SIAM REVIEW
Vol. 47, No. 1, pp. 67–95

A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication*

Brendan Vastenhouw†
Rob H. Bisseling‡

Abstract. A new method is presented for distributing data in sparse matrix-vector multiplication. The method is two-dimensional, tries to minimize 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 minimized. After the matrix partitioning, the input and output vectors are partitioned with the objective of minimizing the maximum communication volume per processor. Experimental results of our implementation, Mondriaan, for a set of sparse test matrices show a reduction in communication volume compared to one-dimensional methods, and in general a good balance in the communication work. Experimental timings of an actual parallel sparse matrix-vector multiplication on an SGI Origin 3800 computer show that a sufficiently large reduction in communication volume leads to savings in execution time.

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

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

DOI. 10.1137/S0036144502409019

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, 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 communication overhead is minimal.

The natural parallel algorithm for sparse matrix-vector multiplication with an arbitrary distribution of the matrix and the 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.

*Received by the editors May 29, 2002; accepted for publication (in revised form) April 18, 2004; published electronically February 1, 2005.
http://www.siam.org/journals/sirev/47-1/40901.html

†Image Sciences Institute, University Medical Center Utrecht, P.O. Box 85500, 3508 GA Utrecht, The Netherlands (brendan@isi.uu.nl).

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

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 $u_i = \sum_{t=0}^{p-1} u_{it}$.

To facilitate theoretical time analysis, processors are assumed to synchronize globally between the phases. In an actual implementation, this requirement may be relaxed.

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 minimize the communication cost in the algorithm above. A good distribution scheme has the following characteristics:

(i) It tries to spread the matrix nonzeros evenly over the processors, to minimize the maximum amount of work of a processor in phase 2.

(ii) It tries to minimize the true *communication volume*, i.e., the total number of data words communicated, 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.)

(iii) It tries to spread the communication evenly over the processors, with respect to both sending and receiving, to minimize the maximum number of data words sent and received by a processor in phases 1 and 3.

(iv) It tries to partition the matrix in both dimensions, e.g., by splitting it into rectangular blocks. Under certain conditions, two-dimensional (2D) partitioning limits the number of destination processors 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, it also limits the communication in phase 3. Although a one-dimensional (1D) row distribution has the advantage that it removes phases 3 and 4, the price to be paid is high: the elements of a column must be distributed over a larger number of processors, and the number of destination processors of v_j can reach $p-1$. Of course, for certain well-structured matrices this number may be far less. For instance, for sparse matrices from the finite-element field, it is only a small constant. Such matrices are expected to gain little from a 2D approach.

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 minimize the number of cut edges. An edge (i, j) is *cut* if vertices i and j are assigned to different processors. This 1D method is the basis of the partitioning algorithms implemented in software such as Chaco [25] and Metis [31], which have 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 [32], has recently been developed.

Hendrickson [22] criticizes the graph partitioning approach because it can handle only square symmetric matrices, imposes the same partitioning for the input and

output vectors, and does not necessarily try to minimize the communication volume, nor the number of messages, nor the maximum communication load of a processor. Hendrickson and Kolda [23] 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 often we have been fortunate that the effect of these disadvantages has been limited. This is because many applications originate in differential equations discretized on a grid, where the number of neighbors 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 the potential for improvement of the communication load balance. Hendrickson and Kolda [24] present an alternative, the bipartite graph model, which identifies the rows of an $m \times n$ matrix with a set of m row vertices, the columns with a set of n column vertices, and the nonzero elements a_{ij} with edges (i, j) between a row vertex i and a 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 1D row distribution that conforms to the partitioning of the row vertices. The vertices are partitioned by a multilevel algorithm that tries to minimize the number of cut edges while keeping the difference in work between processors less than the work of a single matrix row or column. A disadvantage of this approach is that only an approximation to the communication volume is minimized, and not the true volume. The bipartite 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*. A hyperedge is a subset of $\mathcal{V}$. In their row-net model, each row of an $m \times n$ matrix corresponds to a hyperedge and each column to a vertex; each vertex has a weight equal to the number of nonzeros in the corresponding column. (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: 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 total vertex weight is balanced among the sets 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. (Phase 1 vanishes.) The advantage of this approach is that it minimizes the true communication volume and not an approximation to the volume. This indeed leads to less communication: experimental results for the PaToH (partitioning tool for hypergraphs) program show a 35% reduction in volume for a set of test matrices from the Rutherford–Boeing collection [14, 15] and some linear programming (LP) 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. The hypergraph-based approach can also be applied to rectangular matrices, as shown by Pinar et al. [40] for the partitioning of rectangular LP matrices. (Taking a different approach, Pinar and Aykanat [38] transform rectangular LP matrices to an undirected graph representing

the interaction between the rows, thus enabling the use of graph partitioners. This minimizes an approximation to the communication volume.)

Hu, Maguire, and Blake [30] 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 factorization. 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 1D 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 optimized 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 2D partitioning as a means of reducing communication. Fox et al. [18, Chap. 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 [36] and Hendrickson, Leland, and Plimpton [26] 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 2D algorithms with a 1D algorithm and find gains of a factor of 2.5 on an Intel iPSC/860 hypercube. Ogielski and Aiello [37] partition the rows and columns of a matrix A by first permuting them randomly, giving a matrix PAQ, and then distributing the rows and columns of PAQ by blocks. This gives a 2D partitioning with an expected good load balance. Pinar and Aykanat [39] 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 2D rowwise jagged partitioning.

Bisseling [6] presents a 2D 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; communication is done within chains of processors of minimal length. For instance, v_j is broadcast to a set of processors (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 communication is 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 analyze the communication of various distributions, taking the maximum of the number of data words sent by a processor and the number received as the local cost, and taking the maximum over all the processors of this local cost as the cost function for the communication. 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 extend their previous 1D hypergraph-based partitioning method to two dimensions. In a coarse-grain approach [12], 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 while trying to spread the nonzeros in all the row sets simultaneously by solving a multiconstraint 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 1D 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 1D distribution. In [11], Çatalyürek and Aykanat take a different, fine-grain approach which formulates the matrix partitioning problem as a hypergraph partitioning problem by identifying each nonzero with a vertex, each row with a hyperedge, and each column with a hyperedge (thus reversing the roles of vertices and hyperedges compared to previous hypergraph-based methods). Since individual nonzeros are assigned to processors, the resulting 2D partitioning is in principle the most general possible. Experiments for a set of square test matrices show an average savings of 43% in communication volume compared to the volume of a 1D hypergraph-based implementation.

Berger and Bokhari [3] present a recursive bisection-based strategy for partitioning nonuniform 2D 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 optimization technique, which has been used, for instance, in parallel circuit simulation; see Fox et al. [18, Chap. 22]. It can also be used for partitioning matrices, as has been done by Romero and Zapata [41] to achieve good load balance in sparse-matrix vector multiplication.

In the present work, we bring several 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 [45] 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 bidiagonalization [21], 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 [46]), but adopting such an approach would make our algorithm more complicated.

