J(s)$, to emphasise the true local structure. A good splitting function h leads to many empty rows and columns. For instance, the first split leads to 16 empty columns above the splitting line, which means that all the nonzeros in the corresponding matrix columns are located below the line, thus causing no communication. (In an implementation, empty rows and columns can be deleted from the data structure. In a figure, we have some freedom where to place them.)

3. Vector partitioning. After the matrix distribution has been chosen with the aim of minimising communication volume under the computational load balance constraint, we can now choose the vector distribution freely to achieve other aims as well, such as a good balance in the communication or an even spread of the vector components, as long as we assign input vector components to one of the processors that have nonzeros in the corresponding matrix column, and output vector components to one of the processors that have nonzeros in the corresponding matrix row.

We assume that the input vector and output vector can be assigned independently, which will usually be the case for rectangular, nonsquare matrices. Because the communication pattern in phase 3 of the computation of $A\mathbf{v}$ is the same as that in phase 1 of the computation of $A^T\mathbf{u}$, except for a reversal of the roles of sends and receives, we can partition the output vector for multiplication by A using the method for partitioning input vectors, but then applied to A^T. Therefore, we will discuss only the partitioning of the input vector.

Define V_s as the set of indices j corresponding to vector components v_j assigned to processor $P(s)$. The number of sends of $P(s)$ in phase 1 equals

$$N_{\mathrm{s}}(s) = \sum_{j \in V_s} {q_j}'(A_0, \ldots, A_{p-1}) \tag{3.1}$$

and the number of receives equals

$$N_{\mathrm{r}}(s) = |\{j : 0 \leq j < n \wedge j \notin V_s \wedge (\exists i : 0 \leq i < m \wedge a_{ij} = 1 \wedge (i,j) \in A_s)\}|. \tag{3.2}$$

A vector partitioning method could attempt to minimise:

1. $\max_{0 \leq s < p} N_{\mathrm{s}}(s)$, the maximum number of sends of a processor.
2. $\max_{0 \leq s < p} N_{\mathrm{r}}(s)$, the maximum number of receives of a processor.
3. $\max_{0 \leq s < p} |V_s|$, the maximum number of components of a processor.

The first two aims are equally important, for the following reasons. First, from a computer hardware point of view, congestion at a communication link to a processor can occur both because of outgoing and incoming communication. This justifies trying to minimise both. Second, as said above, we partition the output vector of $A\mathbf{v}$ by partitioning it as the input vector of $A^T\mathbf{u}$. If our partitioning method would only minimise the number of sends of the input vector but not the number of receives, this could lead to many receives in phase 1 of the multiplication by A^T and hence to many sends in phase 3 of the multiplication by A. Third, sometimes the output vector $A\mathbf{v}$ is subsequently multiplied by A^T, either immediately or after some vector operations. (This happens for instance in Lanczos bidiagonalisation and in the conjugate gradient

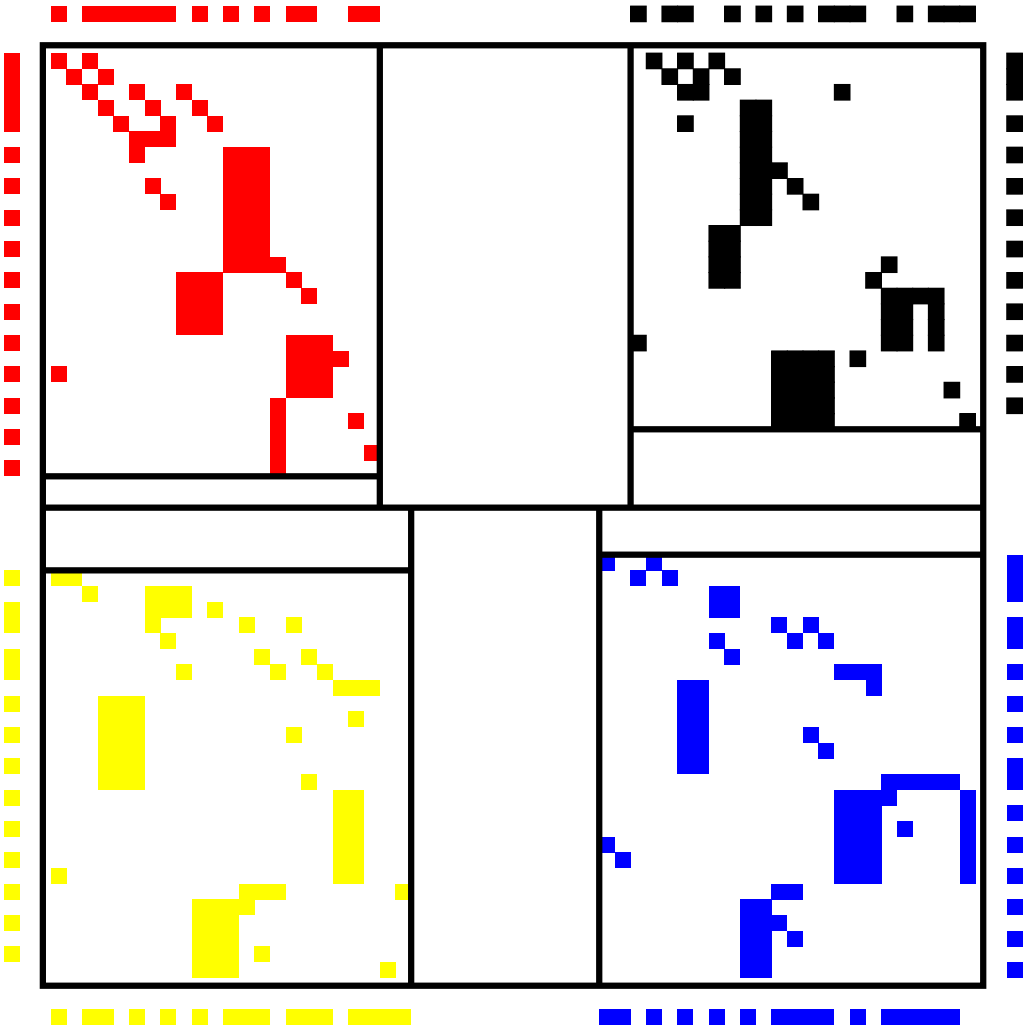

FIG. 2.3. *Local view of the distribution from Figure 2.2. The greedy best-direction strategy chooses for this matrix to partition first in the horizontal direction (i.e., with sign* $= -1$*) and then to partition the resulting parts both in the vertical direction (i.e., with sign* $= 1$*). As a result, the red, yellow, blue, and black processors possess submatrices of size* 27×21*,* 26×23*,* 27×24*, and* 24×22*, respectively. Empty local rows and columns have been removed; they are collected in separate blocks. The local components of the input and output vectors are shown along the border of the local submatrix, above or below the submatrix for the input vector and to the left or right for the output vector.*

method applied to the normal equations [20].) This multiplication can be carried out using the stored matrix A in its present distribution, and the result $A^T A\mathbf{v}$ can be delivered in the distribution of $\mathbf{v}$. The algorithm for the multiplication by A^T using the stored matrix A is similar to the multiplication by A, except that the direction is reversed. The communication pattern of phase 1 for A^T is the same as that of phase 3 for A, except that the sends and receives are interchanged. A data partitioning for multiplication by A that minimises both the number of sends and receives is therefore also optimal for multiplication by A^T.

