SIAM J. SCI. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 21, No. 6, pp. 2048–2072

PARTITIONING RECTANGULAR AND STRUCTURALLY UNSYMMETRIC SPARSE MATRICES FOR PARALLEL PROCESSING[*]

BRUCE HENDRICKSON[†] AND TAMARA G. KOLDA[‡]

Abstract. A common operation in scientific computing is the multiplication of a sparse, rectangular, or structurally unsymmetric matrix and a vector. In many applications the matrix-transpose-vector product is also required. This paper addresses the efficient parallelization of these operations. We show that the problem can be expressed in terms of partitioning bipartite graphs. We then introduce several algorithms for this partitioning problem and compare their performance on a set of test matrices.

Key words. matrix partitioning, iterative method, parallel computing, rectangular matrix, structurally unsymmetric matrix, bipartite graph

AMS subject classifications. 05C50, 65F10, 65F50, 65Y05

PII. S1064827598341475

1. Introduction.

Matrix-vector and matrix-transpose-vector products that repeatedly involve the same large, sparse, structurally unsymmetric or rectangular matrix arise in many iterative algorithms. Examples include algorithms for solving linear systems, least squares problems, and linear programs. To efficiently implement these types of methods in parallel, the nonzeros of the sparse matrix must be distributed among processors in such a way that the computational work per processor is balanced and the interprocessor communication is low. This can usually be achieved by an appropriate partitioning of the matrix. Specifically, given a structurally unsymmetric or rectangular matrix A, the key is to find permutations P and Q so that the nonzero values of PAQ are clustered in the diagonal blocks as illustrated in Figure 1.1. As we show in section 3, this nearly block diagonal structure helps reduce the communication cost in matrix-vector products. Furthermore, by requiring that the block rows (or block columns) have approximately the same number of nonzeros, the floating point operations are well balanced among processors.[1]

Despite the utility of rectangular or structurally unsymmetric matrix partitioning, little work has been done in this area until recently. If the matrix is square and structurally symmetric, the problem can be expressed in terms of graph partitioning, and a number of good algorithms and software tools have been developed for this use [24, 29, 45]. These methods can be used for partitioning a square, structurally

[*]Received by the editors July 6, 1998; accepted for publication (in revised form) April 13, 1999; published electronically May 17, 2000. This work was supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy, under contracts DE–AC05–96OR22464 and DE-AC04-94AL85000 with Lockheed Martin Energy Research Corporation. A preliminary version of this work appeared in Applied Parallel Computing in Large Scale Scientific and Industrial Problems, 4th International Workshop, PARA'98, Umeå, Sweden, 1998.

http://www.siam.org/journals/sisc/21-6/34147.html

[†]Parallel Computing Sciences Dept., Sandia National Laboratories, Albuquerque, NM 87185–1110 (bah@cs.sandia.gov).

[‡]Computational Sciences and Mathematics Research Dept., Sandia National Laboratories, Livermore, CA 94551–9217 (tgkolda@sandia.gov). The work of this author was performed while at a postdoctoral fellowship at Oak Ridge National Laboratory, Oak Ridge, TN.

[1]Note that our approach is specifically targeted for sparse matrices. For dense matrices or sparse matrices with nonzero patterns that are difficult to exploit, two-dimensional decompositions are typically used; see Hendrickson, Leland, and Plimpton [27] or Lewis and van de Geijn [36].

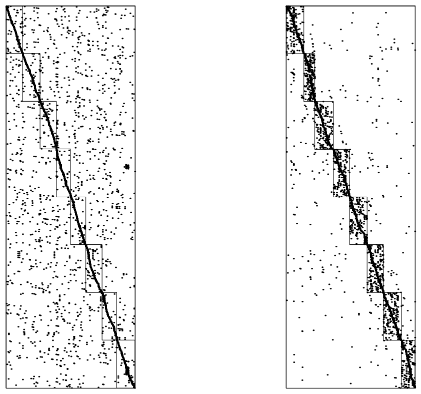

FIG. 1.1. *Matrix before and after partitioning.*

unsymmetric matrix A by considering the sparsity pattern of the $A + A^T$ matrix. But this trick is appropriate only if the matrix is nearly structurally symmetric. The square symmetric methods are not applicable to rectangular matrices.

In section 3, we describe the matrix-vector and matrix-transpose-vector kernels and show how the partitioning affects communication. Further, we show that we need only to use the row partition to maintain balance in the number of nonzeros per processor and consequently have some leeway in the column partition that we can exploit for other purposes. For example, in the case of preconditioned iterative methods for structurally unsymmetric matrices, we can use this freedom to find a partition that is good for both the matrix *and* its explicit preconditioner. We discuss this further in sections 2–4.

In section 4, we describe the relationship between matrix partitioning and graph partitioning. An $m \times n$ rectangular or structurally unsymmetric matrix corresponds to a *bipartite* graph on $m + n$ nodes with the number of edges equal to the number of nonzeros in the matrix. We show that the matrix partitioning problem can be described as a bipartite graph partitioning problem in which edge cuts are related to parallel communication and constraints on the partition sizes correspond to work load per processor.

In section 5, several algorithms for partitioning the bipartite graphs are presented. Modifications of the well-known spectral [39], Kernighan–Lin [31]/Fiduccia–Mattheyses [12], and multilevel [6, 26, 29, 30] methods are given for the bipartite graph model. The modification of the spectral method was previously introduced by Berry, Hendrickson, and Raghavan [5]. Further, the alternating partitioning method of Kolda [33] is presented; this method is specific to the bipartite case.

Finally in section 6, we measure the performance of various methods for partitioning rectangular or structurally unsymmetric matrices. We compare different methods on a collection of matrices from least squares, linear programming, truncated singular value decomposition (SVD), and preconditioned linear systems. Our results indicate that the best approach is generally the multilevel method with either Fiduccia–Mattheyses or alternating partitioning combined with Fiduccia–Mattheyses refinement.

Several authors have used partitioned bipartite graphs of matrices for different parallelization objectives. Ferris and Horn [11] find vertex separators in the bipartite graph to reorder the matrix into arrowhead form. This is useful in the parallelization of linear programs and other optimization problems. Coon and Stadtherr [8] use a bipartite graph partitioning model to identify parallelism in sparse Gaussian elimination. Neither of these efforts addresses the problem of parallelizing matrix vector products.

Previous attempts to address the general matrix partitioning problem for matrix vector multiplication include the work of Kolda [33] and an earlier report on this research [23].

The authors have recently become aware of a closely related work by Çatalyürek and Aykanat [7]. In their approach, the structure of the unsymmetric matrix is represented by a hypergraph in which rows are vertices and columns are hyperedges. A distinct advantage of their approach over ours is that their partitioning problem correctly models the communication volume in parallel matrix-vector multiplication; as we discuss in section 4, our approach only models the communication volume approximately. Balanced against this weakness, our approach has several advantages relative to theirs. Unlike the approach of Çatalyürek and Aykanat, our method treats rows and columns equivalently and generates a well-defined partition of each. Also, as we discuss in section 3, our approach allows us to model the application of both a matrix and some kinds of preconditioners. We will return to the issue of the relationship between these two approaches in our conclusions.

2. Applications.

Matrix-vector products involving sparse, rectangular, or structurally unsymmetric matrices occur in a wide variety of numerical methods. One very important example is the solution of a unsymmetric system

$$Ax = b,$$

with an iterative method such as biconjugate gradient (BiCG) [14] or standard quasi-minimal residual (QMR) [16]. During each iteration, these methods require the computation of Ar and $A^T s$ for some vectors r and s. (It is worth noting that there are solvers for unsymmetric systems that do not require the $A^T s$ product, e.g., transpose-free QMR [15]). To use the partitioned matrix, PAQ, we can solve

$$(PAQ)y = Pb,$$

where $Q^T x = y$. Note that permuting the rows and columns of a matrix changes its eigenvalues; however, because we do not know the exact role that eigenvalues play in these methods, we cannot predict whether the effect will be positive or negative. In this case, the number of rows and columns assigned to each partition must be equal so that the diagonal blocks of PAQ are square and the data layout of the vectors is correct for other parallel operations (like dot products). If A is structurally symmetric or nearly so, a symmetric partitioning scheme is likely more appropriate. (Furthermore, a symmetric reordering keeps the eigenvalues intact.)

Generally, iterative methods involve preconditioning. Suppose we have an explicit preconditioner such as an *approximate inverse* $M \approx A^{-1}$. (See Benzi and Tůma [3] for a survey of approximate inverse preconditioners.) In that case, we need to find P and Q such that both PAQ and $Q^T M P^T \approx (PAQ)^{-1}$ are well partitioned. By well partitioned we mean that (1) the communication costs are low, (2) the block rows of PAQ are balanced (i.e., have approximately equal numbers of nonzeros), and (3) the block rows of $Q^T M P^T$ are balanced. Note that conditions (2) and (3) are stronger than merely requiring that the block rows of $P(A + M^T)Q$ are balanced, and these conditions are necessary because there is usually a synchronization point between the application of the matrix and the preconditioner. Once a particular P and Q are determined, in the case of left preconditioning we need to solve

$$(Q^T M P^T)(PAQ)y = (Q^T M)b,$$

where $y = Q^T x$. In essence, we need only reorder the variables according to Q^T throughout the iterative method. If M is a right preconditioner, we solve

$$(PAQ)(Q^T M P^T)y = Pb,$$

where $y = PM^{-1}x$. In this case, we reorder the variables throughout the method by P. Note that we may even use this idea when AM or MA is symmetric positive definite and a method such as (preconditioned) conjugate gradients [19] is used.