The remainder of this paper is organized as follows. Section 2 presents a 2D method for partitioning the sparse matrix that attempts to minimize the communication volume. Section 3 presents a method for partitioning the input and output vectors that attempts to balance the communication volume among 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. 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 the elements a_{ij}, with $0 \leq i < m$ and $0 \leq j < n$, are either 0 or 1. Without loss of generality, we assume that each row and column has at least one nonzero. (Empty rows and columns can easily be removed from the problem.) 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 processors, $p \geq 1$, 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 words 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. Such an assignment is better than assignment to a nonowner, which causes 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*

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

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

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

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

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

Note that 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 of A, and if we assign each subset to a processor, then $V(A_0, \ldots, A_{p-1})$ is exactly the communication volume in the natural parallel algorithm. This is because every v_j is sent from its owner to all the other μ'_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 λ'_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 λ'_i, for $0 \leq i < m$, and by μ'_j, for $0 \leq j < n$, from which the result follows by summing. We will only treat the case of the λ'_i; the case of the μ'_j is similar. Let i be a row index. We have to prove that

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

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

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

which is the left-hand side. If A_{k-2} and A_{k-1} do not have a nonzero in row i, the left-hand side and the right-hand side of (2.6) both equal $\lambda'_i(A_0, \ldots, A_{k-3})$. ∎

This theorem is a generalization 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 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. Minimizing V thus minimizes an upper bound on the cost of phase 4. Balancing the communication load in phase 3 automatically 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.

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 $\mu'_j = 0$ 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 $\lambda'_i = 1$. The problem of assigning columns to two processors is exactly the two-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 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 1. To calculate the work load, every vertex j is weighted by the number of nonzeros c_j of column j, giving the criterion

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

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 [35]. To capture these methods, we define a hypergraph splitting function h on a matrix subset A by

$$(A_0, A_1) \leftarrow h(A, \mathit{sign}, \epsilon). \tag{2.10}$$

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 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. To keep our overall algorithm simple, we do not follow this approach.

Next, we consider the case $p = 4$. Aiming at a 2D partitioning we could first partition the columns into sets J_0 and J_1, and then the rows into sets I_0 and I_1. This would 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

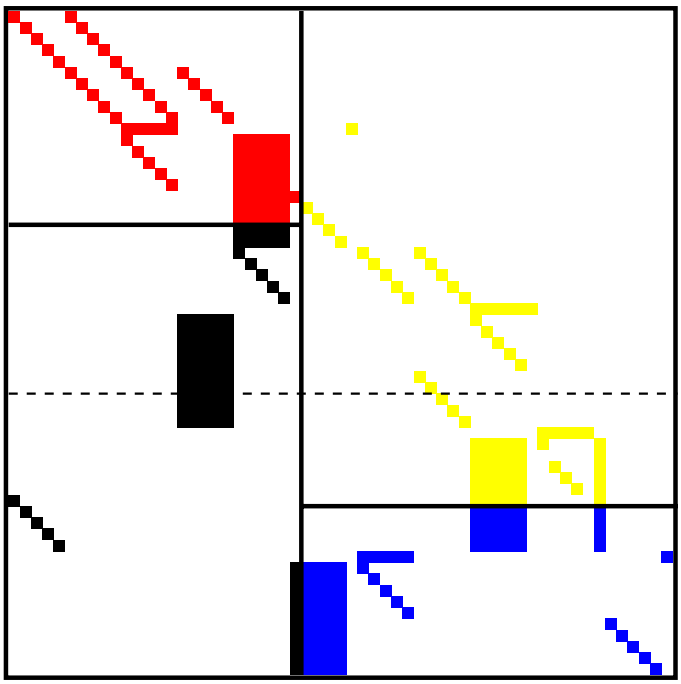

Fig. 2.1 *Block distribution of the* 59×59 *matrix* `impcol_b` *with* 312 *nonzeros from the Rutherford–Boeing collection* [14, 15] *over four processors, depicted by the colors red, yellow, blue, and black. The matrix is first partitioned into two blocks of columns, and then each block is partitioned 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.*

of J_1, and vice versa. This will often lead to a compromise partitioning of the rows. Dropping the Cartesian constraint enlarges the set of possible partitionings and hence gives better solutions. Therefore, we partition both parts separately. Theorem 2.2 implies that this can even be done independently, because

$$V(A_0, A_1, A_2, A_3) = V(A_0 \cup A_1, A_2 \cup A_3) + V(A_0, A_1) + V(A_2, A_3), \tag{2.11}$$

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 Figure 2.1. For ease of understanding, the matrix shown in the figure has been split by a simple scheme that is solely based on minimizing the computational load imbalance and that partitions the matrix optimally into contiguous blocks. (In general, however, we also try to minimize 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 $\epsilon \approx 2.6\%$, compared to the 128 nonzeros, or $\epsilon \approx 64\%$, for the Cartesian case. (Of course, 1D partitionings can also easily achieve good load balance, but our goal is to partition in both dimensions.) The total communication volume is about the same, 66 vs. 63. It is clear that independent

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

Algorithm 1 *Recursive bipartitioning algorithm with alternating directions.*

partitioning gives much better possibilities to improve the load balance or minimize the communication cost.

The method used to obtain a four-way partitioning from a two-way partitioning into equal-sized parts can be applied repeatedly, resulting in a recursive p-way partitioning algorithm with p a power of 2, given as 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 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 different value $\delta_p \leq \epsilon$ as load imbalance parameter for the current bipartitioning, instead of $\epsilon/\log_2 p$. The partitioning direction need not be chosen alternatingly; it could also be determined by trying both row and column partitionings and then choosing the best direction.

For the alternating-direction strategy, we can guarantee an upper bound on the number of processors μ_j that holds a matrix column j. The bound is $\mu_j \leq \sqrt{p}$, for $0 \leq j < n$, if p is an even power of 2. 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, $\mu_j \leq \sqrt{2p}$ if p is an odd power of 2 and the first bipartitioning has $sign = -1$; otherwise the bound is $\mu_j \leq \sqrt{p/2}$.

A straightforward generalization of Algorithm 1 to the case where p is not necessarily a power of 2 can be obtained by generalizing the splitting function to

$$(A_0, A_1) \leftarrow h(A, sign, \epsilon_0, \epsilon_1, f_0). \tag{2.12}$$

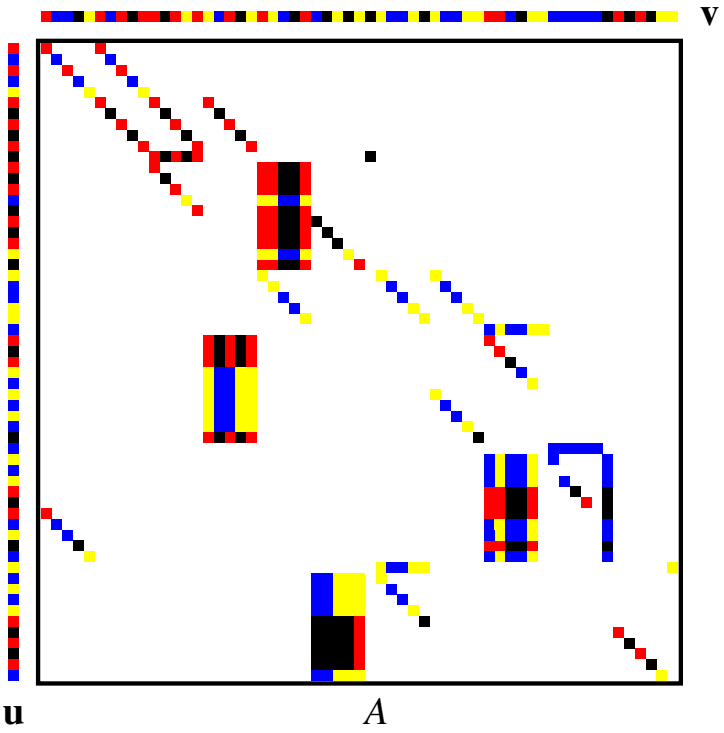

Fig. 2.2 *Global view of the distribution of the matrix* `impcol_b` *over four processors by the recursive bipartitioning algorithm with the best-direction strategy. The processors are depicted by the colors red, yellow, blue, and black; they possess* 79, 78, 79, *and* 76 *nonzeros, respectively. Also given is a distribution of the input and output vectors that assigns each 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.