The third aim, balancing the number of vector components, is less important,

because it does not influence the time of the matrix-vector multiplication itself. It only affects the time of vector operations such as norm or inner product computations, or DAXPYs, in the remaining part of iterative solvers. Often the vector operations are much less time consuming than the matrix-vector multiplication. We will use load balance in vector operations only to break ties when our primary objectives are equally met. (If desired, the maximum number of components can be included in a cost function, with a weight factor reflecting its relative importance in the iterative solver concerned.)

Consider the assignment of a vector component v_j to a processor. If $q_j = 0$, v_j can be assigned to an arbitrary processor. If $q_j = 1$, v_j has to be assigned to the processor that has all the nonzeros of column j. In both cases, no communication occurs. Now assume that $q_j \geq 2$. Assigning v_j to a processor increases the number of sends of that processor by $q_j{}' = q_j - 1$ and the number of receives of the $q_j - 1$ other processors that have part of matrix column j by one. If we take as the cost incurred by a processor the sum of the number of sends and receives, i.e. $N_{\rm s}(s) + N_{\rm r}(s)$ for $P(s)$, we see that the sum for the sender increases by $q_j - 1 \geq 1$ and for the receivers by one. This suggests a greedy assignment of v_j to the processor with the smallest sum so far, among those that have part of column j. This heuristic assigns the inevitable cost one to all the processors concerned, but tries to avoid, as much as possible, to increase the maximum sum.

An alternative would be to take the maximum of the number of sends and receives as the cost incurred by a processor, i.e. $\max(N_{\rm s}(s), N_{\rm r}(s))$ for $P(s)$. This treats sends and receives equally and it even has the advantage that it reflects the actual architecture of a processor and its communication links more closely: links are usually bidirectional meaning that incoming and outgoing communication can take place simultaneously. Unfortunately, this cost function does not lead to a simple heuristic and furthermore it is not very sensitive to gradual improvements because sometimes the effect of a component assignment is hidden by the cost function to be minimised: for instance if $N_{\rm s}(s) > N_{\rm r}(s)$ for the receivers $P(s)$ and $N_{\rm s}(s') + q_j - 1 \leq N_{\rm r}(s')$ for the sender $P(s')$, then the current cost remains unchanged in every processor, even though more communication takes place. Because of this, we expect the sum to be a better optimiser than the maximum. Furthermore, the sum provides an upper bound on the maximum which is at most twice the maximum, so that a good spread of the communications (i.e., sends and receives) over the processors implies that the communication time is within a factor of two from the optimal time. We expect sends and receives to average out, so that most likely the results will be even better. For these reasons, we base our vector distribution algorithm on the sum cost-function.

Our vector partitioning algorithm is presented as Algorithm 2. In step 1 of the algorithm, the sums are initialised to the number of inevitable communications, one send or receive per nonempty column part. Initialising the sums before assigning components has the advantage that all inevitable communications are taken into account from the first moment that choices must be made. Processors with many such communications will be assigned fewer components during the algorithm. In step 3, the increment of $q_j - 2$ represents the extra communications of the sender. In step 4, where $q_j = 2$, the sums are not increased anymore. Now, an attempt is made to balance the number of sends with the number of receives. Each time a choice between sending from $P(s)$ to $P(s')$ or vice versa has to be made, this is done on the basis of the current values of $N_{\rm s}(s), N_{\rm r}(s), N_{\rm s}(s'), N_{\rm r}(s')$. The component v_j is assigned to $P(s)$ if $N_{\rm s}(s) + N_{\rm r}(s') \leq N_{\rm s}(s') + N_{\rm r}(s)$, and to $P(s')$ otherwise. This gives rise to

VectorPartition($A_0, \ldots, A_{p-1}, \mathbf{v}, p$)
input: $A_0, \ldots, A_{p-1}$ is a p-way partitioning of a sparse $m \times n$ matrix A.
$\mathbf{v}$ is a vector of length n.
p is the number of processors, $p \geq 1$.
output: p-way partitioning of $\mathbf{v}$.

1. **for** $s := 0$ **to** $p - 1$ **do**
   $sum(s) := |\{j : 0 \leq j < n \wedge q_j \geq 2 \wedge$
   $(\exists i : 0 \leq i < m \wedge a_{ij} = 1 \wedge (i,j) \in A_s)\}|$;
2. **for** $j := 0$ **to** $n - 1$ **do**
   if $q_j = 1$ **then**
   Assign v_j to unique owner of nonzeros in column j
3. **for all** $j : 0 \leq j < n \wedge q_j \geq 3$ **do**
   Assign v_j to $P(s)$ with current lowest $sum(s)$;
   $sum(s) := sum(s) + q_j - 2$;
4. **for all** $j : 0 \leq j < n \wedge q_j = 2$ **do**
   Assign v_j to one of the two owners, trying to balance
   the number of sends with the number of receives;
5. **for** $j := 0$ **to** $n - 1$ **do**
   if $q_j = 0$ **then**
   Assign v_j to $P(s)$ with current lowest number of components.

Algorithm 2: Vector partitioning algorithm

a communication in the least busy send-receive direction. In step 5, the components with $q_j = 0$ represent empty columns and can be assigned to every desired processor. Such columns are unlikely to occur in practice. The final result of the algorithm is an assignment of vector components to processors with $\max_{0 \leq s < p}(N_{\mathrm{s}}(s) + N_{\mathrm{r}}(s))$ minimised.

The order of the assignments in steps 3 and 4 may influence the quality of the resulting vector distribution. Therefore, we have left the order open by using a **for all** statement. In our implementation of step 3, we handle the columns with $q_j \geq 3$ in random order. An alternative would be to handle them in order of increasing q_j, because this would let us choose from more processors at the end of the algorithm. This freedom can better be utilised at the end, when more is known about the accumulated sums. Another possibility is to handle the columns in order of decreasing q_j. This is motivated by the desire in a greedy algorithm to assign large values first, which is particularly important for widely varying values. In our greedy vector distribution algorithm, however, the range of values of q_j is limited because of the upper bound $q_j \leq \sqrt{p}$ (for p an even power of two, in the alternating-direction strategy), and thus we expect the advantage to be small. Note that the difference with the standard greedy binning algorithm by decreasing value is that here we cannot choose from all bins (i.e., processors), but only from a subset of q_j bins. In practice, we found little difference in outcome between the random, increasing, and decreasing strategies.

4. Square matrices. In this section, we discuss the special case where the matrix is square and the input and output vector distribution must be chosen the same. This extra constraint makes it more difficult to balance the communication, and some-

times it may even lead to an increase in communication volume.