Like iterative methods for linear systems, iterative methods for least squares problems require numerous matrix-vector products, and in this case, the matrices are rectangular. Consider a system of the form

$$\min \|Ax - b\|_2,$$

where A is an $m \times n$ matrix with $m > n$. This problem can be solved by iterative methods such as LSQR [38] that require computations of the form Ar and $A^T s$ every iteration. Using the permuted matrix does not change the minimal value of the least squares objective function.

Another situation in which A is rectangular arises in interior point methods for linear programming,

$$\begin{array}{rl} \min & c^T x \\ \text{s.t.} & Ax = b, \\ & x \geq 0. \end{array}$$

Here A is a real $m \times n$ matrix with $m \leq n$. At each iteration of the method, the next search direction is computed by solving the set of equations

$$\begin{bmatrix} D & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} w \\ v \end{bmatrix}, \tag{2.1}$$

where y is the dual variable and D is a diagonal matrix that changes each iteration. Alternatively, we may solve the normal equations,

$$(AD^{-2}A^T)\Delta y = r.$$

See Wang and O'Leary [46] for an algorithm that solves these equations iteratively as well as an overview of other such methods. When iterative solvers are employed, frequent multiplications involving A and A^T are needed. Even when using direct methods, multiplies by A and A^T are required to compute w and v or r at each iteration. Permuting A does not change the eigenvalues of either of the two systems mentioned previously.

Lastly, computing the truncated SVD of a large sparse matrix A via a Lanczos procedure requires frequent multiplies by A and A^T. This arises in, for example, latent semantic indexing for information retrieval [4], clustering for hypertext matrices [5], and geophysical applications [43]. Permuting A does not change its singular values, and the singular vectors of the original matrix are just permutations of those for the permuted matrix.

3. Parallel matrix-vector multiplication. Since matrix-vector multiplications are ubiquitous numerical kernels, it is important to devise effective algorithms for their parallel execution. To perform this operation efficiently, we must evenly divide the computational load while requiring a minimum amount of communication. In this section we show how matrix partitioning can be used to obtain this objective for the matrix-vector and matrix-transpose-vector multiply operations.

Suppose that an $m \times n$ matrix A has already been reordered and partitioned into a block $p \times p$ structure,

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{bmatrix}, \tag{3.1}$$

where p is the number of processors. Here A_{ij} is of size $m_i \times n_j$, where $\sum_i m_i = m$ and $\sum_j n_j = n$. We assume that most of the nonzeros are on the block diagonal as a result of the partitioning.

We present algorithms for a *row-based partitioning*; that is, each processor is assigned a block row, and we assume that the m_i's have been chosen in such a way that the number of nonzeros per block row is nearly equal. For now we assume nothing about the n_j's. The algorithm we describe for computing Ax is widely used; see, e.g., [42].

Analogous algorithms exist for a *column-based partitioning*. Specifically, if we have a matrix that is partitioned into block columns, we can simply work with the transpose of the matrix that is partitioned by rows.

3.1. Matrix-vector multiply (row-based). For the row-based algorithm, processor i owns the ith block row of A, that is,

$$\begin{bmatrix} A_{i1} & A_{i2} & \cdots & A_{ip} \end{bmatrix}.$$

To compute the product $y = Ax$ in parallel, divide the vector x into conformal block format,

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix},$$

where block x_i is of length n_i. Processor i holds x_i.

Consider the procedure from the point of view of processor i. First, a message is sent to each processor $j \neq i$ for which $A_{ji} \neq 0$. This message contains only those elements of x_i corresponding to nonzero *columns* in A_{ji}. While the processor waits to receive messages, it computes the contribution from the diagonal matrix block,

$$y_i^{(i)} = A_{ii} x_i.$$

The block A_{ii}, while still sparse, may be dense enough to exhibit good data locality. Then, for each $j \neq i$ such that A_{ij} is nonzero, a message is received containing a sparse vector $\bar{x}_j$ that has only the elements of x_j corresponding to nonzero columns in A_{ij}, and

$$y_i^{(j)} = A_{ij} \bar{x}_j$$

is computed. (We assume that processor i already knows which elements to expect from processor j.) Finally, the ith block of the product y is computed via the sum

$$y_i = \sum_j y_i^{(j)}.$$

Block y_i is of size m_i.

3.2. Matrix-transpose-vector multiply (row-based). In the row-based method, to compute $z = A^T v$, processor i holds v_i, the ith block of v of size m_i, and the ith block row of A. As before, the procedure is sketched from processor i's point of view. First, the off-diagonal blocks are used to compute

$$z_j^{(i)} = A_{ij}^T v_i$$

for each $j \neq i$ for which $A_{ij} \neq 0$. Observe that the number of nonzeros in $z_j^{(i)}$ is equal to the number of nonzero rows in A_{ij}^T, i.e., the number of nonzero columns in A_{ij}. Next, processor i sends to each other processor $j \neq i$, the nonzero[2] elements of $z_j^{(i)}$, if any. While waiting to receive messages from the other processors, processor i computes the diagonal block contribution

$$z_i^{(i)} = A_{ii}^T v_i.$$

Next, from each processor j such that $A_{ji} \neq 0$, it receives $\bar{z}_i^{(j)}$, which contains only the nonzero elements of $z_i^{(j)}$. (Again, we assume that processor i already knows which elements to expect from processor j.) Finally, processor i computes the ith component of the product,

$$z_i = z_i^{(i)} + \sum_{j \neq i} \bar{z}_i^{(j)}.$$

Block z_i is of size n_i.

3.3. Analysis. We now present some facts for the row-based kernels; analogous facts exist for the column-based kernels.

In both the matrix-vector and matrix-transpose-vector algorithm, a processor is responsible for the multiplication associated with the matrix blocks it owns. This leads to the following fact.

FACT 1. *The number of multiplies that processor i performs in either the matrix-vector or matrix-transpose-vector operations is equal to the number of nonzeros in block row i.*

Thus, the workload per processor is the same for both the matrix-vector and matrix-transpose-vector multiplies. If the partitioning process ensures that the numbers of nonzeros per block row are nearly equal, the computational workload per processor will be balanced.

Recall that a message goes from i to j in computing Ax if A_{ji} is nonzero, and only the elements of x_i corresponding to nonzero columns in A_{ji} are sent. This leads to the following.

[2] Here we mean any elements that are not guaranteed to be zero by the structure of A_{ij}. Elements that are zero by cancellation are still communicated.

FACT 2. *The number of messages sent by processor i in the matrix-vector multiply is equal to the number of nonzero blocks A_{ji} with $j \neq i$. Further, the volume of messages sent by processor i is the sum of the number of nonzero columns in each A_{ji} with $j \neq i$.*

Similarly, a message goes from i to j in computing $A^T v$ if A_{ij} is nonzero, and only the nonzero elements of $z_j^{(i)}$ are sent.

FACT 3. *The number of messages sent by processor i in the matrix-transpose-vector multiply is equal to the number of nonzero A_{ij} with $j \neq i$. Further, the volume of messages sent by processor i is the sum of the number of nonzero columns in A_{ij} with $j \neq i$.*

Combining Facts 2 and 3 yields the following three facts.

FACT 4. *The total number of messages sent in either the matrix-vector or matrix-transpose-vector multiply is equal to the number of nonzero off-diagonal blocks.*

FACT 5. *If a message is sent from processor i to processor j in the matrix-vector multiply, then a message of the same length will be sent from processor j to processor i in the matrix-transpose-vector multiply.*

This means that the matrix-vector and matrix-transpose-vector multiplies share the same communication pattern with the direction of the messages reversed.

FACT 6. *In either the matrix-vector or matrix-transpose-vector multiply, the total message volume is equal to the sum of the number of nonzero columns in each off-diagonal block.*

As our numerical results in section 6 show, reducing the total number of nonzeros in the off-diagonal blocks typically reduces the total message volume and the maximum message volume handled by a single processor, but the relationship between these different quantities can be complex.

It is useful to observe that a single decomposition can lead to efficient matrix-vector *and* matrix-transpose-vector products, and this helps facilitate parallelization of the applications described in section 2.

We note that the amount of information about the structure of A required by each processor is small. Specifically, to decide what to receive when performing $y = Ax$, processor i needs only know which columns are nonzero within each nondiagonal block of block row i (which it owns). Similarly, when sending data, processor i must only know which columns within each nondiagonal block of block column i are nonzero. Since processor i does not own block column i, this small amount of data is communicated in a set-up phase before the numerical computation.

In the preceding discussion, we assumed that the m_i's are chosen so that the nonzeros per block row (and hence the work per processor) are balanced. We made no assumption about the n_j's, and we can exploit this freedom in several ways.

(1) Choose the n_j's to minimize communication in the matrix-vector products. This is accomplished by leaving the n_j's unconstrained.

(2) Choose the n_j's to each be nearly equal, which would balance `BLAS-1` operations on the n-long vectors. These operations are a component of most iterative methods.

(3) As discussed further in the next section, if we have an approximate inverse preconditioner, say $M \approx A^{-1}$, we can simultaneously partition A and M. Our partitioned matrices are given by PAQ and $Q^T M P^T$. We can choose the m_i's to balance the work associated with A and the n_j's to likewise balance the effort of computing with M.