This function tries to assign a fraction f_0 of the workload to processor 0 and a fraction $1 - f_0$ to processor 1, guaranteeing that

$$W(A_0) \leq (1 + \epsilon_0) f_0 W(A), \quad W(A_1) \leq (1 + \epsilon_1)(1 - f_0) W(A). \tag{2.13}$$

For $\epsilon_0 = \epsilon_1 = \epsilon$ and $f_0 = 1/2$, the original function is retrieved. The splitting function is called in the generalized algorithm with $f_0 = p_0/p$, where $p_0 = \lfloor p/2 \rfloor$; furthermore, let $p_1 = \lceil p/2 \rceil$. This choice makes the fractions as close to $1/2$ as possible and thus minimizes the number of split levels. The imbalance parameters are ϵ/q_0 and ϵ/q_1, where $q_r = \lceil \log_2 p_r \rceil + 1$ for $r = 0, 1$. The generalized algorithm calls itself recursively with p_0 and p_1 processors.

The result of the recursive bipartitioning algorithm is a p-way partitioning of the matrix A. Processor 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 viewpoints.

Figure 2.2 gives the *global view* of the matrix and vector partitioning, showing the original matrix 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 vol-

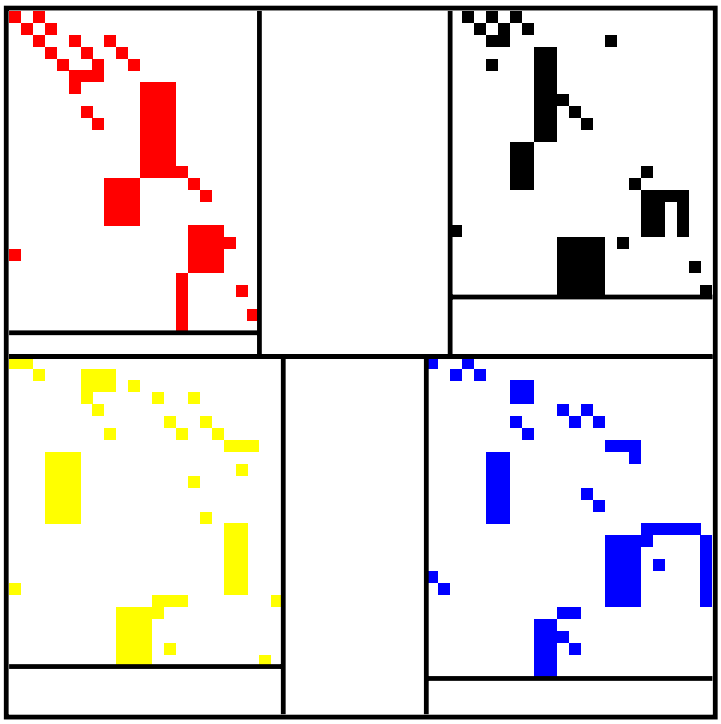

Fig. 2.3 *Local view of the matrix distribution from Figure* 2.2. *The best-direction strategy chooses first to partition in the horizontal direction and then to partition the resulting parts both in the vertical direction. As a result, the red, yellow, blue, and black processors possess submatrices 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.*

ume is 76, which is slightly more than the volume of 66 of the simple block-based distribution method shown by the bold lines in Figure 2.1. (Here, the block-based method is lucky because of the presence of the four large blocks of nonzeros. 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 color than the vector component above it causes communication, and similarly for rows.

Figure 2.3 gives the *local view* of the matrix partitioning, showing the submatrices stored locally at the processors; these submatrices fit in the space of the original matrix. This view displays the structure of the local submatrices. We have removed empty local rows and columns, thus reducing the size of $I_s \times J_s$, to emphasize 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.) Note that nonzeros within the same column of a submatrix $I_s \times J_s$ belong to the same column of the original matrix A, but that there is no such relation between nonzeros from different submatrices, because columns are broken

into parts by the first split and the resulting column parts are permuted to different positions in the top and bottom part of the picture.

3. Vector Partitioning. After the matrix distribution has been chosen with the aim of minimizing 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, the total communication volume is not affected.

We assume that the input and output vectors 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 $A^T\mathbf{u}$, but with the roles of sends and receives reversed, 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 $\mathcal{V}_s$ as the set of indices j corresponding to vector components v_j assigned to processor s. The number of data words sent by processor s in phase 1 equals

$$N_{\mathrm{s}}(s) = \sum_{j \in \mathcal{V}_s} \mu'_j(A_0, \ldots, A_{p-1}) \tag{3.1}$$

and the number of data words received equals

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

A vector partitioning method could attempt to minimize the following:

1. $\max_{0 \leq s < p} N_{\mathrm{s}}(s)$, the maximum number of data words sent by a processor;
2. $\max_{0 \leq s < p} N_{\mathrm{r}}(s)$, the maximum number of data words received by a processor;
3. $\max_{0 \leq s < p} |\mathcal{V}_s|$, 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 because of both outgoing and incoming communication. This justifies trying to minimize both. Second, 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 minimize the number of data words sent but not the number of data words received, this could lead to many data words received in phase 1 of the multiplication by A^T and hence to many data words sent 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 bidiagonalization and in the conjugate gradient method applied to the normal equations [21].) This multiplication can be carried out using the stored matrix A in its present distribution, and the result $A^TA\mathbf{v}$ can be delivered in the distribution of $\mathbf{v}$. The communication pattern of phase 1 for A^T is the same as that of phase 3 for A, except that sending and receiving are interchanged. A data partitioning for multiplication by A that minimizes both the number of data words sent and the number of data words received 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 linear vector operations such as norm or inner product computations, or DAXPYs, in the remaining part of iterative solvers. Often these vector operations are much less time consuming than the matrix-vector multiplication. If desired, we could use load balance in linear vector operations to break ties when our primary objectives are equally met. (Moreover, the maximum number of components could 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 $\mu_j = 1$, v_j has to be assigned to the processor that has all the nonzeros of column j, so that no communication occurs. Now assume that $\mu_j \geq 2$. Assigning v_j to a processor increases the number of data words sent by that processor by $\mu_j - 1$ and the number received by the $\mu_j - 1$ other processors involved by 1. If we take as the cost incurred by a processor the sum of the number of data words sent and received, i.e., $N_{\rm s}(s) + N_{\rm r}(s)$ for processor s, we see that the sum for the sender increases by $\mu_j - 1 \geq 1$ and for the receivers by 1. 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 a cost of at least 1 to all the processors involved and tries to avoid increasing the maximum cost as much as possible.

Our vector partitioning algorithm is presented as Algorithm 2. In step 1 of the algorithm, the sum of each processor is initialized to the number of data words that inevitably must be communicated, one word sent or received per nonempty column part. Initializing the sums before assigning components has the advantage that this amount of unavoidable communication is already taken into account from the first moment that choices must be made. Processors with much unavoidable communication will be assigned fewer components during the algorithm. In step 3, the increment of $\mu_j - 2$ represents the extra communication of the sender. In step 4, where $\mu_j = 2$, the sums are not increased anymore. Now, an attempt is made to balance the number of data words sent with the number received. The choice between sending from processor s to s' or vice versa is made 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 processor s if

$$N_{\rm s}(s) + N_{\rm r}(s') \leq N_{\rm s}(s') + N_{\rm r}(s), \tag{3.3}$$

and to s' otherwise. This gives rise to one data word communicated in the least busy send-receive direction. 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. (The default of our implementation is to handle the columns with $\mu_j \geq 3$ in random order and those with $\mu_j = 2$ in a fixed order.)