First, we consider a square nonsymmetric matrix. Iterative algorithms such as GMRES [36], QMR [18], BiCG [16], and Bi-CGSTAB [37] target this type of matrix. These algorithms are most conveniently carried out in parallel if all vectors involved are distributed in the same way, to facilitate vector operations such as norm and inner product computations and DAXPYs. The matrix partitioning can be done as before, but the vector partitioning must be modified to treat the input and output vector in the same way. This implies that the partitioning of $\mathbf{v}$ determines the communication in both phases 1 and 3. We cannot balance these phases separately any more. The vector partitioning algorithm is a straightforward modification of Algorithm 2, where a sum now represents the total sum for phase 1 and phase 3, and a component v_j is now assigned to a processor in the intersection of the owner set of column j and the owner set of row j. If the preceding matrix partitioning has been done by Algorithm 1, then a processor $P(s)$ in the intersection owns a submatrix $I_s \times J_s$ with $(j, j) \in I_s \times J_s$. Because the submatrices are disjoint, there can only be one submatrix containing (j, j), and hence the intersection contains at most one processor. If furthermore $a_{jj} = 1$, then the intersection contains exactly one processor, namely the owner of a_{jj}; otherwise, the intersection may be empty.

If the intersection is empty, each of the $p_j + q_j$ processors involved can be chosen as the owner of v_j, but the communication volume increases by one. This is a consequence of the fact that we cannot simultaneously satisfy the assumption from Section 2 that v_j is assigned to a processor that holds nonzeros in matrix column j, and u_j is assigned to a processor that holds nonzeros in matrix row j. At the time of the vector distribution it is too late to prevent this extra communication. We may, however, increase the chances of obtaining a nonempty intersection by slightly modifying the matrix partitioning. Following Çatalyürek and Aykanat [10], we add dummy nonzeros a_{jj} to the matrix diagonal before the matrix is partitioned, to make it completely nonzero. We exclude dummy nonzeros from nonzero counts for the purpose of computational load balancing. Most likely, a dummy nonzero a_{jj} attracts other (genuine) nonzeros both from row j and from column j to its processor during the matrix partitioning; in that case the resulting intersection is nonempty. If this does not happen, the intersection is empty and we still must perform the extra communication. Since the dummies are irrelevant for the vector partitioning, we can delete them at the end of the matrix partitioning.

In the nonsymmetric square case, the transposed matrix A^T can be applied using the stored matrix A, at the same communication cost as for A, as discussed in Section 3 for the rectangular case. This is useful in iterative algorithms such as QMR and BiCG that require multiplication of a vector or related vectors by both A and A^T.

Next, we consider a square symmetric matrix, which is the target of algorithms such as conjugate gradients [26]. Here, input and output vectors should also be distributed in the same way. Provided sufficient memory is available to store the complete matrix A, and not only the part below or on the main diagonal, we may choose to ignore the symmetry when determining the matrix distribution, treating a symmetric matrix in the same way as a square nonsymmetric matrix. Our matrix bipartitioning is biased towards assigning nonzeros a_{ij} and a_{ji} to the same processor: it prefers to assign nonzero a_{ij} to the same processor as the (genuine or dummy) nonzero a_{jj} and it also prefers to assign a_{ji} to that processor. To save some memory, nonzeros a_{ij} and a_{ji} that are assigned to the same processor can be represented by one entry in the output of the matrix partitioning; this entry should be marked as

representing two elements. The same can be done in the local data structure of a processor executing the matrix-vector multiplication. An additional advantage is that the data structure needs to be accessed only once per pair of nonzeros, thus saving some operations.

To impose the constraint that a_{ij} and a_{ji} must be assigned to the same processor would limit the number of solutions to the optimisation problem. In principle, there is no need to do this, except perhaps when available memory space is tight. The two-dimensional nature of our partitioning scheme does not lead to benefits from symmetry, and instead we may benefit more from the flexibility of decoupling the assignment of a_{ij} from that of a_{ji}.

As an interesting alternative, we could execute algorithm 1 on a matrix A' obtained from A by deleting one nonzero of each pair (a_{ij}, a_{ji}) on input, adding it back on output, and assigning it to the same processor as its partner. This *symmetric partitioning* method has the advantage that phase 1 and phase 3 have the same communication pattern, although with sends and receives reversed, so that they are balanced simultaneously by the vector partitioning. Thus, we only have to balance phase 1. Furthermore, the set of owners of row j equals that of column j, so that the intersection is much larger, giving better possibilities for balancing the communication. The resulting communication volume for the symmetrically partitioned matrix A is at most twice that of the matrix A' partitioned by recursive bipartitioning. The multiplication factor may indeed be less than two, because some communications may be saved: a component v'_i sent to a nonzero $a_{j'i'}$ from A' may be the same as a component v_j sent to a nonzero a_{ij} from $A-A'$. The disadvantage of symmetric partitioning is that it does not take the complete set of communication obligations into account when trying to minimise the communication volume in the matrix partitioning.

The standard recursive bipartitioning method pays a price for imposing the constraint of an identical input and output distribution. Minimising the maximum communication volume of a processor does not guarantee minimising it in the separate phases 1 and 3. Still, the inequality

$$\max_s(V_s^1 + V_s^3) \leq \max_s V_s^1 + \max_s V_s^3 \leq 2 \cdot \max_s(V_s^1 + V_s^3) \tag{4.1}$$

gives a lower and upper bound on the total communication cost $\max_s V_s^1 + \max_s V_s^3$ of the matrix-vector multiplication algorithm. In practice, we expect the cost to be close to the lower bound. It is possible to improve the balance of the separate phases, and also the balance between the number of sends and receives, for instance by postprocessing the vector distribution, moving components from busy processors to less busy ones. We have not explored this further, but it may be worth while to develop heuristics for this purpose.

5. Experimental results.

5.1. Implementation. We have implemented Algorithm 1, the recursive bipartitioning of the matrix, and Algorithm 2, the vector partitioning, in a program called Mondriaan[1]. The hypergraph bipartitioning function h which bipartitions a matrix has been implemented as a multilevel algorithm, similar to the bipartitioning in PaToH [10]. For column bipartitioning, our implementation is as follows. Empty rows and columns are removed from the matrix before the bipartitioning starts.

[1]The program Mondriaan is named after the Dutch painter Piet Mondriaan (1872-1944) who is renowned for his colourful rectangle-based compositions.

First, in the coarsening phase, the matrix is reduced in size by merging columns in pairs. An unmarked column j is picked and its *neighbouring columns* are determined, i.e., those columns j' with a nonzero $a_{ij'}$ such that a_{ij} is also nonzero. The unmarked column j' with the largest number of such nonzeros is chosen as the match for j. The resulting merged column has a nonzero in row i if a_{ij} or $a_{ij'}$ is nonzero. The amount of work represented by the new column is the sum of the amounts of its constituent columns (initially, before the coarsening, the amount of work of a column equals its number of nonzeros). Both j and j' are then marked and a successful matching is registered. To prevent dominance of a single column, a match is forbidden if it would yield a column with more than 20% of the total amount of work. If no unmarked neighbouring column exists, then j is marked and registered as unmatched at this level. This process is repeated until all columns are marked. (This matching scheme is the same as Heavy Connectivity Matching [10].) We found it advantageous to choose the columns j in order of decreasing number of nonzeros. The matrix could be further reduced in size by deleting singleton rows (i.e., rows with one nonzero), since the content of such a row will not influence the matchmaking any more; this optimisation is not included in our implementation. As a result, the matrix size will be nearly halved. The coarsening is repeated until the matrix is sufficiently small; we choose as our stopping criterion a size of at most 200 columns or a coarsening phase that only reduces the number of columns by less than 5%. The coarsening phase requires both row-wise and column-wise access to the matrix. Therefore, it is convenient to use as data structure both compressed row storage (CRS) and compressed column storage (CCS), but without numerical values.