As mentioned earlier, a matrix can be partitioned by rows *or* columns, whichever

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \times & & \times & \times & \times \\ 2 & & \times & & \times & \\ 3 & \times & & \times & & \times \\ 4 & \times & \times & & \times & \times \\ 5 & \times & & \times & \times & \times \end{array}$$

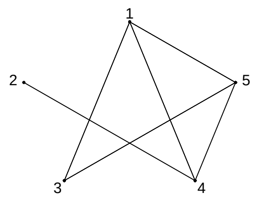

FIG. 4.1. *Graph of a symmetric matrix.*

leads to better performance. For example, consider a row partitioning of a matrix that has dense rows but no dense columns. It may be difficult to balance the load since a single processor is saddled with all the nonzeros in the dense row. Furthermore, the processor owning the dense row will need to receive a large amount of information to compute its contribution to Ax. Partitioning the matrix by columns resolves these problems. Not only is the load balancing problem easier, but the communication volume now depends on the nonzero *rows* in the off-diagonal blocks. A dense row will contribute only one nonzero row to any block that contains it, so the communication volume will generally be reduced.

4. A bipartite graph model.

As discussed in section 3, the key to an efficient parallel matrix-vector multiplication algorithm is in the partitioning of the rows and columns of the matrix. For structurally symmetric matrices, this problem has been well studied and is generally phrased in terms of graph partitioning. The structure of an $n \times n$ structurally symmetric matrix $A = [a_{ij}]$ can be described by an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \{1, 2, \ldots, n\}$ and $(i, j) \in \mathcal{E}$ if and only if a_{ij} (and hence a_{ji}) is nonzero (see Figure 4.1). Vertices and edges can have weights if desired. A partitioning of the vertices of $\mathcal{G}$ corresponds to a symmetric partitioning of the rows and columns of A. For example, a division of the vertices into two sets induces a block 2×2 structure for the matrix. Each edge that crosses between the two sets corresponds to a nonzero value in the off-diagonal blocks of the matrix. The standard approach to structurally symmetric matrix partitioning is to try to minimize these cross edges, while maintaining some balance on the number of rows (or the number of nonzeros) in the two sets. This *graph bisection problem* is known to be NP-hard [17].

This approach is not well suited to rectangular or structurally unsymmetric matrix partitioning. If the matrix is rectangular, then the graph model does not apply. If the matrix is square, the standard graph model can only encode a symmetric structure. A directed graph model can encode nonsymmetry in a square matrix, but more generally, these approaches force the row partition to be identical to the column partition since a vertex in the graph represents both a row and a column of the matrix. Although this is reasonable for structurally symmetric matrices, it is unnecessarily restrictive for structurally unsymmetric ones; that is, a better partition may be achieved by allowing the rows and columns to be partitioned separately.

For the rectangular or structurally unsymmetric case, an alternate graph model of the matrix can be used. The nonzero structure of an $m \times n$ matrix $A = [a_{ij}]$ corresponds to an undirected *bipartite* graph $\mathcal{G} = (\mathcal{R}, \mathcal{C}, \mathcal{E})$ with $\mathcal{R} = \{r_1, \ldots, r_m\}$, $\mathcal{C} = \{c_1, \ldots, c_n\}$, and $(r_i, c_j) \in \mathcal{E}$ if and only if $a_{ij} \neq 0$ (see Figure 4.2). Note that no edge connects two rows or two columns. If desired, edges and vertices can have weights assigned to them. A partitioning of the vertices in $\mathcal{R}$ induces a division of the rows of the matrix; likewise, a partitioning of the $\mathcal{C}$ vertices corresponds to a division of columns. Unlike the standard graph model, the bipartite model allows a

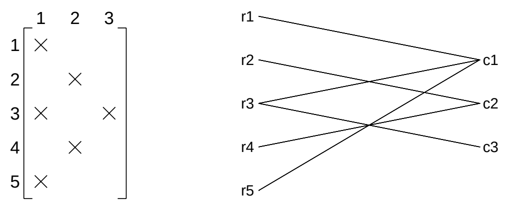

FIG. 4.2. *Bipartite graph of a matrix.*

different number of row and column vertices and can represent unsymmetric structure. Further, the row and column partitions are separate.

More formally, we propose the following *bipartite graph partitioning problem.* Given a bipartite graph $\mathcal{G} = (\mathcal{R}, \mathcal{C}, \mathcal{E})$ with weighted edges and vertices, we wish to find p disjoint partitions $\mathcal{P}_i \equiv \mathcal{R}_i \cup \mathcal{C}_i$ with $\mathcal{R}_i \subseteq \mathcal{R}$ and $\mathcal{C}_i \subseteq \mathcal{C}$ such that the following three criteria are satisfied.

(1) The total weight of edges crossing between partitions is minimized.
(2) There is a bound (possibly infinite) on the maximum difference in total row vertex weight between any two partitions.
(3) There is a bound (possibly infinite) on the maximum difference in total column vertex weight between any two partitions.

This is a generalization of the standard graph partitioning problem.

The matrix partitioning problem from the matrix-vector multiply in section 3 can be expressed in the bipartite graph partitioning model. Suppose we want to divide the matrix over p processors. As discussed in section 3 this can be accomplished by either a row-based or a column-based partition. Without loss of generality, we will focus on the row-based option. Assign each vertex $r_i \in \mathcal{R}$ a weight equal to the number of nonzeros in row i of A. This weight corresponds to the number of multiplication operations a processor will have to perform if it owns this row. Let edges and column vertices have unit weights. Now apply bipartite graph partitioning so that (1) the total number (or weight) of edges crossing between the partitions ($\mathcal{P}_i = \mathcal{R}_i \cup \mathcal{C}_i,\ i = 1, \ldots, p$) is minimized and (2) the total vertex weight in each set $\mathcal{R}_i$ is approximately equal. The first constraint leads to low communication while the second ensures load balance. Such a partitioning corresponds to a nearly block diagonal structure for the matrix. Note that no constraints on column balance are necessary; that is, the bound in condition (3) of the bipartite graph partitioning problem is infinite.

Several caveats are necessary. First, with weights on the vertices, perfect load balance may be difficult or impossible to achieve since the set of vertex weights may not divide into equally sized sets. In practice it is much simpler to merely require that the difference between the total vertex weights in $\mathcal{R}_i$ and $\mathcal{R}_j$ be less than or equal to the maximum weight of any single row vertex. Second, with no restrictions on the column vertices we can divide them in any way—perhaps even assigning no columns to a given partition if that is what is best for the communication pattern. Third, the edges are each given weight 1, but other edge weighting schemes are possible. For example, we could weight an edge from r_i to c_j by $|a_{ij}|$ if, for some reason, we want to encourage large matrix values to be in the block diagonal. Lastly, as discussed in section 3, the communication volume induced by a partition is not equal to the number of graph edges cut but rather to the number of columns in the off-diagonal blocks that have nonzeros in them. This column count is nicely captured in the hypergraph model of Çatalyürek and Aykanat [7], but it can also be expressed in the bipartite graph model. Specifically, each of these nonzero columns corresponds to a vertex with neighbors in another partition. However, this more accurate metric is more difficult to model and minimize than the number of edges cut for two reasons. First, the number of vertices with neighbors in another partition is a function that changes in a discontinuous manner. Reducing the number of neighbors from 2 to 1 does not reduce the cost, but reducing it from 1 to 0 does. This is problematic for a local search algorithm. Second, there is a large body of literature on graph partitioning to minimize edge cuts that we could draw upon. For these reasons, we choose to focus on edge cuts as an approximation to the true communication volume. We will discuss the appropriateness of this approximation further in section 7. It is worth noting that the same approximation is used (although not widely acknowledged) in the standard graph partitioning model [22].

By not constraining the partition of the columns, we allow for whatever partition leads to the minimal number of edge cuts. Other possible objectives are discussed in section 3.3. One alternative is to balance the vector (i.e., `BLAS-1`) operations associated with the n-long vectors. This can be accomplished by setting the weight of each column vertex to 1 and adding the additional constraint (3) that the difference in total vertex weight between any pair $\mathcal{C}_i$ and $\mathcal{C}_j$ be no more than 1 (i.e., the maximum vertex weight in $\mathcal{C}$).

The other objective mentioned in section 3.3 is to enable efficient matrix-vector products for two matrices simultaneously, as in the case when an approximate inverse preconditioner is employed in an iterative method to solve a linear system. Specifically, for square A and M, we want to find P and Q such that PAQ and Q^TMP^T (or equivalently, PM^TQ) are both well partitioned. We can address this by partitioning an appropriately weighted bipartite graph. Before there was an edge from r_i to c_j, if a_{ij} was nonzero and each edge was weighted as 1. Now, $(r_i, c_j) \in \mathcal{E}$ if *either* a_{ij} or m_{ji} is nonzero. Further, the weight of the edge from r_i to c_j is

$$w(r_i, c_j) = \begin{cases} 2 & \text{if } a_{ij} \neq 0 \quad \textbf{and} \quad m_{ji} \neq 0, \\ 1 & \text{if } a_{ij} \neq 0 \quad \textbf{or} \quad m_{ji} \neq 0. \end{cases}$$

The weight of vertex c_j is equal to the number of nonzeros in column j of M^T (or row j of M). We add the condition (3) that the difference in total vertex weight between any pair $\mathcal{C}_i$ and $\mathcal{C}_j$ be no more than the maximum vertex weight in $\mathcal{C}$. The solution of the resulting bipartite graph partitioning problem produces a balanced row decomposition of A and a balanced column decomposition of M^T. The weighted cut edges reflect the total communication volume required by the two matrix-vector products. Note that partitioning the two matrices independently would require that the vector being multiplied would need to be reordered in between the multiplications, resulting in additional communication.