We expect Algorithm 2 to minimize $\max_{0 \leq s < p}(N_{\rm s}(s) + N_{\rm r}(s))$ because each of its assignments greedily minimizes this cost function and because it takes all knowledge about inevitable communication cost into account from the start. As a result of successful matrix partitioning, μ_j is often small: in the typical case, $\mu_j = 1$ for the vast majority of columns, avoiding communication altogether; $\mu_j = 2$ for most of the remaining columns; and $\mu_j \geq 3$ for relatively few columns. The range of possible values for μ_j is restricted by $\mu_j \leq p$. In the case of the alternating-direction strategy, $\mu_j \leq \sqrt{p}$, provided p is an even power of 2. Thus, the algorithm adds relatively little cost to the cost that was inevitable from the start. Furthermore, we expect Algorithm 2 to minimize the metric $\max_{0 \leq s < p} \max(N_{\rm s}(s), N_{\rm r}(s))$ as well, because the relatively large number of columns with $\mu_j = 2$ gives many opportunities for balancing sending and receiving.

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

Algorithm 2 *Vector partitioning algorithm.*

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

First, we consider a square nonsymmetric matrix. Iterative algorithms such as GMRES [42], QMR [19], BiCG [17], and Bi-CGSTAB [43] 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 avoid communication during linear vector operations such as norm or inner product computations, or 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 anymore. The vector partitioning algorithm is a straightforward modification of Algorithm 2, where a sum now represents the total sum for phases 1 and 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 each processor 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 $a_{jj} = 1$, 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 $\lambda_j + \mu_j$ processors involved can be chosen as the owner of v_j, but the communication volume increases by 1. This is a consequence of the fact that we cannot simultaneously satisfy the assumptions 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. This situation can occur

only if $a_{jj} = 0$, and hence an upper bound on the volume after constrained vector partitioning is

$$V(\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})) \leq V + |\{j : 0 \leq j < n \wedge a_{jj} = 0\}| \leq V + n. \tag{4.1}$$

We may try to reduce the additional volume 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 [27]. We assume that the diagonal is completely nonzero, which is quite common and holds, e.g., for positive definite matrices. In the matrix partitioning, we may try to exploit the symmetry by requiring the matrix partitioning to assign a_{ij} and a_{ji} to the same processor. The following *symmetric partitioning* method achieves this. First, we create a lower triangular matrix L from A by deleting the nonzeros a_{ij} with $i < j$; then we execute Algorithm 1 or one of its variants on L; and finally we assign each deleted nonzero a_{ij} to the same processor as its partner a_{ji}. The communication volume for the resulting partitioning of A is exactly twice the volume for L. This is because, by a remark above, the intersection of the owners of nonzeros in row j of L and the owners in column j consists of one processor, namely, the owner of l_{jj} (and hence of v_j and u_j). Each value v_j sent by this processor in phase 1 of the multiplication by L must also be sent in phase 1 of the multiplication by A. Correspondingly, a contribution to u_j is received by the same processor in phase 3 of the multiplication by A. A similar remark can be made for the values u_i from phase 3 for L. (This reasoning also holds for 1D partitionings of L, but it may not hold if a more general partitioning algorithm is used that splits L into arbitrary disjoint subsets of the nonzeros, not necessarily submatrices.) As a result, phases 1 and 3 of the multiplication by A have the same communication pattern, although with sending and receiving reversed. The advantage of symmetric partitioning is that it is based on solving a smaller partitioning problem, which may lead to a better and faster solution; the disadvantage is that it restricts the possible solutions.

5. Experimental Results.

5.1. Implementation. We have implemented several variants of Algorithm 1, the recursive bipartitioning of the matrix, and Algorithm 2, the vector partitioning, in a program called Mondriaan.[1] Our implementation assumes that p is a power of 2. The hypergraph bipartitioning function h given in (2.10) has been implemented as

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

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.

First, in the coarsening phase, the matrix is reduced in size by merging columns in pairs. An unmarked column j is picked and its *neighboring 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 neighboring 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. 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 rowwise and columnwise 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 randomly bipartitioned, taking care to balance the amount of work between the two parts, and the bipartitioning is then improved by running the Kernighan–Lin algorithm [33] in the faster Fiduccia–Mattheyses version [16], which we denote by KL–FM. The whole procedure is carried out eight times and the best result is kept. In the KL–FM 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 uncoarsening levels, each time separating the columns that were merged at the corresponding coarsening level, in the first instance assigning them to the same processor as the merged column. After the uncoarsening at a level is finished, KL–FM is run once to refine the partitioning.

5.2. Test Matrices. We have tested version 1.01 of Mondriaan to check the quality of the partitioning produced, using a test set of sparse matrices from publicly available collections, supplemented with a few of our own matrices (which are also 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 ordered 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

Table 5.1 *Properties of the rectangular test matrices.*

Name	Rows	Columns	Nonzeros	Application area
dfl001	6071	12230	35632	Linear programming
cre_b	9648	77137	260785	Linear programming
tbdmatlab	19859	5979	430171	Information retrieval
nug30	52260	379350	1567800	Linear programming
tbdlinux	112757	20167	2157675	Information retrieval

Table 5.2 *Properties of the square test matrices.*

Name	Rows/ columns	Diagonal nonzeros	Nonzeros	Application area
west0381	381	1	2157	Chemical engineering
gemat11	4929	13	33185	Power flow optimization
memplus	17758	17758	99147	Circuit simulation
onetone2	36057	9090	227628	Circuit simulation
lhr34	35152	102	764014	Chemical engineering

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

Name	Rows/ columns	Nonzeros	Application area
cage10	11397	150645	DNA electrophoresis
hyp_200_2_1	40000	200000	Laplacian operation
finan512	74752	596992	Portfolio optimization
bcsstk32	44609	2014701	Structural engineering
bcsstk30	28924	2043492	Structural engineering

make one exception: to facilitate comparison with results in other work [24], we removed 27,003 explicitly stored zeros from the matrix memplus, leaving 99,147 entries that are numerically nonzero. The number of nonzeros is for the complete matrix (below, on, and above the main diagonal), and this also holds in the symmetric case.

The matrices west0381, gemat11, bcsstk32, and bcsstk30 were obtained from the Rutherford–Boeing collection [14, 15]; the matrix memplus from the Matrix Market [8]; and dfl001 and cre_b (part of the Netlib LP collection [20]), nug30, onetone2, lhr34, and finan512 from the University of Florida collection [13]; 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 [45]; it represents the 5979 English-language documents in HTML format of the Matlab 5.3 CD-ROM, containing 48,959 distinct terms, of which 19,859 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 [44] 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. The total communication volume is perhaps the most important metric for expressing communication cost. Reducing the communi-

Table 5.4 *Communication volume of p-way partitioning for rectangular test matrices. The lowest volume is marked in boldface.*