Second, the small matrix produced by the coarsening phase is bipartitioned using eight runs with different initial balanced partitions of the Kernighan-Lin algorithm [30] in the faster Fiduccia-Mattheyses version [15], which we denote by KL-FM. In this algorithm, columns are moved from one matrix part to the other based on their gain value, i.e., the difference between the number of cut rows after and before a move. Here, a *cut row* is a row with nonzeros in both parts. For the sake of brevity, we will omit the details.

Third, in the uncoarsening phase, the matrix is increased in size at successive levels, each time separating the columns that were matched at the current level, in first instance assigning them to the same processor as before the separation. After the separation at a level is finished, KL-FM is run once to refine the partitioning.

5.2. Test matrices. We have tested version 1.0 of Mondriaan to check the quality of the partitioning produced, using a test set of publicly available sparse matrices, supplemented with a few of our own matrices (which will also be made available). Table 5.1 presents the rectangular (nonsquare) matrices; Table 5.2 the square matrices without structural symmetry; and Table 5.3 the structurally symmetric matrices (with $a_{ij} \neq 0$ if and only if $a_{ji} \neq 0$). In the following, we will call these matrices rectangular, square, and symmetric. Note that structural symmetry is relevant here and not numerical symmetry ($a_{ij} = a_{ji}$), because the sparsity pattern determines the communication requirements and the amount of local computation. The matrices in the tables are sorted by increasing number of nonzeros. The number of nonzeros given is the total number of explicitly stored entries, irrespective of their numerical value. Thus we include entries that happen to be numerically zero. We make one exception: to facilitate comparison with results in other work [23], we removed 27003 explicitly stored zeros from the matrix `memplus`, leaving 99147 entries that are numerically nonzero. The number of nonzeros is for the complete matrix (below, on,

name	rows	columns	nonzeros	application area
well1850	1850	712	8758	surveying
dfl001	6071	12230	35632	linear programming
gemat1	4929	10595	47369	power flow optimisation
cre_b	9648	77137	260785	linear programming
tbdmatlab	19859	5979	430171	information retrieval
nug30	52260	379350	1567800	linear programming
tbdlinux	112757	20167	2157675	information retrieval

TABLE 5.1
Properties of the rectangular test matrices.

name	rows/ columns	diagonal nonzeros	nonzeros	application area
impcol_b	59	17	312	chemical engineering
west0381	381	1	2157	chemical engineering
gemat11	4929	13	33185	power flow optimisation
memplus	17758	17758	99147	circuit simulation
onetone2	36057	9090	227628	circuit simulation
lhr34	35152	102	764014	chemical engineering

TABLE 5.2
Properties of the square test matrices.

and above the main diagonal), and this holds also in the symmetric case.

The matrices well1850, gemat1, impcol_b, west0381, gemat11, bcsstk32, and bcsstk30 were obtained from the Rutherford-Boeing collection [13, 14]; the matrix memplus from the Matrix Market [8]; dfl001 and cre_b (part of the Netlib LP collection [19]), nug30, onetone2, lhr34, and finan512 were obtained from the University of Florida collection [12]; hyp_200_2_1 is a matrix generated by the MLIB package [7] representing a five-point Laplacian operator on a 200×200 grid with periodic boundaries. The matrix tbdmatlab is a term-by-document matrix used for testing our web search application; it represents the 5979 English-language documents in HTML format of the Matlab 5.3 CD-ROM, containing 48959 distinct terms, of which 19859 are used as keywords (the other terms are stopwords). The nonzeros represent scaled term frequencies in the documents. The matrix tbdlinux is a term-by-document matrix describing the documentation of the SuSE Linux 7.1 operating system. The matrix cage10 [38] is a stochastic matrix describing transition probabilities in the cage model of a DNA polymer of length 10 moving in a gel under the influence of an electric field.

5.3. Communication volume and balance. Table 5.4 presents the total communication volume for the partitioning of the rectangular test matrices by using the Mondriaan program with six different direction-choosing strategies: a one-dimensional strategy that always chooses to partition in the row direction; a strategy that always chooses the column direction; a strategy that alternates between the two directions, but starts with the row direction; an alternating strategy that starts with the column direction; a ratio-based strategy that chooses to partition in the row direction if the number of nonempty rows is larger than the number of nonempty columns, and vice versa; the best-direction strategy that tries both directions and greedily chooses the best. The number of processors ranges between 2 and 64; the computational load

name	rows/ columns	nonzeros	application area
cage10	11397	150645	DNA electrophoresis
hyp_200_2_1	40000	200000	Laplacian operation
finan512	74752	596992	portfolio optimisation
bcsstk32	44609	2014701	structural engineering
bcsstk30	28924	2043492	structural engineering

TABLE 5.3
Properties of the structurally symmetric test matrices. All diagonal elements are nonzero.

imbalance specified is $\epsilon = 0.03$, which is the value used in the experiments reported in [10, 11]. In a few cases, our program was not able to achieve the specified load balance; the corresponding results are omitted.

The main conclusion that can be drawn from Table 5.4 is that the best-direction strategy is the best in the majority of cases (29 out of 42 problem instances, i.e. matrix/p combinations). In general, this strategy is about as good as the best of the purely one-dimensional strategies, except for the term-by-document matrices where it is much better than both. The ratio-based strategy is almost as good as the best-direction strategy and is best in 12 out of 42 problem instances. The alternating-direction strategies perform less well; they suffer from the disadvantage that they force partitioning in unfavourable directions. Based on this comparison (and others), we have made the best-direction strategy the default of our program.

Note that a table entry gives the result of one run of our program, with a random number seed set at a fixed default value. (Using different seeds would give slightly different results, but our overall conclusions still hold.) This explains for instance why we have more than two different results for $p = 2$ (e.g. for dfl001, we have 681, 862, and 1481 as possible outcomes, where both 681 and 862 correspond to a column partitioning). Slight fluctuations also explain the exceptional decrease in communication volume when going from $p = 8$ to $p = 16$ for the matrix well1850. Note that these two program runs are completely different, for instance because the partitioning with $p = 8$ is allowed to reach a computational imbalance of 3% after the third split level, whereas the partitioning with $p = 16$ is allowed less (since it has one more split level to go).