5. Algorithms for bipartite graph partitioning.

Now that the rectangular and structurally unsymmetric matrix partitioning problems have been modeled using a bipartite graph, we need algorithms for partitioning such graphs. In this section we propose several algorithms that are adapted from techniques for the standard graph model and one that is specific to bipartite graphs. Each method partitions the bipartite graph into two sets ($\mathcal{P}_1 = \mathcal{R}_1 \cup \mathcal{C}_1$ and $\mathcal{P}_2 = \mathcal{R}_2 \cup \mathcal{C}_2$). Any power-of-two number of sets can be generated by dividing the two sets recursively. And further, any number of sets can be produced this way by a simple generalization of the partitioning problem to generate sets of a specified size ratio.

5.1. Alternating partitioning. The alternating partitioning method, introduced by Kolda [33], is specific to bipartite graphs. Given a column partition, the algorithm produces the best possible row partition. It then takes this new row partition and generates the best possible column partition. The algorithm alternates back and forth between rows and columns until no further improvement is observed. The initial partition can be random, or it can be the output of some other algorithm.

Given a partitioning of the column vertices, the optimal row vertex partition can be computed in the following manner. Let s_i^1 denote the total edge weight between row vertex i and adjacent column vertices in partition 1; similarly, let s_i^2 denote the total edge weight between row vertex i and adjacent column vertices in partition 2. Then $s_i \equiv s_i^1 - s_i^2$ is the *gain* associated with assigning node i to partition 1. (Conversely, $-s_i = s_i^2 - s_i^1$ is the gain associated with assigning node i to partition 2.) Our goal is to assign the vertices to sets in such a way that the total gain of vertices assigned to partition 1 is maximized. In the unconstrained or constrained with unit weights cases, this can be done optimally as shown in the following two theorems.

THEOREM 5.1. *Suppose that the column partition ($\mathcal{C}_1, \mathcal{C}_2$) is fixed and that there is no constraint on the row partition. Let the s_i's (as described above) be sorted so that*

$$s_{i_1} \geq s_{i_2} \geq \cdots \geq s_{i_m}.$$

Select j^* so that for all $j \leq j^*$, s_{i_j} is positive, and for all $j > j^*$, s_{i_j} is nonpositive. Then an optimal assignment of the row vertices is $\mathcal{R}_1 = \{r_{i_1}, \ldots, r_{i_{j^*}}\}$ and $\mathcal{R}_2 = \{r_{i_{j^*+1}}, \ldots, r_m\}$.

This result follows from the observation that each row is placed in its optimal partition. Note that the optimal solution is unique unless one or more s_{i_j} values is zero.

When the total row vertex weight in each partition is constrained, we can generalize the algorithm in a natural way. Choose a dividing point $\hat{j}$ as close as possible to j^* that satisfies the bounds on the total vertex weight. If the row vertices are unit weighted (or equally weighted), then this approach is optimal, as shown by the following theorem.

THEOREM 5.2. *Suppose that the column partition is fixed. Let the s_i's be sorted so that*

$$s_{i_1} \geq s_{i_2} \geq \cdots \geq s_{i_m}.$$

Select j^* so that for all $j \leq j^*$, s_{i_j} is positive, and for all $j > j^*$, s_{i_j} is nonpositive. Let $\hat{j}$ be the closest index to j^* that satisfies the balance constraint. If the row vertices have equal weights, then an optimal assignment of the row vertices is $\mathcal{R}_1 = \{r_{i_1}, \ldots, r_{i_{\hat{j}}}\}$ and $\mathcal{R}_2 = \{r_{i_{\hat{j}+1}}, \ldots, r_{i_m}\}$.

Proof. By Theorem 5.1, j^* is an optimal assignment if there are no balance constraints. The choice of $\hat{j}$ ensures that a minimal number of vertices are placed in a set for which their gain is negative. Further, the vertices with the smallest negative gains are chosen. $\square$

The general weighted and constrained case is more complicated. Some of the vertices may need to be moved out of their preferred set. The question is how to move the minimum amount of gain value while aggregating sufficient weight. This is equivalent to the *knapsack problem* which is known to be NP-hard [18].

Let $|\mathcal{E}|$ denote the number of edges in $\mathcal{G}$ or correspondingly the number of nonzeros in A, and let $|\mathcal{R}|$ and $|\mathcal{C}|$ denote the number of row and column vertices. An *iteration* consists of finding a row partition given a fixed column partition and then finding a column partition given a fixed row partition. It is not hard to show that the complexity of each iteration is $O(|\mathcal{E}| + |\mathcal{R}| + |\mathcal{C}|)$. The computational steps in an iteration are the generation of gain values for each vertex and the determination of j^* (or $\hat{j}$) via a weighted median procedure. Computing the gains for all vertices requires an addition or subtraction for each edge, at a cost of $O(|\mathcal{E}|)$. Finding the weighted median of a set of k values requires $O(k)$ operations (see, for instance, problem 10.2 in [9]), and it is used on a set of $|\mathcal{R}|$ gains and then a set of $|\mathcal{C}|$ gains. Our implementation actually uses a simpler, binary search algorithm for median finding. Although it works well in practice, it is not guaranteed to run in linear time.

The number of iterations is variable but guaranteed finite [33]. Alternatively, a maximum allowable number of iterations can be specified.

This method was derived from the semidiscrete decomposition that was introduced by O'Leary and Peleg [37] for image compression and that was also used for latent semantic indexing in information retrieval by Kolda and O'Leary [32, 35, 34].

5.2. Kernighan–Lin/Fiduccia–Mattheyses. The Kernighan–Lin [31] algorithm is a widely used method for improving a graph partition. As with alternating partitioning, the initial partition can be random, or it can be the output of another algorithm. A reformulation by Fiduccia and Mattheyses [12] improved the performance of the basic approach.

The Fiduccia–Mattheyses (FM) algorithm consists of a sequence of passes over the graph in which vertices are moved from one partition to the other. Move selection is based on the gain concept described in section 5.1, but gains are computed relative to the partition the vertex is currently in. The vertex with the largest gain value is the one whose move will maximally reduce the number of edges cut. Moves that worsen the quality of the partition are allowed, which enables the algorithm to escape local minima. Moves are permitted only if they do not violate the balance constraints *or* if the set the vertex is leaving is larger than its goal weight. Within a pass, vertices are allowed to move only once to avoid infinite looping. The basic structure of a pass is as follows.

(1) Mark all vertices as *eligible*.
(2) For each vertex, compute the *gain* associated with moving it from its current partition to the other; the gain may be negative.
(3) Among moves that improve the balance criteria or that at least do not violate the balance constraints, select the eligible node with the greatest gain. If there are no further eligible nodes, exit.
(4) Move the selected node to the other partition, mark it as *ineligible*, and update the gains of all of its neighbors.
(5) If this is the best partition yet seen that obeys all constraints, save it.
(6) Go to step 3.

Fiduccia and Mattheyses observed that careful use of data structures allows a single pass to be performed in linear time. A priority queue can be used to keep track of the gain values for each type of move (i.e., from set 1 to set 2 or from set 2 to set 1). A bucket sort can be used to organize the initial gains and to efficiently update the gain values. In this way, a pass through the outer loop can be implemented to run in time $O(|\mathcal{E}| + |\mathcal{R}| + |\mathcal{C}|)$. See Fiduccia and Mattheyses [12] for a detailed discussion of data structures.

We have adapted this basic algorithm to address the bipartite graph partitioning problem. The key change is that there are now four types of moves: rows or columns can move from either the first or second set. We maintain a priority queue for each of these move types. To select a vertex to move, we examine the first item in each of the four queues and choose the move with the highest gain that obeys the balance considerations in step 3. In this way, we ensure that the runtime is linear in the size of the graph.

In practice the performance can be improved by stopping the outer loop when a new best partition has not been encountered in a while—say within the past 50 moves, for instance. Another optimization (not in our current implementation) is to evaluate the gain values *lazily*. In the standard FM algorithm, the gain for *every* vertex is calculated before each pass. The gains are updated as the sequence of moves changes them. In the *lazy* implementation, only the gain values of vertices with neighbors in the other partition are computed before each pass. If a vertex moves to the boundary (i.e., one of its neighbors moves to the other set), then its gain is calculated and kept updated from then on. If we have a reasonably good starting partition, then the number of vertices on the partition boundary should be small, and most gains will never need to be calculated. For multilevel algorithms (like the approach described in section 5.4), FM is used to improve partitions that are already fairly good. In this setting, lazy evaluation can significantly reduce execution times [26].

5.3. Spectral. A popular algorithm for standard graph partitioning is *spectral bisection*, which uses an eigenvector of the *Laplacian matrix* associated with the graph [25, 39, 41]. We can apply spectral partitioning to a rectangular or structurally unsymmetric problem by first *symmetrizing* it. Given a bipartite graph $\mathcal{G} = (\mathcal{R}, \mathcal{C}, \mathcal{E})$ of a matrix A, form the corresponding structure matrix $\bar{A} = [\bar{a}_{ij}]$ ($\bar{a}_{ij}$ is nonzero if $(r_i, c_j) \in \mathcal{E}$ and its value is equal to the weight of the edge), and then form the symmetric $(m+n) \times (m+n)$ matrix

$$\tilde{A} = \begin{bmatrix} 0 & \bar{A} \\ \bar{A}^T & 0 \end{bmatrix}.$$

The symmetric $\tilde{A}$ has a well-defined Laplacian matrix that can be used for partitioning. The symmetric partitioning of $\tilde{A}$ can then be used to generate both row and column partitions of A. This approach was used by Berry, Hendrickson, and Raghavan [5].