Name	p	1D row	1D col	2D alt row	2D alt col	2D best dir
`df1001`	2	1514	**684**	1513	695	714
	4	3096	1498	2170	2209	**1484**
	8	4693	**2544**	3941	3185	2550
	16	6122	3714	4883	4924	**3713**
	32	7536	4930	6521	5905	**4925**
	64	9140	6241	7587	7645	**6224**
`cre_b`	2	20377	755	20196	749	**747**
	4	35977	**1872**	22897	16490	**1872**
	8	47936	3199	35803	18227	**3195**
	16	60360	4708	37028	34440	**4698**
	32	71779	6600	51406	36755	**6564**
	64	84497	9240	54862	53586	**9214**
`tbdmatlab`	2	5033	6485	5034	6486	**5029**
	4	14456	15181	11295	11384	**10857**
	8	30610	26835	20775	19872	**17774**
	16	58132	42140	29571	29370	**28041**
	32	99413	62102	45525	40611	**39381**
	64	152132	93418	56458	56819	**52467**
`nug30`	2	191273	**26417**	191261	26439	26435
	4	287279	56370	212991	185711	**55924**
	8	334237	88525	304462	211728	**88148**
	16	357444	**126220**	328530	320535	126255
	32	369188	167702	375729	347621	**167448**
	64	378197	212567	397624	414942	**212303**
`tbdlinux`	2	**15765**	24902	15766	24965	15767
	4	43147	54759	40822	40507	**30667**
	8	91951	96568	68012	70202	**49096**
	16	172077	154276	98885	100011	**73240**
	32	299114	228033	145532	140671	**105671**
	64	486484	330466	187112	189442	**146771**

cation volume is always desirable. Sometimes the volume completely determines the communication time of a matrix-vector multiplication on a particular computer—for instance, on a simple Ethernet-based PC cluster where communication is serialized. Table 5.4 presents the total communication volume for the partitioning of the rectangular test matrices using the Mondriaan program with five different direction-choosing strategies: a 1D 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; the best-direction strategy that tries both directions and chooses the best. The number of processors ranges between 2 and 64; the computational load imbalance specified is $\epsilon = 0.03$, which is the value used in the experiments reported in [10, 11, 12]. Each result in the table represents the average over 100 runs of the Mondriaan program (each run with a different random number seed). The results have been rounded to the nearest integer. The maximum standard deviation obtained for the results of the matrix `df1001` is 21%; for the other four matrices, it is 10%, 2.4%, 3.7%, and 2.2%, respectively. Often, the standard deviation is smaller.

Table 5.5 *Communication volume of p-way partitioning for square test matrices. The lowest volume for the first five strategies is marked in boldface.*

Name	p	1D row	1D col	2D alt row	2D alt col	2D best dir	2D eq best dir
west0381	2	**50**	56	**50**	56	**50**	185
	4	179	255	194	220	**177**	371
	8	417	510	456	488	**409**	606
	16	792	864	772	753	**692**	907
	32	(1213)	(1317)	1207	1228	**1124**	1404
	64			1675	1644	(**1471**)	(1830)
gemat11	2	92	**58**	92	**58**	**58**	1255
	4	185	180	268	147	**134**	2310
	8	335	388	421	329	**286**	3592
	16	520	631	668	558	**474**	4646
	32	838	1066	1020	977	**814**	5657
	64	1869	2376	2010	1891	**1657**	6861
memplus	2	2755	2543	2741	2543	**2541**	2539
	4	4910	4710	5210	5132	**4561**	4562
	8	6566	6450	7112	7143	**6183**	6220
	16	8312	8155	8928	8845	**7886**	7886
	32	9864	9721	10536	10443	**9255**	9262
	64	11787	11629	12391	12313	**11075**	11058
onetone2	2	524	1079	**513**	1082	534	940
	4	1698	2103	1750	1933	**1626**	2661
	8	2545	3090	2621	2854	**2518**	4338
	16	3364	4224	3625	3741	**3319**	5630
	32	4594	5888	4922	5288	**4522**	7853
	64	6563	8726	7349	7331	**6455**	10866
lhr34	2	402	1364	394	1331	**376**	64
	4	1562	2338	1884	1968	**1470**	6457
	8	2476	3127	3024	3424	**2337**	14091
	16	**3945**	6240	6171	6298	4183	21635
	32	**6455**	11801	(11367)	11677	(6724)	29202
	64	(10241)	15421	15501	(15016)	(**9916**)	37535

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 (25 out of 30 problem instances, i.e., matrix/p combinations). In the remaining cases, it is very close to the optimum. For the term-by-document matrices, the best-direction strategy is much better than the best of the purely 1D strategies, for instance, gaining a factor of 2.25 for tbdlinux/64. The alternating-direction strategies also gain, but not as much. For the LP matrices, the best-direction strategy is about as good as the best of the purely 1D strategies. Here, the alternating-direction strategies perform less well; they sometimes force partitioning in unfavorable directions, so that their volume lies between the volumes of the two 1D strategies. Based on this comparison and others, we have made the best-direction strategy the default of our program.

Table 5.5 presents the total communication volume for the partitioning of the square test matrices using the Mondriaan program with six different strategies, namely, the five strategies shown in Table 5.4, each followed by unconstrained vector partitioning, and the best-direction strategy with dummy addition followed by vector partitioning with the constraint $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$. (We found dummy addition almost always to be advantageous.) In a few cases, our program was unable to achieve the specified

load balance. If this happened in one or more runs for a particular problem instance, but the average imbalance over all the runs was still within the specified value, the result is shown parenthesized in the table. For the smallest matrix, `west0381`, the average imbalance is above the specified value for the 1D strategies with $p = 64$. Therefore, we have omitted these results. (This is a hard problem instance, since each processor should obtain 33.7 nonzeros on average, and 34 nonzeros at most.) For the largest matrix, `lhr34`, we have included some results (between brackets) with an average imbalance slightly above the threshold of 3%; for these results, the average was within 3.1% and the excess imbalance was due to a single run (out of a hundred) with a higher imbalance than allowed. The maximum standard deviation obtained for the five matrices is 10%, 18%, 3.4%, 25%, 33%, respectively. The exceptionally high maxima for the matrices `onetone2` and `lhr34` are due to the case $p = 2$; for $p \geq 4$, the maxima are 12% and 14%, respectively.

The results from Table 5.5 show that the best-direction strategy is the best of the first five strategies, winning in 27 out of 30 instances. Compared to the rectangular case, the gains are only modest: the largest gain percentage compared to the best 1D strategy is 26% for `gemat11`/4. Imposing the constraint on the vector distribution causes much extra communication, except for the matrix `memplus`, which has only nonzeros on the diagonal (see Table 5.2), and hence no extra communication. For this matrix, the last two columns of Table 5.5 represent the same strategy.

Table 5.6 presents the total communication volume for the partitioning of the symmetric test matrices using the Mondriaan program with four different partitioning strategies, including the symmetric strategy based on partitioning the lower triangular part of the input matrix. We present results for the 1D row-direction strategy, but we omit the results for the column-direction strategy, because these are almost the same for symmetric matrices. For the same reason, we only present results for the alternating strategy that starts in the row direction. For all strategies, the matrix partitioning is followed by a constrained vector partitioning. All diagonal elements of the symmetric matrices are already nonzero, so that no dummies need to be added during the matrix partitioning and the communication volume does not increase during the vector partitioning; cf. (4.1). The maximum standard deviation obtained for the five matrices is 4.8%, 6.8%, 28%, 17%, and 21%, respectively.

The results of Table 5.6 show that, similar to the rectangular and square case, the best-direction strategy performs best. Exploitation of symmetry is advantageous for the two stochastic matrices, `cage10` and `finan512`. The gain can be up to a factor of 3, for `finan512`/2. (For $p = 2$, symmetric partitioning turns an essentially 1D strategy into a 2D strategy, which is beneficial here.) For the other three matrices, which represent computational grids, symmetry cannot explicitly be used to our advantage. The two finite-element matrices, `bcsstk30` and `bcsstk32`, display gains in the range 5–13% for the best-direction strategy compared to the 1D strategy, for $p \geq 4$. This indicates that even in applications where 1D methods work very well, modest gains can be obtained by using our 2D method.