The total communication volume is perhaps the most important metric, and sometimes it completely determines the communication time of the matrix-vector multiplication on a particular computer (e.g. on a simple cluster of PCs connected by a bus-like network). Reducing the communication volume is always desirable. On many architectures, however, it is also important to balance the communication, and indeed our algorithm tries to spread the communication duties evenly over the processors as part of the vector partitioning. Table 5.5 shows the communication balance for the rectangular test matrices partitioned using the best-direction strategy. For each phase, the table gives the average communication volume per processor (rounded upwards), $\lceil V/p \rceil$, the maximum number of data sent per processor, $\max_{0 \leq s < p} N_{\mathrm{s}}(s)$, and the maximum number received, $\max_{0 \leq s < p} N_{\mathrm{r}}(s)$. In the ideal case these three numbers are equal and the communication is perfectly balanced. The average is the best value that can be achieved for the maximum number of sends and receives. For $p = 2$, perfect balance is actually achieved for all rectangular matrices, due to the preference we give to sending data in the least busy direction when $p_i = 2$ or $q_j = 2$.

name	p	1D row	1D col	2D alt row	2D alt col	2D ratio	2D best dir
well1850	2	**36**	405	**36**	405	**36**	**36**
	4	102	849	523	522	102	**98**
	8	**169**	1806	631	1510	**169**	173
	16	326	2890	1357	1507	326	**307**
	32	**642**		1936	2897	**642**	683
	64		5544	3729	3487	2155	**2141**
dfl001	2	1481	**681**	1481	**681**	**681**	862
	4	2954	1599	2128	2205	1599	**1413**
	8	4731	2561	4190	3729	2561	**2406**
	16	5967	3654	4875	5110	3654	**3636**
	32	7506	4945	6509	5892	4945	**4930**
	64	9307	6344	7227	7987	6382	**6249**
gemat1	2	2826	**167**	2826	**167**	**167**	169
	4	4345	**194**	3123	2895	**194**	260
	8	5126	520	4365	3045	520	**498**
	16		783	4771	4729	783	**684**
	32	5698	1330	5779	5231	1330	**1310**
	64		2301	6457	6714	2301	**2194**
cre_b	2	19328	**672**	19328	**672**	**672**	764
	4	37988	**1978**	22975	15748	**1978**	2031
	8	48105	**3166**	36093	19034	**3166**	3391
	16	59084	4896	38801	31372	4896	**4616**
	32	72456	**6208**	52358	40158	**6208**	6755
	64	83887	**9251**	53479	54006	**9251**	9413
tbdmatlab	2	**5056**	6438	**5056**	6438	**5056**	**5056**
	4	14650	14949	11305	11298	14650	**11005**
	8	30982	26804	20757	19863	20791	**17792**
	16	56923	42291	29361	29143	29656	**27735**
	32	98791	62410	45536	40601	42324	**40497**
	64	152309	92598	56606	57113	56146	**51594**
nug30	2	190734	**26453**	190734	**26453**	**26453**	26519
	4	287073	59242	210060	187871	59242	**57445**
	8	333084	**86344**	305428	210919	**86344**	92241
	16	357115	127661	330123	319686	127661	**122657**
	32	368312	176200	375189	346195	176200	**169880**
	64	377939	**211720**	400189	415270	**211720**	219334
tbdlinux	2	**15764**	24463	**15764**	24463	**15764**	**15764**
	4	42652	54262	39715	40636	30604	**30444**
	8	90919	96038	68159	69815	58511	**49120**
	16	177347	155604	98171	100786	87998	**75884**
	32	297658	227368	143953	140455	118736	**106563**
	64	486874	325000	188646	187876	164759	**148263**

TABLE 5.4

Communication volume of p-way partitioning for rectangular test matrices. The lowest volume is marked in boldface.

It can be seen that the best-direction strategy automatically recognises when to use a one-dimensional partitioning, and also which one to use, and when to stop using it (e.g. at $p = 64$ for `well1850` and `dfl001`). This table also explains the similar results in the previous table of the best-direction strategy with the one-dimensional row strategy for `well1850`, and with the one-dimensional column strategy for `dfl001`, `gemat1`, `cre_b`, and `nug30`. For these matrices, the communication is well-balanced, and the maximum number sent and received are both within a factor of two of the optimal value for all problem instances, except `well1850`/64 and `gemat1`/32. For the term-by-document matrices, the balance deteriorates slightly because two smaller volumes are balanced instead of one larger volume. Still, the balance is within a factor of three from optimal. For these two matrices, the gain in communication volume by the two-dimensional strategy is somewhat reduced by the deteriorating communication balance, but the overall gain is still significant. (For example, `tbdlinux`/64 with the one-dimensional column strategy has an average of 5079 and maxima of 8314 and 9529 sends and receives, respectively. Its highest value, 9529, is still larger than the sum of the highest values in phases 1 and 3, i.e. 3746+2940=6686, for the two-dimensional strategy.)

Table 5.6 presents the total communication volume for the partitioning of the square test matrices by using the Mondriaan program with eight different strategies, namely the six direction-choosing strategies shown in Table 5.4, each followed by unconstrained vector partitioning, and the best-direction strategy (both with and without dummy addition), followed by vector partitioning with the constraint $\mathrm{distr}(\mathbf{u}) = \mathrm{distr}(\mathbf{v})$. Again, the best-direction strategy is the best of the six direction-choosing strategies, winning in 21 out of 36 instances, but without dramatic gains. The constraint on the vector distribution gives rise to much more communication, except for the matrix `memplus`, which has only nonzeros on the diagonal, see Table 5.2, so that no extra communication occurs. Adding dummies during the matrix partitioning is beneficial, and is better than taking the constraint into account only during the following vector partitioning. The exception is for small matrices (`impcol_b`, `west0381`, and `gemat11`) with large p, where it is better to add a limited number of communications (at most n) at the vector partitioning stage after having partitioned the original matrix, unperturbed by adding dummies.

Table 5.7 shows the communication balance for the square test matrices partitioned using the best-direction strategy with addition of dummies, followed by vector partitioning with the constraint $\mathrm{distr}(\mathbf{u}) = \mathrm{distr}(\mathbf{v})$. In most cases (21 out of 36 instances) the communication balance is within a factor of two from optimal. For the matrix `memplus`, however, the balance is far from optimal, and the same holds for phase 3 for `lhr34`. Note that for all square matrices, communication is usually performed in both phases, and for three matrices this even happens for $p = 2$. (After a one-dimensional matrix partitioning, such as happens for $p = 2$, the vector partitioning can still decide to add communications in the other direction, if it finds this advantageous for reasons of communication balance. This must be disallowed if a final one-dimensional partitioning is desired.)

name	p	phase 1			phase 3		
		avg	max send	max rec	avg	max send	max rec
well1850	2	18	18	18	0	0	0
	4	25	27	26	0	0	0
	8	22	26	27	0	0	0
	16	20	27	29	0	0	0
	32	22	33	75	0	0	0
	64	30	57	68	5	19	19
dfl001	2	0	0	0	431	431	431
	4	0	0	0	354	378	379
	8	0	0	0	301	329	324
	16	0	0	0	228	243	258
	32	0	0	0	155	172	236
	64	3	29	29	95	111	122
gemat1	2	0	0	0	85	85	85
	4	0	0	0	65	82	81
	8	0	0	0	63	71	71
	16	0	0	0	43	69	68
	32	0	0	0	41	88	71
	64	0	0	0	35	62	65
cre_b	2	0	0	0	382	382	382
	4	0	0	0	508	533	533
	8	0	0	0	424	448	445
	16	0	0	0	289	392	463
	32	0	0	0	212	383	293
	64	0	0	0	148	287	259
tbdmatlab	2	2528	2528	2528	0	0	0
	4	2291	3968	4106	461	922	922
	8	1129	2126	2240	1096	1376	1376
	16	730	1401	1788	1005	1187	1602
	32	572	977	1226	695	1253	1355
	64	352	627	1028	455	830	796
nug30	2	0	0	0	13260	13260	13260
	4	0	0	0	14362	14578	14577
	8	0	0	0	11531	14455	13242
	16	0	0	0	7667	9761	11748
	32	0	0	0	5309	9841	7028
	64	0	0	0	3428	6725	4541
tbdlinux	2	7882	7882	7882	0	0	0
	4	6741	9574	11115	871	1742	1742
	8	4356	7601	11139	1784	2907	2907
	16	2463	4531	8292	2281	4102	4350
	32	1978	3842	6577	1353	3467	3198
	64	1166	2230	3746	1152	2940	2168