In order to apply spectral partitioning, the Laplacian of $\tilde{A}$,

$$L = D - \tilde{A}$$

is computed, where $D = \operatorname{diag}\{d_1, d_2, \ldots, d_{m+n}\}$ and $d_i = \sum_j \tilde{a}_{ij}$. The matrix L is symmetric and positive semidefinite. Furthermore, we have the following theorem.

THEOREM 5.3 (Fiedler [13]). *If the graph of $\tilde{A}$ is connected, then the multiplicity of the zero eigenvalue is one.*

Let w denote a Fiedler vector of L, that is, an eigenvector corresponding to the smallest positive eigenvalue of L. Let u denote the first m and v the last n elements of w. Note that u corresponds to rows of A and v to columns. Now sort the elements of u and v so that

$$u_{i_1} \geq u_{i_2} \geq \cdots \geq u_{i_m}$$

and

$$v_{j_1} \geq v_{j_2} \geq \cdots \geq v_{j_n}.$$

This ordering of the elements of u can be used to partition the rows of A. Simply split this sorted list into high-valued and low-valued entries to satisfy the balance criteria. The same algorithm applied to v partitions the columns of A.

For the standard graph partitioning problem, spectral bisection generally produces good partitions, but the eigenvector calculation is expensive.

5.4. Multilevel. The most popular methods for standard graph partitioning use a multilevel approach [6, 26, 29, 30]. A multilevel method starts with a graph that has a large number of vertices, successively merges vertices until it has a coarse graph with a small number of vertices (phase 1), partitions the coarse graph (phase 2), and successively uncoarsens the graph, periodically refining the partition step (phase 3). We have adapted this general framework to the bipartite graph partitioning problem.

5.4.1. Phase 1: Graph coarsening. Let $\mathcal{G} = (\mathcal{R}, \mathcal{C}, \mathcal{E})$ be the current graph. We want to form a smaller graph $\hat{\mathcal{G}} = (\hat{\mathcal{R}}, \hat{\mathcal{C}}, \hat{\mathcal{E}})$ by merging pairs of vertices of $\mathcal{G}$. Row vertices merge only with row vertices, likewise for column vertices. The following procedure determines which row vertices to pair and eventually merge.

(1) Mark all row vertices as *eligible*.
(2) Choose an arbitrary eligible row vertex, say r_i. If no more row vertices are eligible, the pairing is complete.
(3) Find an eligible row vertex r_j with the property that some column vertex is adjacent to both r_i and r_j. If no such row vertex exists, mark r_i as *ineligible* and return to step 2.
(4) Slate vertices r_i and r_j to be merged, and mark both as *ineligible*. Return to step 2.

An analogous procedure is used to determine the column pairing.

Given a set of vertices $\mathcal{V}$ and edges $\mathcal{E}$, a *matching* is a subset of edges $\tilde{\mathcal{E}} \subset \mathcal{E}$ such that no vertex is adjacent to more than one edge in $\tilde{\mathcal{E}}$. A matching $\tilde{\mathcal{E}}$ is *maximal* if no more edges can be added to $\tilde{\mathcal{E}}$ without destroying the matching property. (Note that this is a weaker condition than *maximum* matching which refers to the largest possible set of matching edges in the graph.)

THEOREM 5.4. *If A is the matrix associated with $\mathcal{G}$, then the row pairing algorithm identifies a* maximal matching *among edges of the (symmetric) graph of AA^T. (Similarly, the column pairing constructs a maximal matching among edges of the graph of A^TA.)*

Proof. Recall that a_{ij} is nonzero if and only if $(r_i, c_j) \in \mathcal{E}$. Element (i, j) of AA^T is nonzero if and only if vertices r_i and r_j have a column neighbor in common. Thus, the above process serves as a greedy algorithm for growing a matching in the graph of AA^T. A greedy algorithm generates a maximal matching since, by construction, any unmatched row has no other rows it can pair with. $\square$

THEOREM 5.5. *Let H be the matrix with unit values that has a nonzero structure corresponding to $\mathcal{G}$. The cost of the row-pairing algorithm is $O(|H^Te|_2^2 + |\mathcal{R}|)$, where e is the vector of all ones. (Similarly, the cost of the column-pairing algorithm is $O(|He|_2^2 + |\mathcal{C}|)$.)*

Proof. All the work in the algorithm costs $O(|\mathcal{R}|)$ except for the search for the paired row r_j in step 3. This step can involve examining all paths of length 2 in the bipartite graph. As argued in the proof of Theorem 5.4, each such path will contribute a unit value into HH^T. The number of such paths will thus be the total value of all the entries in HH^T; that is, $e^THH^Te = |H^Te|_2^2$. $\square$

Once all the pairings have been determined, the pairs are merged together. Suppose $\hat{r}_k$ is the result of merging r_i and r_j, then the weight of $\hat{r}_k$ is the sum of the weights of r_i and r_j. There is an edge between $\hat{r}_k$ and $\hat{c}_l$ if any of their constituent vertices were adjacent in G and the weight of the edge is the sum of all the weights of the edges between their constituent vertices. This is analogous to adding the corresponding row and column pairs in A.

The coarse graph maintains the bipartite structure of the original graph and has about half as many vertices. To further coarsen, the process is repeated until the graph has only a small number of vertices, say 100. If at any point too few rows and columns are paired, the coarsening procedure terminates.

5.4.2. Phase 2: Partitioning the coarse graph. Once a small enough bipartite graph has been generated, it is partitioned. Any method can be used; and if the graph is small, the quality of the final answer does not seem sensitive to this choice. In our implementation, we have chosen to use a random partition.

5.4.3. Phase 3: Uncoarsening and refinement. In phase 3, the mergings from phase 1 are successively "undone." If coarse vertex $\hat{r}_k$ is in partition 1, then its two constituent vertices, r_i and r_j, are in partition 1. Before the next "undo" step, a refinement can be performed. In the course of the refinement, for example, r_i may move from partition 1 to partition 2. The "undo" steps continue until the original graph is obtained.

For refinement, we have experimented with three different options: alternating partitioning from section 5.1, FM from section 5.2, and a combination of alternating partitioning followed by FM.

6. Experimental results.

The software is a modification of the Chaco package (written in C) developed by Hendrickson and Leland [24] for partitioning structurally symmetric matrices. All calculations were performed on a 300 MHz Pentium II with 128 MB memory unless otherwise noted.

Table 6.1 lists the methods that are tested. The partitioning is done recursively; that is, first the vertices (rows and columns) are partitioned into two sets, then each of those sets are partitioned into two sets, and so on until we reach the desired number of partitions. If we perform, for example, a row-based partition, each time we split a set into two partitions we require that the difference in the total row vertex weight in each partition be less than or equal to the maximum weight of any single vertex in the set.

The natural partitioning is a simple partition based upon the ordering the matrix had when it was given to us; often those orderings are meaningful. In the row-based case with no constraints on the columns, for example, the ordering of the rows and columns are fixed, but we still need to construct a row partition that obeys the balance constraints and a column partition that minimizes communication. We do this in two steps. First we find the split in the ordering of rows that divides the work into equally sized pieces. Next, with the row partitioning in hand, we find the split in the column ordering that minimizes the communication. This approach is applied recursively; that is, first the nodes are partitioned into two sets, then each of those sets are partitioned, and so on.

The FM and alternating partitioning (AP) methods require some initial partition. Some experimentation convinced us that the methods work best when FM is initialized with a natural partition and AP with a random partition, so all further runs were performed in this way. The spectral method uses the multilevel Rayleigh quotient iteration/Symmlq eigensolver [1] from the Chaco partitioning software [24]. The multilevel (ML) algorithms divide the coarsest graph randomly and use various

TABLE 6.1
Partitioning methods.

Abbreviation	Method
Natural	Natural ordering
FM	Fiduccia–Mattheyses
AP	Alternating partitioning
Spectral	Spectral method
ML–FM	Multilevel Fiduccia–Mattheyses
ML–AP	Multilevel alternating partitioning
ML–AP+FM	Multilevel alternating partitioning plus Fiduccia–Mattheyses

TABLE 6.2
Test matrices.

Matrix	Application	Rows	Columns	NNZ	Density	Dense?
`pig-large`	Least squares	28254	17264	75018	1.5e-4	—
`pig-very`	Least squares	174193	105882	463303	3.7e-4	—
`dfl001`	Linear program	6071	12230	35632	4.8e-4	1 row
`Amatrix`	Linear program	123221	141344	1437692	8.3e-5	72 rows
`we1998`	Truncated SVD	719736	96300	27546437	4.0e-4	1672 cols
`memplus`	Preconditioned	17758	17758	99147	3.1e-4	—
`precond`	linear system			76372	2.4e-4	—

refinement strategies: FM, AP, and AP followed by FM (AP+FM). We handle disconnected graphs specially in all cases except the natural partitioning and FM (because we use the natural ordering to generate the initial partition in this case) by identifying all the connected components, assigning components to partitions in a greedy fashion, and only partitioning what remains. In each case, we repeatedly coarsen until the number of coarse row and column vertices is less than 100. Refinements were performed at *every other* iteration of the uncoarsening phase.

The test matrices were gathered from the various applications discussed in section 1 (see Table 6.2). The two items in the last row of the table refer to a matrix and its preconditioner, as is discussed in section 6.4. Dense rows and columns are noted because that will affect whether the partitioning is row- or column-based. We consider a row or column to be dense if more than 1/32 of its values are nonzero.