5.4. Communication Balance. On many advanced architectures such as PC clusters with sophisticated communication switches and massively parallel computers, it is important to balance the communication, which is precisely the aim of our vector partitioning algorithm. A useful metric for expressing the communication balance obtained is the normalized communication time $\hat{T}$, defined as follows. Let $T(s) = \max(N_{\mathrm{s}}(s), N_{\mathrm{r}}(s))$ be the communication time (in units of one data word sent or received) for processor s, where we assume that a processor can send and receive

Table 5.6 *Communication volume of p-way partitioning for symmetric test matrices. The constraint* $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$ *has been imposed for all strategies; the volumes for the corresponding strategies without this constraint are the same. The lowest volume is marked in boldface.*

Name	p	1D eq row	2D eq alt row	2D eq best dir	2D eq best dir lower
cage10	2	2342	2336	2329	**2021**
	4	5427	5131	5065	**4416**
	8	8849	8201	7937	**6980**
	16	12785	11496	11296	**9660**
	32	17393	15221	14880	**13214**
	64	23346	19701	19272	**17679**
hyp_200_2_1	2	**800**	**800**	**800**	**800**
	4	1534	1542	**1526**	1598
	8	2124	2094	**2056**	2401
	16	2848	2835	**2796**	3246
	32	3778	3808	**3739**	4730
	64	5271	5251	**5116**	6581
finan512	2	292	293	292	**100**
	4	811	909	705	**416**
	8	1587	1609	1258	**679**
	16	2115	2268	1861	**1246**
	32	2712	3296	2373	**1814**
	64	9483	10038	**9309**	10326
bcsstk32	2	**1259**	1263	**1259**	1598
	4	2915	2926	**2581**	2995
	8	5373	5581	**4915**	5853
	16	8713	9145	**8254**	9906
	32	13606	14058	**12951**	15082
	64	20560	21131	**19410**	22534
bcsstk30	2	946	946	929	**689**
	4	1991	2116	**1891**	3483
	8	4881	4987	**4350**	5584
	16	9340	9370	**8639**	10247
	32	15855	15791	**14436**	16293
	64	26872	25395	**23464**	25912

simultaneously. Let $T^1(s)$ be the communication time for processor s in phase 1 and $T^3(s)$ the time in phase 3. Then $T = \max_{0\leq s<p} T^1(s) + \max_{0\leq s<p} T^3(s)$ is the total communication time of the matrix-vector multiplication, where we assume that the communication time of a phase is determined by the busiest processor. The normalized communication time is then $\hat{T} = Tp/V$, i.e., the ratio between the time T and the time V/p for perfectly balanced communication. It holds that $1 \leq \hat{T} \leq p$.

Table 5.7 shows the normalized communication time for the test matrices partitioned using the best-direction strategy. As before, the results are averages over 100 runs of our partitioning program. For $p = 2$, near-perfect balance ($\hat{T} \approx 1$) is achieved for all rectangular matrices, due to the preference we give to sending data in the least busy direction when $\lambda_i = 2$ or $\mu_j = 2$. For most matrices, the communication is reasonably well balanced, with $\hat{T} = 3.45$ in the worst case memplus/64. For the term-by-document matrices, the decrease in communication volume obtained by the 2D strategy is somewhat counteracted by the deteriorating communication balance, but the overall time saving is still significant. For example, tbdlinux/64 has $\hat{T} = 1.86$ and hence $T = 9590$ with the 1D column-direction strategy, but only

Table 5.7 *Normalized communication time of p-way partitioning for the 2D best-direction strategy. The constraint* $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$ *has been imposed for the square matrices (second group) and the symmetric matrices (third group). Dummies have been added for the square matrices.*

Name	$p = 2$	4	8	16	32	64
dfl001	1.00	1.12	1.08	1.14	1.39	1.52
cre_b	1.00	1.05	1.18	1.44	1.66	1.91
tbdmatlab	1.00	1.80	2.12	2.08	2.12	2.14
nug30	1.00	1.01	1.26	1.55	1.71	1.90
tbdlinux	1.00	1.71	2.31	2.61	2.89	3.06
west0381	1.22	1.46	1.52	1.65	1.70	(2.03)
gemat11	1.08	1.72	1.84	1.85	1.94	1.96
memplus	1.46	2.14	2.51	3.09	3.26	3.45
onetone2	1.08	1.44	1.66	1.89	2.13	2.92
lhr34	1.00	1.89	1.86	1.94	2.10	2.20
cage10	1.03	1.26	1.57	1.73	1.93	2.15
hyp_200_2_1	1.00	1.28	1.49	1.70	1.91	2.04
finan512	1.00	1.47	1.95	2.26	2.31	2.33
bcsstk32	1.07	1.60	2.06	2.44	2.54	2.72
bcsstk30	1.03	1.62	2.16	2.24	2.37	2.52

$T = 7024$ for the 2D best-direction strategy. One reason for the better balance of the 1D method is that only one large phase must be balanced, instead of two smaller phases.

Rectangular matrices provide the best opportunities for communication balancing, because the input and output vectors can be partitioned independently. Square matrices with a completely nonzero diagonal, such as memplus and all the symmetric matrices, do not provide any opportunity for balancing, because the constraint $\text{distr}(\mathbf{u}) = \text{distr}(\mathbf{v})$ forces v_j and u_j to be assigned 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 vectors are distributed in the same way as the matrix diagonal, as in previous methods [6, 7, 10, 11, 12, 25, 31]. For square matrices with zeros on the diagonal, we have some opportunities for balancing: if the intersection between the owners of row and column j is empty, we can choose one of the owners in the union of the two sets, trying to optimize the communication balance.

5.5. Comparison. To check the quality of our implementation, and in particular that of the splitting function h, we compare several results to previously published ones. The matrix dfl001 was used for testing in [24]. The best result, for a 1D column partitioning with the ML–FM method, was a volume of 5875 for $p = 8$; for the same strategy, our result is 2544. The maximum volume per processor (data words sent or received) is 1022; our result is 674. 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 [24] 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 [24], and 6566 for our 1D row method; the maximum volume is 1339 and 2427, respectively.

The matrix hyp_200_2_1 was used for testing in [7], where the corresponding 200×200 grid was partitioned into "digital circles." (A digital circle is the set of all grid points within a certain Manhattan distance from a center point; see [7] for an

illustration.) For $p = 64$, block partitioning of the grid gave a corresponding volume of 6400; the 1D Mondriaan result is 5271. An approximate digital circle with 625 points has a boundary of 73 neighboring grid points, which would give $V = 4672$ for $p = 64$ if we could fit such circles together. This means that our hypergraph-based partitioning improves considerably upon the block partitioning and comes close to a lower bound for partitioning into digital circles.

The square matrices `gemat11`, `onetone2`, `lhr34`, `finan512`, `bcsstk32`, and `bcsstk30` were used for testing in [10]. We can compare, for instance, `gemat11` with [10, Table 3]. For our 1D 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 0.73, 0.94, 1.15, and 1.45, 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. (The scaled communication volume of a partitioned matrix equals V/n; see [10, Table 3].) We may conclude that our bipartitioning implementation is similar in quality to that of the other hypergraph-based partitioners and that this is a good basis for our 2D partitioner.