TABLE 5.5

Communication balance of p-way partitioning for rectangular test matrices for the two-dimensional best-direction strategy

name	p	1D row	1D col	2D alt row	2D alt col	2D ratio	2D best dir	2D eq best dir dum	2D eq best dir
impcol_b	2	**28**	41	**28**	41	43	**28**	28	42
	4	76	107	78	77	**75**	76	84	82
	8	132	155	127	125	124	**121**	137	145
	16		199	190	**171**	179		196	
	32	**200**	249	236	214	237	205	249	242
	64		**251**	298	281	292		297	
west0381	2	**50**	55	**50**	55	55	**50**	183	218
	4	**182**	265	194	200	203	185	389	438
	8	467	535	486	517	468	**464**	590	727
	16	840	816	762	773	747	**732**	900	1026
	32	1136	1326	1178	1193	1219	**1090**	1411	1418
	64			1687	1610	1615	**1443**	1818	1786
gemat11	2	92	**58**	92	**58**	**58**	**58**	1235	2550
	4	211	**132**	297	157	133	138	2290	3551
	8	341	348	433	302	302	**301**	3528	4518
	16	529	626	669	576	595	**502**	4687	5069
	32	**832**	1058	1040	906	930	855	5600	5538
	64	**1677**	2584	2089	1751	1906	1746	6836	6531
memplus	2	2778	**2543**	2778	**2543**	**2543**	**2543**	2543	2543
	4	4816	4641	5220	5132	5045	**4614**	4614	4614
	8	6673	6639	7108	7314	6798	**6222**	6222	6222
	16	8319	8134	8922	8882	8755	**7998**	7998	7998
	32	10121	9692	10464	10360	10129	**9486**	9486	9486
	64	11695	11524	12236	12339	12098	**11227**	11227	11227
onetone2	2	**478**	1044	**478**	1044	1125	**478**	836	2926
	4	1827	2024	**1620**	1854	1792	1632	2660	9071
	8	2537	3196	**2521**	2942	2757	2528	4501	15055
	16	3344	4307	3717	3767	3674	**3233**	5726	14957
	32	**4538**	5638	4904	5463	4683	4639	7679	21849
	64	**6761**	9091	7253	6889	7396	6769	10856	25372
lhr34	2	**284**	1370	**284**	1370	1190	**284**	64	9722
	4	**1263**	2251	1606	1881	2072	1483	6391	27615
	8	2581	3680	2566	4043	3443	**2299**	14256	31466
	16	**3949**	6268	5921	5998	6376	4730	22031	34669
	32	**6413**	12548	12595	10512	9702	7081	29871	39965
	64	9989	15656	15282	15152	14484	**9577**	36999	42542

TABLE 5.6

Communication volume of p-way partitioning for square test matrices. The lowest volume for the first six strategies is marked in boldface.

name	p	phase 1			phase 3		
		avg	max send	max rec	avg	max send	max rec
impcol_b	2	14	14	14	0	0	0
	4	15	20	21	7	13	14
	8	9	14	13	8	9	10
	16	8	11	11	5	9	10
	32	5	9	8	4	7	9
	64	3	11	5	2	5	5
west0381	2	19	29	29	73	97	97
	4	51	57	62	47	57	58
	8	45	59	66	30	43	46
	16	31	43	45	26	40	38
	32	26	40	44	19	34	29
	64	17	28	27	12	26	25
gemat11	2	392	435	435	226	257	257
	4	291	424	440	283	455	434
	8	270	433	481	171	311	301
	16	120	210	235	174	288	263
	32	103	178	180	73	118	128
	64	55	94	102	53	102	84
memplus	2	0	0	0	1272	1843	1843
	4	399	802	802	755	1288	1609
	8	445	817	683	334	930	828
	16	266	714	805	235	600	425
	32	95	408	460	202	458	390
	64	96	293	386	80	241	179
onetone2	2	2	4	4	416	416	416
	4	382	492	597	284	458	424
	8	266	388	450	297	430	375
	16	220	300	400	139	289	263
	32	135	202	291	106	261	204
	64	99	189	230	72	205	138
lhr34	2	0	0	0	32	32	32
	4	975	1742	1742	624	1248	1277
	8	864	1310	1808	919	2135	1755
	16	859	1208	1385	519	1085	922
	32	579	911	1036	355	797	639
	64	382	603	682	197	559	381

TABLE 5.7

Communication balance of p-way partitioning for square test matrices for the two-dimensional best-direction strategy with $\mathrm{distr}(\mathbf{u}) = \mathrm{distr}(\mathbf{v})$ *and with use of dummies.*

Table 5.8 presents the total communication volume for the partitioning of the symmetric test matrices by using the Mondriaan program with eight different strategies, namely the six direction-choosing strategies shown in Table 5.4, each followed by unconstrained vector partitioning, and the best-direction strategy with a single nonzero representing both a_{ij} and a_{ji} in the input matrix, either chosen as the entry with $i \geq j$, or by flipping a coin. No dummies are added, because all diagonal elements are already nonzero. This implies that the communication volume does not increase during the vector partitioning if we apply the constraint $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$. The results of the table show that similar to the rectangular and square case the best-direction strategy performs best. We observe that exploitation of symmetry is advantageous for two matrices, `cage10` and `finan512`, which are both stochastic matrices. This advantage occurs if we represent entries a_{ij} with $i \geq j$ in our input matrix. The gain can be up to a factor of two, for `finan512`/8. Making a random choice $i \geq j$ or $i \leq j$ turns out to be bad; we attribute this to the fact that possible similarity between rows or columns is destroyed, severely hampering matching. On the other hand, representing only the lower triangular matrix part leaves much of the similarity intact, both in the rows and columns. For the other three matrices, which represent computational grids, symmetry cannot explicitly be used to our advantage.

Table 5.9 shows the communication balance for the symmetric test matrices partitioned using the best-direction strategy, followed by vector partitioning with the constraint $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$. In only 14 out of 30 instances the communication balance is within a factor of two from optimal. We explain this worsening balance as follows: for square matrices with a completely nonzero diagonal, such as `memplus` and all the symmetric matrices, the constraint $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$ forces assigning v_j and u_j to the same processor as the diagonal element a_{jj}, see Section 4. This has the advantage of avoiding an increase in communication volume by the constraint, but it leaves no choice during the vector partitioning. Thus the vector partitioning is determined by the matrix partitioning, through the matrix-diagonal assumption, as in previous methods [6, 7, 10, 11, 24, 28]. For square matrices with zeros on the diagonal, this is not the case. Here, dummies are added before the matrix partitioning, but they are removed before the vector partitioning. (Keeping them would impose unnecessary constraints on the vector distribution.) If the intersection between the owners of row and column j is empty, which happens sometimes, we can choose between all the owners in the union of the two sets, trying to optimise the communication balance.