For each test matrix, we show two tables. The first table details the communication pattern. The *Edge cuts* column lists the number of nonzeros outside the block diagonal, that is, the edges in the bipartite graph that are *cut* by the given vertex partition. The *Part time* column lists the time (in seconds) to compute the partition. The *Total msgs* and *Total vol* columns list, respectively, the total number of messages and total message volume for computing either Ax or A^Tv. (Recall from Facts 4 and 6 that those values are equal for Ax and A^Tv.) The *Max msg* and *Max vol* columns list, respectively, the maximum number and maximum volume of messages handled by a *single* processor in the computation of Ax or A^Tv, incoming or outgoing.

The second table for each matrix lists the block partition information. We have partitioned these matrices to balance the number of multiplies per processor, that is, the number of nonzero matrix elements per processor. Each processor holds one block row or one block column. Columns 2–5 list the details for the *Block rows*. The *Min rows* and *Max rows* list, respectively, the minimum and maximum number of rows in any block row. The *Min NZ* and *Max NZ* columns list, respectively, the minimum and maximum number of nonzeros in any block row. Although the numbers of rows owned by processors may vary significantly, variation in the number of nonzeros should be

TABLE 6.3
Communication pattern for row-based partitioning of the `pig-large` *matrix on eight processors.*

Method	Edge cuts	Part time	Total msgs	Total vol	Max msgs	Max vol
Natural	49048	0.21	32	21172	7	5534
FM	15659	1.40	56	11309	7	2618
AP	20251	2.16	56	11714	7	1985
ML-FM	8013	3.14	55	2454	7	607
ML-AP	10299	2.74	56	4830	7	1138
ML-AP+FM	7671	5.25	56	2292	7	443
Spectral	5693	167.93	53	2721	7	829

TABLE 6.4
Block information for the row-based partitioning of the `pig-large` *matrix on eight processors.*

Method	Block row				Block column			
	Min rows	Max rows	Min NZ	Max NZ	Min cols	Max cols	Min NZ	Max NZ
Natural	3124	6375	9372	9381	0	6203	0	39439
FM	3342	3786	9376	9379	1318	2955	6851	11483
AP	3310	3740	9376	9379	2154	2162	7748	10917
ML-FM	3397	3652	9376	9379	2003	2280	8223	10150
ML-AP	3206	3954	9376	9379	2155	2161	8756	10329
ML-AP+FM	3441	3621	9376	9378	2006	2303	8701	10198
Spectral	3138	3694	9373	9381	2018	2374	8600	10412

small when the partition is row-based since this balances the computational work. The next four columns list analogous values for the *Block columns*.

We choose the number of processors, p, in each case so that the number of nonzeros per processor is 10,000, give or take a factor of 2, except for the `Amatrix` which has 30,000 nonzeros per processor.

6.1. Least squares. The `pig-large` and `pig-very` matrices are from *least squares* problems relating to pig breeding data [21, 28] and were obtained from Duff [10].

The `pig-large` matrix is of size 28,254 × 17,264 with 75,018 nonzeros. The results of row-based partitioning the `pig-large` matrix over eight processors are given in Tables 6.3 and 6.4. The natural partitioning takes a small amount of time to compute (0.21 seconds) because the matrix still must be divided in such a way that each block row has approximately the same number of nonzeros. Notice that the natural partitioning requires the fewest messages (32) but the highest total volume (21,172). Also note that the minimum number of columns in a block is zero, which means that the processors with zero columns have no parts of the vector x in the Ax computation. Those processors will not have any messages to send nor any diagonal component ($A_{ii}x_i$) to compute and will be idle until they receive messages from the other processors.

In contrast, the various partitioning methods increase the total message count to at or near the maximum of 56 but drastically reduce the total message volume (by a factor of more than 9 in the best case) and the maximum volume handled by a single processor (by a factor of more than 12 in the best case). Further, the partitionings yield more balanced column partitions even though no constraint was used. Of course, the number of nonzeros handled by each processor is about equal as required. In fact, the number of nonzeros handled by a single processor varies by far less than 1% as we

TABLE 6.5
Communication pattern for row-based partitioning of the `pig-very` *matrix on* 32 *processors.*

Method	Edge cuts	Part time	Total msgs	Total vol	Max msgs	Max vol
Natural	292055	2.45	290	109180	30	16772
FM	99252	26.53	837	75270	31	4808
AP	142967	23.99	931	88610	31	5227
ML-FM	38552	40.41	851	12973	31	1332
ML-AP	54261	37.84	926	24737	31	1929
ML-AP+FM	38096	60.87	831	13019	31	1240

TABLE 6.6
Block information for the row-based partitioning of the `pig-very` *matrix on* 32 *processors.*

Method	Block row				Block column			
	Min rows	Max rows	Min NZ	Max NZ	Min cols	Max cols	Min NZ	Max NZ
Natural	4825	14481	14475	14481	0	16720	0	129623
FM	4976	6004	14476	14480	1213	4814	9844	21187
AP	4857	6621	14477	14480	2673	3841	9942	20778
ML-FM	5234	5802	14475	14483	2964	3544	13851	16231
ML-AP	4892	6637	14477	14480	3210	3509	12821	16529
ML-AP+FM	5263	5646	14476	14480	2960	3529	13191	16150

can see by looking at the minimum and maximum number of nonzeros in each block row.

The ML-AP+FM method yielded the best partitioning and required about 5 seconds of processing time, on par with the other methods. In general, the ML methods yielded the best total volume and maximum single processor volume. The FM method was the fastest partitioning method but did not reduce the message volume as much as the other methods. The spectral method was slower than all others by a factor of more than 30 but did not produce the best partition.

Recall that our methods attempt to find partitionings that minimize the number of edge cuts. This does not correspond exactly to total message volume but is merely an approximation. On this problem, notice that the reduction in edge cuts correspond roughly to the reduction in total message volume. For example, the ML-AP+FM method has the fewest edge cuts as well as the least communication volume.

The `pig-very` matrix is of size 174,193 × 105,882 with 463,303 nonzeros. Tables 6.5 and 6.6 show the results of partitioning this matrix row-wise over 32 processors. In this case we do not show results for the spectral method because it was too time consuming.

The results are very similar to the results obtained for the `pig-large` matrix. There is a correspondence (albeit an imperfect one) between edge cuts and total message volume. The ML methods yield the best partitions, in the best case reducing the total message volume by a factor of 8. The maximum message volume handled by a single processor is decreased by a factor of more than 13 in the best case at the cost of about three times more messages.

The natural partitioning seems promising in terms of message count, but the maximum message volume handled by a *single* processor is more than that handled by *all* 32 processors for the ML partitionings.

6.2. Linear programming. The 6,071 × 12,230 `dfl001` matrix is a linear programming constraint matrix with 35,632 nonzeros. This matrix was obtained from

TABLE 6.7
Communication pattern for column-based partitioning of the `dfl001` *matrix on eight processors.*

Method	Edge cuts	Part time	Total msgs	Total vol	Max msgs	Max vol
Natural	30989	0.08	44	19804	7	8751
FM	8132	1.80	56	7493	7	1247
AP	12171	0.86	56	11552	7	1967
ML-FM	6553	2.02	56	5875	7	1022
ML-AP	7860	1.49	56	7040	7	1145
ML-AP+FM	6651	2.68	56	5959	7	994
Spectral	14734	40.61	55	10633	7	2993

TABLE 6.8
Block information for the column-based partitioning of the `dfl001` *matrix on eight processors.*

Method	Block row				Block column			
	Min rows	Max rows	Min NZ	Max NZ	Min cols	Max cols	Min NZ	Max NZ
Natural	0	3088	0	18545	1375	1602	4449	4457
FM	588	823	4173	4611	1289	1707	4448	4462
AP	659	858	3743	5512	1017	1977	4449	4464
ML-FM	696	856	4042	4749	1297	1822	4448	4460
ML-AP	755	763	4001	4883	1324	1749	4453	4455
ML-AP+FM	717	812	4194	4729	1323	1763	4444	4460
Spectral	426	1215	1859	7299	932	1937	4448	4458

Netlib.[3] The matrix contains one dense row and so was partitioned column-wise.

Tables 6.7 and 6.8 show the results of partitioning the `dfl001` matrix over eight processors. The original matrix does not have much structure, and the only reason the total number of messages for the natural partition is only 44 (versus 56) is that some partitions contain no rows. In the best case we can reduce the total message volume by a factor of more than 3 and the maximum message volume on a single processor by a factor of more than 8. The block columns are very balanced in terms of the number of nonzeros per block. The block rows are reasonably balanced for the FM and ML methods although this was not enforced by any constraint. Again we can observe that edge cuts correspond to total message volume.

The 123,221 $\times$ 141,344 `Amatrix` was obtained from Rothberg [40]. This matrix has 1,437,692 nonzeros and contains 72 dense rows. Tables 6.9 and 6.10 contain the results of a column-based partitioning of this matrix over 128 processors. This is an interesting partitioning problem because even though all of the partitionings reduce the edge cuts by at least 25%, the total message volume is not reduced much and in some cases (AP, ML-AP, ML-AP+FM) even increases. Thus, for this problem, the assumption that edge cuts correlate with communication volume is invalid. Despite this, the partitioning is still beneficial because it reduces the total message volume handled by a single processor by a factor of 5 in the best case and even decreases the maximum number of messages that any processor handles. Further, the FM and AP methods do better than the ML methods in that they have a smaller total number of messages, approximately the same total message volume, a smaller number of maximum messages per processor, and approximately the same maximum volume per processor. Further, computing the partitionings for the FM and AP methods is faster than for the ML methods.