5.6. Timings of Parallel Sparse Matrix-Vector Multiplication. To check whether a reduction in communication volume achieved by using a 2D distribution actually leads to a reduction in execution time compared to a 1D distribution, we have implemented the four-phase parallel sparse matrix-vector multiplication algorithm from section 1 and have run it on a 32-processor subsystem of the 1024-processor Silicon Graphics Origin 3800 parallel computer located at SARA in Amsterdam. Each processor of this machine has a MIPS RS14000 CPU with a clock rate of 500 MHz and a theoretical peak performance of 1 Gflop/s, a primary data cache of 32 Kbyte, a secondary cache of 8 Mbyte, and a memory of 1 Gbyte.

As test matrices, we choose the two term-by-document matrices, `tbdmatlab` and `tbdlinux`, because these rectangular matrices display the largest reduction in communication volume, and the largest finite-element matrix, `bcsstk30`, because this symmetric matrix displays only a modest reduction. The contrast between these two types of matrices should give us insight into the trade-off between 1D and 2D distributions. For `bcsstk30`, we impose $\mathrm{distr}(\mathbf{u}) = \mathrm{distr}(\mathbf{v})$.

Our program is a highly optimized implementation of the four-phase algorithm. The data structure used to store the sparse matrix is CRS. All overhead has been removed by preprocessing, so that only the numerical values of vector components v_j and contributions u_{is} are sent during the multiplication, but no indexing or other tagging information. The program has been optimized to take advantage of a 1D distribution by removing two phases and hence an unnecessary global synchronization in the 1D case. The communication is performed using a one-sided put primitive, which is very efficient. All data destined for the same processor are combined into one message. The program has been written in the programming language C and the communication interface BSPlib [29], and it has been run using version 1.4 of the Oxford BSP toolset [28] implementation of BSPlib. We compiled our program using the standard SGI ANSI C-compiler with optimization flags `-O2` for computation and `-flibrary-level 2 bspfifo 500000 -fcombine-puts -fcombine-puts-buffer 256K,128M,4K` for communication.

We performed experiments for one 1D row, 1D column, and 2D partitioning of each problem instance. The measured execution times are given in Table 5.8. The specific partitioning used was obtained by running Mondriaan with the default random number seed. For reference, the resulting communication volumes are also given in Table 5.8; they differ somewhat from the corresponding averages over 100 runs given

Table 5.8 *Communication volume (in data words) and time (in ms) of parallel sparse matrix-vector multiplication on an SGI Origin* 3800. *The lowest volume and time are marked in boldface.*

Name	p	Volume			Time		
		1D row	1D col	2D	1D row	1D col	2D
`tbdmatlab`	1	0	0	0	5.74	**5.71**	5.77
	2	**5056**	6438	**5056**	3.28	3.31	**3.20**
	4	14650	14949	**11005**	2.08	2.06	**1.95**
	8	30982	26804	**17792**	1.62	1.40	**1.34**
	16	56923	42291	**27735**	1.34	1.19	**1.17**
	32	98791	62410	**40497**	1.77	**1.58**	1.70
`tbdlinux`	1	0	0	0	**67.55**	67.61	74.15
	2	**15764**	24463	**15764**	36.65	32.26	**32.16**
	4	42652	54262	**30444**	14.06	12.22	**12.14**
	8	90919	96038	**49120**	6.49	**6.35**	6.62
	16	177347	155604	**75884**	5.22	4.22	**4.20**
	32	297658	227368	**106563**	4.32	4.08	**3.23**
`bcsstk30`	1	0	0	0	50.99	**50.96**	56.18
	2	948	948	**940**	28.37	28.24	**26.04**
	4	**2099**	**2099**	2124	6.00	6.03	**5.83**
	8	5019	5019	**4120**	**2.87**	2.90	2.88
	16	9344	9344	**8491**	**1.53**	1.56	1.64
	32	15593	15593	**14771**	**1.08**	1.12	1.17

in Tables 5.4 and 5.6. The execution time of a matrix-vector multiplication has been obtained by averaging over 100 multiplications, performed as iterations in the main loop of the program. Each such experiment was carried out three times, and the smallest timing value was taken as the result, since this value presumably was least influenced by interference from other activities on the parallel computer. (The system guarantees exclusive access to the CPUs involved, but in standard operating mode it cannot guarantee exclusive access to all machine resources.)

The timings given in Table 5.8 for the term-by-document matrices show that the 2D method performs best in most cases. For small p, the computation time is dominant and the savings in communication time for a 2D method are relatively small compared to the total time. Furthermore, the difference in volume between the best 1D method and the 2D method is small (for $p = 2$, there is no difference). For larger p, communication time becomes more important and the savings become larger. Note, for instance, the savings of over 21% in total time for `tbdlinux`/32, leading to a speedup of 21 compared to the best $p = 1$ time, which is close to the time of an overhead-free sequential program. Table 5.8 reveals superlinear speedups, e.g., 5.6 for `tbdlinux`/4. This must be due to beneficial cache effects. For `tbdmatlab`/32, execution time starts to increase, due to the increase in communication time per data word and the global synchronization time as a function of p. We measured the time of an isolated global synchronization as 0.05 ms for $p = 16$ and 0.14 ms for $p = 32$. If we include message startup costs for an all-to-all communication pattern, these values become 0.33 ms for $p = 16$ and 1.01 ms for $p = 32$. For large p, the reduction in communication volume obtained for `tbdmatlab` does not compensate for the extra synchronization time needed in the 2D case. The extra time is independent of the problem size, and therefore it is less important for the larger problem `tbdlinux`/32.

The timings given in Table 5.8 for the finite-element matrix `bcsstk30` do not show an advantage for the 2D method. The average saving in communication volume

as given by Table 5.6 is small, and for the particular partitioning given by Table 5.8 significant savings only occur for $p \geq 8$. For these values of p, however, the extra synchronization time is larger than the savings in communication time.

6. Conclusions and Future Work. In this work, we have presented a new 2D 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 minimize the true communication volume; it tries to spread the communication evenly; and it is 2D. The experimental results of our implementation, Mondriaan, show that for many matrices this indeed leads to lower communication cost than for a comparable 1D implementation such as Mondriaan in 1D mode. For term-by-document matrices, the gain of the new method is large; for most other test matrices, it is small but noticeable. Somewhat surprisingly, even for finite-element matrices the new method displays a gain.

Our algorithms minimize two metrics, namely, total communication volume and maximum amount of communication per processor. We make no attempt to reduce the number of messages: in the worst case, $2(p-1)p$ messages are sent by all the processors together, in case each processor communicates with all the others in both communication phases of the matrix-vector multiplication. 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 and then choosing the best. This has the advantage that the strategy adapts itself automatically to the matrix, without requiring any prior knowledge.

The main motivation for using 2D partitioning methods is an expected reduction in communication volume, which in turn should lead to performance gains in the actual parallel sparse matrix-vector multiplication, especially for large matrices. We have observed such gains for term-by-document matrices. One-dimensional methods, however, also have their advantages. For instance, rowwise partitioning halves the upper bound on the number of messages since phase 3 can be skipped. This may be particularly important for small matrices and for parallel computers with high startup costs for sending messages. For very sparse matrices with only a few nonzeros per row, 1D partitioning may lead to a lower communication volume. (In principle, the 2D best-direction strategy should detect this automatically and produce a 1D distribution, but this may not always happen.) Furthermore, in the 1D case no redundant additions are performed, because phase 4 can be skipped. The length of the average local row will be larger, since rows are kept complete. This may reduce data-structure overhead in phase 2 and hence improve the computing speed. (A good 2D partitioner tries to keep rows complete as well, and our program Mondriaan often succeeds in this.) One-dimensional methods respect the connection between a matrix row and a variable (i.e., a vector component), allowing us to store all related data together on one processor. This may be important in particular applications, e.g., in certain finite-element computations. Two-dimensional methods, however, break this connection, since they may spread the matrix elements of a row over several processors. This requires explicit assembly of the matrix and perhaps even the use of a different data structure in the iterative solver part of an application. The trade-off between these concerns depends, of course, on the specific problem at hand. Overall, we expect that for large problems the gains in communication performance obtained by using 2D partitioning are well worthwhile.