Finally, to check the quality of our implementation, and in particular that of the splitting function h, we compare some results to previously published results. The matrix `dfl001` was used in [23], for $p = 8$. The best result, for a one-dimensional column partitioning with the ML-FM method is a volume of 5875; for the same strategy, our result is 2561. The maximum volume per processor (sends or receives) is 1022; our result is 725. Note however, that for a fair comparison, time should also be taken into account: our computation took about 4 seconds on a 500 Mhz Sun Blade workstation, whereas the partitioning in [23] took 2 seconds on a PC with a 300 Mhz Pentium II processor. (Spending more time can help improve solution quality.) The results for `memplus` are: total volume 6333 for ML-FM in [23], and 6673 for our one-dimensional row method; the maximum volume is 1339 and 2667, respectively. The matrix `hyp_200_2_1` was used in [7], where the communication cost $b = 0.016$ translates into a total communication volume of about 5800 for $p = 100$. This cost is for a partitioning of the square computational grid into 'digital circles', which is better than square blocks. Our present results are a volume of 4877 for $p = 64$ and

name	p	1D row	1D col	2D alt row	2D alt col	2D ratio	2D best dir	2D best dir lower	2D best dir random
cage10	2	2348	2278	2348	2278	2337	2348	**2022**	3150
	4	5551	5512	5401	4987	5482	5284	**4228**	7174
	8	8963	8801	8117	8197	8132	8256	**6932**	10666
	16	12769	12766	11555	11401	11317	11272	**9458**	15366
	32	17720	17369	15039	15138	15209	14757	**13310**	19888
	64	23181	23405	19855	19971	19238	19232	**17650**	25422
hyp_200_2_1	2	800	800	800	800	800	800	**799**	846
	4	1445	1550	**1444**	1547	1547	1445	1596	2114
	8	2282	2184	2146	2142	2244	**1933**	2418	3962
	16	2745	2940	**2692**	2851	2845	2763	3226	5024
	32	3869	**3752**	3767	3851	3939	3854	4704	6294
	64	4877	5346	**4672**	5602	4787	5187	6525	7500
finan512	2	146	292	146	292	292	146	**100**	720
	4	876	949	1168	1168	1022	657	**392**	1906
	8	1314	1533	1387	1387	1460	1095	**550**	2250
	16	2190	1972	1898	3017	2190	1825	**1234**	2256
	32	2337	2336	3411	2706	2967	2336	**1738**	7522
	64	9286	9667	9889	10957	10210	**9282**	10198	14350
bcsstk32	2	1622	1622	1622	1622	**1259**	**1259**	1770	1992
	4	2951	2951	2506	2506	2710	**2490**	2958	4998
	8	5593	5593	6058	5753	6499	**5057**	7534	10188
	16	9303	9156	8931	8931	8940	**8320**	9702	15520
	32	13742	13742	14827	14897	13594	**12736**	16702	24884
	64	19765	20196	21165	21353	21563	**19121**	21826	37794
bcsstk30	2	948	1158	948	1158	1158	948	**620**	1334
	4	2099	**1757**	3602	2307	2367	2083	2924	3572
	8	5019	4492	4642	4613	4428	4869	**4102**	8382
	16	9344	9949	9945	8808	9261	**8707**	10572	15648
	32	15593	15418	15127	16607	16363	**14169**	17232	27868
	64	27269	26599	25492	26510	26227	**23545**	26388	41756

TABLE 5.8

Communication volume of p-way partitioning for symmetric test matrices. The lowest volume is marked in boldface.

7519 for $p = 128$, which by interpolation indicates that our hypergraph-based partitioning recognises the step-like boundaries between domains that are characteristic for digital circles. (For square blocks, the theoretical volume would be 6400 for $p = 64$ and 8000 for $p = 100$.) The square matrices gemat11, onetone2, lhr34, finan512, bcsstk32, and bcsstk30 were used in [10]. We can compare for instance gemat11 with [10, Table 3]. For our one-dimensional row partitioning with dummies and with $\mathrm{distr}(\mathbf{u}) = \mathrm{distr}(\mathbf{v})$, we obtain for $p = 8, 16, 32, 64$ scaled volumes of 0.72, 0.96, 1.14, and 1.42, respectively, which is close to the values 0.75, 0.96, 1.15, and 1.32, of PaToH-HCM, and 0.79, 1.00, 1.18, and 1.33 of hMetis. (We obtain the scaled communication volume by dividing the total volume by n.) We may conclude that our bipartitioning

name	p	phase 1			phase 3		
		avg	max send	max rec	avg	max send	max rec
cage10	2	1174	1222	1222	0	0	0
	4	710	885	885	612	783	835
	8	413	684	739	620	967	905
	16	381	533	682	324	650	564
	32	257	469	493	205	370	346
	64	167	293	311	135	318	262
hyp_200_2_1	2	400	400	400	0	0	0
	4	362	370	371	0	0	0
	8	86	173	174	157	208	208
	16	104	155	154	70	120	123
	32	39	96	98	82	139	138
	64	37	74	73	45	94	92
finan512	2	73	73	73	0	0	0
	4	165	219	219	0	0	0
	8	28	111	108	110	213	225
	16	115	152	155	0	0	0
	32	60	82	81	14	77	69
	64	82	168	170	64	162	155
bcsstk32	2	0	0	0	630	669	669
	4	445	733	795	178	229	229
	8	441	911	888	192	372	335
	16	408	812	925	113	322	328
	32	173	359	457	226	501	466
	64	177	331	347	122	297	292
bcsstk30	2	474	494	494	0	0	0
	4	181	340	340	341	459	458
	8	342	652	675	268	634	659
	16	190	526	495	355	721	797
	32	211	574	516	232	667	593
	64	194	495	566	174	357	394

TABLE 5.9

Communication balance of p-way partitioning for symmetric test matrices for the two-dimensional best-direction strategy with $\mathrm{distr}(\mathbf{u}) = \mathrm{distr}(\mathbf{v})$.

implementation is similar in quality to that of the other hypergraph-based partitioners, and that this is a good basis for our two-dimensional approach.

6. Conclusions and future work. In this work, we have presented a new two-dimensional method for distributing the data for sparse matrix-vector multiplication. The method has the desirable characteristics stated in Section 1: it tries to spread the matrix nonzeros evenly over the processors; it tries to minimise the true communication volume; it tries to spread the communication operations evenly; and it is two-dimensional. The experimental results of our implementation, Mondriaan, show that for many matrices this indeed leads to lower communication cost than for a comparable one-dimensional implementation such as Mondriaan in one-dimensional mode. For term-by-document matrices, we observe a large gain. The gain is visible in

two metrics: total communication volume and maximum volume per processor. We make no attempt to reduce the total number of messages: our upper bound is $2(p-1)$ messages per processor. We consider this number less important because it does not grow with the problem size. The best variant of our algorithm uses the strategy of trying both splitting directions, each time choosing the best. This has the advantage that the strategy adapts itself automatically to the matrix, without requiring any prior knowledge thereof.

Our data distribution method is a two-stage process: first, the matrix is distributed, which determines the communication volume and the computational load balance; after that the input and output vectors are distributed, which determines the communication load balance. The separation between the stages makes it easier to optimise our data distribution.