[3]http://www.netlib.org/lp/

Table 6.9
Communication pattern for column-based partitioning of the `Amatrix` *matrix on* 128 *processors.*

Method	Edge cuts	Part time	Total msgs	Total vol	Max msgs	Max vol
Natural	1414303	4.60	766	358721	126	94346
FM	967664	54.42	4461	325724	94	18718
AP	1006357	40.58	5427	372168	83	19780
ML-FM	993194	74.18	7770	330508	119	21551
ML-AP	975488	78.20	6088	376243	107	19575
ML-AP+FM	965200	115.16	5877	397407	119	18179

Table 6.10
Block information for the column-based partitioning of the `Amatrix` *matrix on* 128 *processors.*

Method	Block row				Block column			
	Min rows	Max rows	Min NZ	Max NZ	Min cols	Max cols	Min NZ	Max NZ
Natural	0	68010	0	826156	361	3896	11196	11265
FM	1	7377	248	157166	381	4111	11202	11262
AP	126	1997	999	158408	383	4548	11179	11274
ML-FM	35	1927	940	174351	382	2409	11199	11258
ML-AP	98	2161	1062	92798	373	3898	11195	11278
ML-AP+FM	40	4011	1032	112416	376	4010	11195	11256

6.3. Truncated SVD. The 719,736 $\times$ 96,300 `we1998` matrix with 27,546,437 nonzeros is used in a geophysical application where a truncated SVD must be computed (see Vasco, Johnson, and Marques [43]); the matrix was provided by Vasco and Marques [44]. This matrix has 1672 dense columns and so was partitioned row-wise. Because of the size of the matrix, the problem was run on an SGI Onyx with two processors and six gigabytes of memory, so the timings cannot be compared with the timings of the other problems.

In Tables 6.11 and 6.12, we show the result of partitioning `we1998` over 1024 processors. The situation is almost the opposite of that for `Amatrix`. The number of edge cuts is only modestly reduced, but the total message volume is halved by every partitioning method. The total number of messages goes up by a factor of about 125, depending on the method, but the maximum number of messages handled by a single processor is actually reduced by about 20%, and the maximum volume handled by a single processor is reduced by a factor of about 400.

The block rows are very evenly divided with each containing about 27,000 nonzeros. The block columns, on the other hand, are not so even, with some blocks being assigned no columns. However, in the natural partitioning one partition has 40% of the columns and 7% of the nonzeros. Since the partition is row-wise, this has no impact on load balance but leads to the very large value for maximum communication volume in Table 6.11. With the other decompositions, no partition has more than 0.7% of the columns and 0.6% of the nonzeros.

6.4. Preconditioned linear systems. Here we give results for working with a preconditioned linear system. As mentioned earlier, the goal is to partition a matrix A and its approximate inverse preconditioner M so that both PAQ and Q^TMP^T are well partitioned; that is, the work per processor is balanced, and the communication costs are low.

TABLE 6.11
Communication pattern for row-based partitioning of the `we1998` *matrix on* 1024 *processors.*

Method	Edge cuts	Part time	Total msgs	Total vol	Max msgs	Max vol
Natural	25030959	95.61	3065	6481846	1023	5067422
FM	21952461	651.83	401074	3108517	826	12271
AP	22089460	1541.85	375587	3061108	769	10796
ML-FM	21831402	1428.13	354089	2989724	787	15224
ML-AP	21895852	2011.43	352771	2930819	813	13142
ML-AP+FM	21823900	2512.78	351472	2930031	763	11010

TABLE 6.12
Block information for the row-based partitioning of the `we1998` *matrix on* 1024 *processors.*

Method	Block row				Block column			
	Min rows	Max rows	Min NZ	Max NZ	Min cols	Max cols	Min NZ	Max NZ
Natural	277	1377	26700	27126	0	38141	0	20495167
FM	155	2013	26745	27037	4	643	2327	136809
AP	153	2978	26666	27126	0	475	0	175255
ML-FM	154	2162	26782	27014	1	593	88	141041
ML-AP	155	3275	26648	27150	0	577	0	144909
ML-AP+FM	154	3195	26787	27014	1	544	48	132539

The `memplus` matrix is available from MatrixMarket.[4] (It contained 27,003 explicitly stored zeros, which were removed.) The matrix is of size 17,758 with 99,147 nonzeros. We used research code provided by Benzi and Tůma [2] to generate an approximate inverse preconditioner via the method of Grote and Huckle [20]. The resulting preconditioner had 76,372 nonzeros. The two matrices were combined into a bipartite graph with weighted edges and vertices as described in section 4. The `memplus` matrix will be partitioned row-wise and the *transpose* of the preconditioner will be partitioned column-wise.

The results of the various partitioning strategies for `memplus` and its preconditioner are given in Tables 6.13 and 6.14. There are two rows for each partitioning strategy: the first corresponds to `memplus` and the second to the *transpose* of the preconditioner. Using ML-FM, the total message volume is reduced by nearly a factor of 6 for the matrix and by over 16 for the preconditioner, although the number of messages does increase in each case. Further, the maximum message volume on a single processor is reduced by a factor of nearly 5 and more than 8, respectively. The FM, ML-AP, and ML-AP+FM methods behaved similarly. The AP method was not quite as good as the previously mentioned four methods. The spectral method was nearly as bad as no partitioning at all.

The number of nonzeros per block row is required to be nearly equal for `memplus` and likewise for the block columns of the transposed preconditioner.

We have also added a row for the *symmetric* ML-FM scheme in Chaco [24]. The scheme partitions the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ defined by $\mathcal{V} = \{1, 2, \ldots, n\}$, where n is the order of the matrix, and $(i, j) \in \mathcal{E}$ if either a_{ij}, a_{ji}, m_{ij}, or m_{ji} is nonzero with an edge weight equal to the number of those entries that are nonzero. The weight of vertex i is equal to the number of nonzeros in row i of A plus the number of nonzeros in column i of M. The resulting symmetric matrix was converted into a weighted

[4]http://math.nist.gov/MatrixMarket/

Table 6.13
Communication pattern for `memplus` *and its (transposed) preconditioner on eight processors.*

Method	Edge cuts	Part time	Total msgs	Total vol	Max msgs	Max vol
Natural	84044	0.21	38	37468	7	6655
	68495		51	42545	7	7545
FM	26276	1.99	55	9793	7	1822
	7334		48	4232	7	1350
AP	38462	2.53	46	19695	7	5336
	10933		39	7668	7	3187
ML-FM	16076	4.34	56	6333	7	1339
	4996		55	2595	7	886
ML-AP	16416	5.11	56	7155	7	1659
	4145		51	2243	7	1147
ML-AP+FM	16609	6.85	56	7200	7	1903
	3024		51	2077	7	1177
Spectral	55722	78.60	52	30113	7	5218
	43823		48	32065	7	6672
Sym ML-FM	16515	1.71	56	6056	7	2877
	853		37	583	7	161

Table 6.14
Block information for `memplus` *and its (transposed) preconditioner on eight processors.*

Method	Block row				Block column			
	Min rows	Max rows	Min NZ	Max NZ	Min cols	Max cols	Min NZ	Max NZ
Natural	204	3664	12279	12503	2009	2429	7242	36717
			1919	16158			9541	9551
FM	1694	2881	12144	12555	2097	2570	10355	13921
			8129	10467			9544	9549
AP	279	2944	12347	12587	2021	2408	10394	17749
			5637	10993			9541	9551
ML-FM	1961	2451	12196	12627	2135	2387	11464	14458
			8387	10664			9543	9549
ML-AP	1767	2563	12205	12658	2147	2484	11327	14373
			7844	10197			9543	9549
ML-AP+FM	1629	2600	12173	12621	2149	2444	10997	15092
			8121	10286			9543	9548
Spectral	264	3928	12269	12519	1909	3180	8474	21221
			3236	14676			9541	9551
Sym ML-FM	1749	2416	11410	14002	1749	2416	11439	13988
			8109	10342			7891	10455

graph and partitioned by the ML partitioning routine in Chaco.

Because this process couples the structure of A and M, the partitioner is unable to balance them independently. Consequently, neither A nor M are well load balanced, as evidenced by the min and max nonzero values for rows of A and columns of M. Although the total work for performing both products is well balanced, this may be insufficient because a synchronization may be necessary in between the two products.

However, by weakening the load balance constraint in this manner, a much better partition is now found, leading to a significant reduction in communication cost. It is also worth noting that the run time of Chaco is decidedly less than that for the bipartite partitioning algorithms. There are two reasons for this: First, the bipartite graph has twice as many vertices, so the partitioning problem is larger. Second, some of the performance enhancing features in Chaco (principally lazy evaluation) are not currently in the bipartite partitioning code.

7. Conclusions. There are numerous algorithms requiring repeated parallel matrix-vector and matrix-transpose-vector multiplies with rectangular or structurally unsymmetric sparse matrices. We outlined parallel matrix-vector multiply routines and demonstrated that their performance depends on the *partitioning* of the matrix. We showed that partitioning a rectangular or structurally unsymmetric matrix corresponds to partitioning a bipartite graph. We also showed that the bipartite partitioning model can allow for simultaneous partitioning of a matrix and its explicit preconditioner. We then presented several methods for the bipartite graph partitioning problem: AP, Kernighan–Lin/FM, spectral, and multilevel.

We gave results for partitioning several large matrices arising from various applications. Overall, we found that the multilevel methods usually work best. The best refinements seem to be either FM or AP plus FM. The latter is a little more expensive in terms of time. The spectral method was by far the worst and failed to even converge on many problems.