To achieve our goals, we had to generalize 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 into 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 a generalized algorithm which can handle all values of p, but we have implemented it only for powers of 2; the Mondriaan program should be adapted to the general case. Third, 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. Fourth, 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. For this purpose, an object-oriented iterative linear system solver package called Parallel Templates, running on top of MPI-1 or BSPlib, has already been developed by Koster [34]. Fifth, the partitioning itself should be done in parallel to enable solving very large problems that do not fit in the memory 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.

Acknowledgments. We are grateful to the anonymous referees for their comments, which greatly helped to improve this paper. We thank the Dutch national computing facilities foundation NCF and the SARA computing center in Amsterdam for providing access to an SGI Origin 3800.

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, 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, 1994.

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

[4] M. W. Berry, Z. Drmač, and E. R. Jessup, *Matrices, vector spaces, and information retrieval*, SIAM Rev., 41 (1999), pp. 335–362.

[5] M. L. Bilderback, *Improving unstructured grid application execution times by balancing the edge-cuts among partitions*, in Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, CD-ROM, SIAM, Philadelphia, 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., Inst. Math. Appl. Conf. Ser. New Ser. 46, 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., IFIP Trans. A 51, Elsevier, 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. N. Bui and C. Jones, *A heuristic for reducing fill-in in sparse matrix factorization*, in Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1993, pp. 445–452.

[10] Ü. V. Çatalyürek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Trans. Parallel Distrib. Systems, 10 (1999), pp. 673–693.

[11] Ü. V. Çatalyürek and C. Aykanat, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, in Proceedings of the 8th International Workshop on Solving Irregularly Structured Problems in Parallel, IEEE, Los Alamitos, CA, 2001, p. 118.

[12] Ü. V. Çatalyürek and C. Aykanat, *A hypergraph-partitioning approach for coarse-grain decomposition*, in Proceedings of Supercomputing 2001, ACM, New York, 2001, p. 42.

[13] T. A. Davis, *University of Florida Sparse Matrix Collection*, http://www.cise.ufl.edu/research/sparse/matrices, 1994–2003.

[14] I. S. Duff, R. G. Grimes, and J. G. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Software, 15 (1989), pp. 1–14.

[15] I. S. Duff, R. G. Grimes, and J. G. Lewis, *The Rutherford–Boeing Sparse Matrix Collection*, Technical Report TR/PA/97/36, CERFACS, Toulouse, France, 1997.

[16] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, in Proceedings of the 19th Design Automation Conference, IEEE, Los Alamitos, CA, 1982, pp. 175–181.

[17] R. Fletcher, *Conjugate gradient methods for indefinite systems*, in Proceedings of the Dundee Biennial Conference on Numerical Analysis, G. A. Watson, ed., Lecture Notes in Math. 506, Springer-Verlag, Berlin, 1976, pp. 73–89.

[18] 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.

[19] R. W. Freund and N. M. Nachtigal, *QMR: A quasi-minimal residual method for non-Hermitian linear systems*, Numer. Math., 60 (1991), pp. 315–339.

[20] D. M. Gay, *Electronic mail distribution of linear programming test problems*, MPS COAL Newsletter, 13 (Dec. 1985), pp. 10–12.

[21] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., The Johns Hopkins University Press, Baltimore, MD, 1996.

[22] B. Hendrickson, *Graph partitioning and parallel solvers: Has the emperor no clothes?*, in Proceedings of the 5th International Workshop on Solving Irregularly Structured Problems in Parallel, A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng, eds., Lecture Notes in Comput. Sci. 1457, Springer-Verlag, Berlin, 1998, pp. 218–225.

[23] B. Hendrickson and T. G. Kolda, *Graph partitioning models for parallel computing*, Parallel Comput., 26 (2000), pp. 1519–1534.

[24] B. Hendrickson and T. G. Kolda, *Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing*, SIAM J. Sci. Comput., 21 (2000), pp. 2048–2072.

[25] B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, in Proceedings of Supercomputing 1995, ACM, New York, 1995.

[26] B. Hendrickson, R. Leland, and S. Plimpton, *An efficient parallel algorithm for matrix-vector multiplication*, Internat. J. High Speed Comput., 7 (1995), pp. 73–88.

[27] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, J. Res. Nat. Bur. Standards, 49 (1952), pp. 409–436.

[28] J. M. D. Hill, S. R. Donaldson, and A. McEwan, *Installation and User Guide for the Oxford BSP Toolset (*v1.4*) Implementation of BSPlib*, Technical Report, Oxford University Computing Laboratory, Oxford, UK, 1998.

[29] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, *BSPlib: The BSP programming library*, Parallel Comput., 24 (1998), pp. 1947–1980.

[30] Y. F. Hu, K. C. F. Maguire, and R. J. Blake, *A multilevel unsymmetric matrix ordering algorithm for parallel process simulation*, Comput. Chem. Engrg, 23 (2000), pp. 1631–1647.

[31] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

[32] G. Karypis and V. Kumar, *Parallel multilevel k-way partitioning scheme for irregular graphs*, SIAM Rev., 41 (1999), pp. 278–300.

[33] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Tech. J., 49 (1970), pp. 291–307.

[34] J. H. H. Koster, *Parallel Templates for Numerical Linear Algebra: A High-Performance Computation Library*, Master's thesis, Mathematical Institute, Utrecht University, Utrecht, The Netherlands, 2002.

[35] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley, New York, 1990.

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

[37] A. T. Ogielski and W. Aiello, *Sparse matrix computations on parallel processor arrays*, SIAM J. Sci. Comput., 14 (1993), pp. 519–530.

[38] A. Pinar and C. Aykanat, *An effective model to decompose linear programs for parallel solution*, in Proceedings of PARA '96, J. Waśniewski, J. Dongarra, K. Madsen, and D. Olesen, eds., Lecture Notes in Comput. Sci. 1184, Springer-Verlag, Berlin, 1997, pp. 592–601.

[39] A. Pinar and C. Aykanat, *Sparse matrix decomposition with optimal load balancing*, in Proceedings of the International Conference on High Performance Computing, IEEE, 1997, pp. 224–229.

[40] A. Pinar, Ü. V. Çatalyürek, C. Aykanat, and M. Pinar, *Decomposing linear programs for parallel solution*, in Proceedings of PARA '95, J. Dongarra, K. Madsen, and J. Waśniewski, eds., Lecture Notes in Comput. Sci. 1041, Springer-Verlag, Berlin, 1996, pp. 473–482.

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

[42] Y. Saad and M. H. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 7 (1986), pp. 856–869.

[43] H. A. van der Vorst, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 13 (1992), pp. 631–644.

[44] A. van Heukelum, G. T. Barkema, and R. H. Bisseling, *DNA electrophoresis studied with the cage model*, J. Comput. Phys., 180 (2002), pp. 313–326.

[45] B. Vastenhouw, *A Parallel Web Search Engine Based on Latent Semantic Indexing*, Master's thesis, Mathematical Institute, Utrecht University, Utrecht, The Netherlands, 2001.

[46] C. Walshaw and M. Cross, *Multilevel mesh partitioning for heterogeneous communication networks*, Fut. Gen. Comput. Syst., 17 (2001), pp. 601–623.