To achieve our goals, we had to generalise the Cartesian matrix distribution scheme to a matrix partitioning into rectangular, possibly scattered submatrices, which we call, in a lighter vein, the *Mondriaan distribution*. This scheme is not as simple as the Cartesian scheme, which includes most commonly used partitioning methods. In the Cartesian scheme, we can view a matrix distribution as the result of permuting the original matrix A into a matrix PAQ, splitting its rows into consecutive blocks, splitting its columns into consecutive blocks, and assigning each resulting submatrix to a processor. This view does not apply anymore. Still, the matrix part of a processor is defined by a set of rows I and columns J, and its set of index pairs is a Cartesian product $I \times J$. We can fit all the submatrices in a nice figure that bears some resemblance to a Mondriaan painting.

Much future work remains to be done. First, we have made several design decisions concerning the heuristics in our algorithm. Further investigation of all the possibilities may yield even better heuristics. Second, we have presented the general distribution method, but have not investigated special situations such as square symmetric matrices in depth. Further theoretical and experimental work in this area is important for many iterative solvers. Third, parallel implementations of iterative solvers such as those in the Templates projects [1, 2] should be developed that can handle every possible matrix and vector distribution. We have started to develop an object-oriented iterative linear system solver package for this purpose. Fourth, the partitioning itself should be done in parallel to enable solving very large problems that do not fit in the memory space of one processor. Preferably, the result of the parallel partitioning method should be of the same quality as that of the corresponding sequential method. Since quality may be more important than speed, a distributed algorithm that more or less simulates the sequential partitioning algorithm could be the best approach. The recursive nature of the partitioning process may be helpful as this already has some natural parallelism.

REFERENCES

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

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

[3] M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Transactions on Computers, C-36 (1987), pp. 570–580.

[4] M. Berry, Z. Drmač, and E. R. Jessup, *Matrices, vector spaces, and information retrieval*, SIAM Review, 41 (1999), pp. 335–362.
[5] M. L. Bilderback, *Edge-cut imbalances produced by graph partitioning algorithms*, in Proceedings High Performance Computing Symposium HPC99, San Diego, Apr. 1999.
[6] R. H. Bisseling, *Parallel iterative solution of sparse linear systems on a transputer network*, in Parallel Computation, A. E. Fincham and B. Ford, eds., vol. 46 of The Institute of Mathematics and its Applications Conference Series, Oxford University Press, Oxford, UK, 1993, pp. 253–271.
[7] R. H. Bisseling and W. F. McColl, *Scientific computing on bulk synchronous parallel architectures*, in Technology and Foundations: Information Processing '94, Vol. I, B. Pehrson and I. Simon, eds., vol. 51 of IFIP Transactions A, Elsevier Science Publishers, Amsterdam, 1994, pp. 509–514.
[8] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, *Matrix Market : a web resource for test matrix collections*, in The Quality of Numerical Software: Assessment and Enhancement, R. F. Boisvert, ed., Chapman and Hall, London, 1997, pp. 125–137.
[9] T. Bui and C. Jones, *A heuristic for reducing fill in sparse matrix factorization*, in Proceedings Sixth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, PA, 1993, pp. 445–452.
[10] Ü. V. Çatalyürek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel and Distributed Systems, 10 (1999), pp. 673–693.
[11] ———, *A hypergraph-partitioning approach for coarse-grain decomposition*, in Proceedings Supercomputing 2001, ACM, 2001.
[12] T. A. Davis, *University of Florida sparse matrix collection*, online collection, http://www.cise.ufl.edu/research/sparse/matrices, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 1994-2001.
[13] I. S. Duff, R. G. Grimes, and J. G. Lewis, *Sparse matrix test problems*, ACM Transactions on Mathematical Software, 15 (1989), pp. 1–14.
[14] ———, *The Rutherford-Boeing sparse matrix collection*, Technical Report TR/PA/97/36, CERFACS, Toulouse, France, Sept. 1997.
[15] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, in Proceedings of the 19th IEEE Design Automation Conference, IEEE, 1982, pp. 175–181.
[16] R. Fletcher, *Conjugate gradient methods for indefinite systems*, in Proceedings of the Dundee Biennial Conference on Numerical Analysis, G. A. Watson, ed., vol. 506 of Lecture Notes in Mathematics, Springer-Verlag, Berlin, 1976, pp. 73–89.
[17] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors: Vol. I, General Techniques and Regular Problems*, Prentice Hall, Englewood Cliffs, NJ, 1988.
[18] R. W. Freund and N. M. Nachtigal, *QMR: a quasi-minimal residual method for non-Hermitian linear systems*, Numerical Mathematics, 60 (1991), pp. 315–339.
[19] D. M. Gay, *Electronic mail distribution of linear programming test problems*, Mathematical Programming Society COAL Newsletter, 13 (1985), pp. 10–12.
[20] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins Studies in the Mathematical Sciences, The Johns Hopkins University Press, Baltimore, MD, third ed., 1996.
[21] B. Hendrickson, *Graph partitioning and parallel solvers: Has the emperor no clothes?*, in Proceedings Irregular'98, A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng, eds., vol. 1457 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1998, pp. 218–225.
[22] B. Hendrickson and T. G. Kolda, *Graph partitioning models for parallel computing*, Parallel Computing, 26 (2000), pp. 1519–1534.
[23] ———, *Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing*, SIAM Journal on Scientific Computing, 21 (2000), pp. 2048–2072.
[24] B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, in Proceedings Supercomputing '95, ACM Press/IEEE Press, 1995.
[25] B. A. Hendrickson, R. Leland, and S. Plimpton, *An efficient parallel algorithm for matrix-vector multiplication*, International Journal of High Speed Computing, 7 (1995), pp. 73–88.
[26] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards, 49 (1952), pp. 409–436.
[27] Y. F. Hu, K. C. F. Maguire, and R. J. Blake, *A multilevel unsymmetric matrix ordering algorithm for parallel process simulation*, Computers and Chemical Engineering, 23 (2000),

pp. 1631–1647.

[28] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing, 20 (1998), pp. 359–392.

[29] ———, *Parallel multilevel k-way partitioning scheme for irregular graphs*, SIAM Review, 41 (1999), pp. 278–300.

[30] B. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, 29 (1970), pp. 291–307.

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

[32] J. G. Lewis and R. A. van de Geijn, *Distributed memory matrix-vector multiplication and conjugate gradient algorithms*, in Proceedings Supercomputing 1993, ACM Press, 1993, pp. 484–492.

[33] A. T. Ogielski and W. Aiello, *Sparse matrix computations on parallel processor arrays*, SIAM Journal on Scientific Computing, 14 (1993), pp. 519–530.

[34] A. Pinar and C. Aykanat, *Sparse matrix decomposition with optimal load balancing*, in Proceedings International Conference on High Performance Computing (HiPC'97), 1997, pp. 224–229.

[35] L. F. Romero and E. L. Zapata, *Data distributions for sparse matrix vector multiplication*, Parallel Computing, 21 (1995), pp. 583–605.

[36] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 856–869.

[37] H. A. van der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 13 (1992), pp. 631–644.

[38] A. van Heukelum, G. T. Barkema, and R. H. Bisseling, *DNA electrophoresis studied with the cage model*, Journal of Computational Physics, (2002, to appear).

[39] C. Walshaw and M. Cross, *Multilevel mesh partitioning for heterogeneous communication networks*, Future Generation Computer Systems, 17 (2001), pp. 601–623.