A number of areas for future study exist. It is important to know if the theoretical gains in performance shown by our results hold in practice, so we are currently implementing the parallel matrix-vector multiply on various parallel architectures. The work on simultaneously partitioning a matrix and its explicit preconditioner can be extended further to the case where there is an explicit *factored* preconditioner. We also intend to optimize the research code we have been using for the partitioning by incorporating many of the enhancements available in the best codes for standard graph partitioning (e.g., lazy evaluation). Last, as the results from the `Amatrix` and `we1998` matrices show, edge cuts may only loosely correlate with communication volume. As we mentioned in section 4, this same approximation is made in the symmetric graph partitioning approaches to the parallelization of partial differential equations (PDEs) [22]. We believe the reason this issue has not received much attention is that for PDE matrices the approximation is a good one since the vertices generally have a small, nearly constant number of neighbors. Matrices from other applications, like those we have considered, tend to have more complex structures and often the approximation is not an accurate one. The hypergraph model of Çatalyürek and Aykanat [7] elegantly encodes the correct metric of communication volume. However, since it treats rows and columns differently, it fails to define a partition of both. Also, it does not seem able to capture the application of multiple matrices. For these reasons, a hybrid model that has the advantages of both approaches would be desirable. Towards this end we plan to investigate alternative refinement strategies within our bipartite model that target a more accurate metric for the communication cost.

Acknowledgments. One of the challenges of this problem was obtaining large matrices to work with. We have the following people to thank for providing us with data and/or data conversion utilities as well as helpful advice: Michele Benzi, Mike Berry, Iain Duff, Osni Marques, Karin Remington, Ed Rothberg, Mike Saunders, Matt St. John, Weichung Wang, and Houngyun Zha. We would like to acknowledge that the following conversion utilities were used: Harwell–Boeing I/O Routines in C (`hbio1.0`)[5] and the `mps2mat` routine available in LIPSOL.[6] Also, thanks to Al Geist for helpful discussions on the fine points of load balancing and to Eric de Sturler on preconditioning of iterative methods. We are also indebted to Michele Benzi and Alex

[5]http://math.nist.gov/mcsd/Staff/KRemington/harwell_io/harwell_io.html
[6]http://www.caam.rice.edu/~zhang/lipsol/

Pothen for their useful comments on earlier drafts of this work.

REFERENCES

[1] S. T. Barnard and H. D. Simon, *A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems*, Concurrency: Practice and Experience, 6 (1994), pp. 101–117.

[2] M. Benzi and M. Tůma, *Private communication*, 1998.

[3] M. Benzi and M. Tůma, *A Comparative Study of Sparse Approximate Inverse Preconditioners*, Tech. Rep. LA-UR-98-0024, Los Alamos National Laboratory, Los Alamos, NM, 1998.

[4] M. W. Berry, S. T. Dumais, and G. W. O'Brien, *Using linear algebra for intelligent information retrieval*, SIAM Rev., 37 (1995), pp. 573–595.

[5] M. W. Berry, B. Hendrickson, and P. Raghavan, *Sparse matrix reordering schemes for browsing hypertext*, in The Mathematics of Numerical Analysis, J. Renegar et al., eds., Lectures in Appl. Math. 32, AMS, Providence, RI, 1996, pp. 99–122.

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

[7] U. V. Çatalyürek and C. Aykanat, *Decomposing linear programs for parallel solution*, in Parallel Algorithms for Irregularly Structured Problems, Third Interational Workshop, Irregular '96, Santa Barbara, CA, 1996, A. Ferreira et al., eds., Lecture Notes in Comput. Sci. 1117, Springer-Verlag, New York, 1996, pp. 75–86.

[8] A. B. Coon and M. A. Stadtherr, *Generalized block-tridiagonal matrix orderings for parallel computation in process flowsheeting*, Comput. Chem. Engrg., 19 (1995), pp. 787–805.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[10] I. Duff, *Private communication*, 1998.

[11] M. C. Ferris and J. D. Horn, *Partitioning mathematical programs for parallel solution*, Math. Programming, 80 (1998), pp. 35–62.

[12] C. M. Fiduccia and R. M. Mattheyses, *A linear time heuristic for improving network partitions*, in Proceedings of the 19th Design Automation Conference, Las Vegas, NV, 1982, IEEE, Piscataway, NJ, pp. 175–182.

[13] M. Fiedler, *Algebraic connectivity of graphs*, Czechoslovak Math. J., 23 (1973), pp. 298–305.

[14] R. Fletcher, *Conjugate gradient methods for indefinite systems*, in Numerical Analysis, Dundee, 1975, G. A. Watson, ed., Lecture Notes in Math. 506, Springer-Verlag, Berlin, 1976, pp. 73–89.

[15] R. W. Freund, *A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems*, SIAM J. Sci. Comput., 14 (1993), pp. 470–482.

[16] 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.

[17] M. Garey, D. Johnson, and L. Stockmeyer, *Some simplified NP-complete graph problems*, Theoret. Comput. Sci., 1 (1976), pp. 237–267.

[18] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

[19] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 2nd ed., The Johns Hopkins University Press, Baltimore, 1989.

[20] M. Grote and T. Huckle, *Parallel preconditioning with sparse approximate inverses*, SIAM J. Sci. Comput., 18 (1997), pp. 838–853.

[21] M. Hegland, *Description and Use of Animal Breeding Data for Large Least Squares Problems*, Tech. Rep. TR-PA-93-50, CERFACS, Toulouse, France, 1993. See also [28].

[22] B. Hendrickson, *Graph partitioning and parallel solves: Has the emporer no clothes?*, in Solving Irregularly Structured Problems in Parallel, 5th International Symposium, Irregular '98, Berkeley, CA, 1998, A. Ferreira et al., eds., Lecture Notes in Comput. Sci. 1457, Springer-Verlag, New York, 1998, pp. 218–225.

[23] B. Hendrickson and T. G. Kolda, *Partitioning sparse rectangular matrices for parallel computations of Ax and A^Tv*, in Applied Parallel Computing in Large Scale Scientific and Industrial Problems, 4th International Workshop, PARA'98, Umeå, Sweden, 1998, B. Kågström et al., eds., in Lecture Notes in Comput. Sci. 1541, Springer-Verlag, New York, 1998, pp. 239–247.

[24] B. Hendrickson and R. Leland, *The Chaco User's Guide, Version* 2.0, Tech. Rep. SAND95-2344, Sandia National Laboratories, Albuquerque, NM, 1995.

[25] B. Hendrickson and R. Leland, *An improved spectral graph partitioning algorithm for mapping parallel computations*, SIAM J. Sci. Statist. Comput., 16 (1995), pp. 452–469.

[26] B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, in Proceedings of the 1995 ACM/IEEE Supercomputing Conference, San Diego, CA, ACM, New York, 1995.

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

[28] A. Hofer, *Schätzung von Zuchtwerten feldgeprüfter Schweine mit einem Mehrmerkmals-Tiermodell*, Ph.D. thesis, ETH-Zurich, 1990. Cited in [21].

[29] G. Karypis and V. Kumar, *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*, Tech. Rep. 95-035, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1995.

[30] G. Karypis and V. Kumar, *Parallel Multilevel Graph Partitioning*, Tech. Rep. 95-036, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1995.

[31] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, 49 (1970), pp. 291–308.

[32] T. G. Kolda, *Limited-Memory Matrix Methods with Applications*, Ph.D. thesis, Applied Mathematics Program, University of Maryland, College Park, MD, 1997. Also available as Department of Computer Science Tech. Rep. CS-TR-3806.

[33] T. G. Kolda, *Partitioning sparse rectangular matrices for parallel processing*, in Solving Irregularly Stuctured Problems in Parallel: 5th International Symposium, Irregular '98, Berkeley, CA, 1998, A. Ferreira et al., eds., in Lecture Notes in Comput. Sci. 1457, Springer-Verlag, New York, 1998, pp. 68–79.

[34] T. G. Kolda and D. P. O'Leary, *A semidiscrete matrix decomposition for latent semantic indexing in information retrieval*, ACM Trans. Inf. Syst., 16 (1998), pp. 322–346.

[35] T. G. Kolda and D. P. O'Leary, *Latent semantic indexing via a semi-discrete matrix decomposition*, in The Mathematics of Information Coding, Extraction and Distribution, G. Cybenko et al., eds., IMA Vol. Math. Appl. 107, Springer-Verlag, New York, 1999, pp. 73–80.

[36] J. G. Lewis and R. A. van de Geijn, *Distributed memory matrix–vector multiplication and conjugate gradient algorithms*, in Proceedings of the 1993 ACM/IEEE Supercomputing Conference, Portland, OR, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 484–492.

[37] D. P. O'Leary and S. Peleg, *Digital image compression by outer product expansion*, IEEE Trans. Comm., 31 (1983), pp. 441–444.

[38] C. C. Paige and M. A. Saunders, *LSQR: An algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Software, 8 (1982), pp. 43–71.

[39] A. Pothen, H. D. Simon, and K.-P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 430–452.

[40] E. Rothberg, *Private communication*, 1998.

[41] H. D. Simon, *Partitioning of unstructured problems for parallel processing*, in Computing Systems in Engineering, vol. 2/3, Pergamon Press, Elmsford, NY, 1991, pp. 135–148.

[42] R. S. Tuminaro, J. H. Shadid, and S. A. Hutchinson, *Parallel sparse matrix vector multiply software for matrices with data locality*, Concurrency: Practice and Experience, 10 (1998), pp. 229–247.

[43] D. W. Vasco, L. R. Johnson, and O. Marques, *Global Earth structure: Inference and assessment*, Geophys. J. Internat., 137 (1999), pp. 381–407.

[44] D. W. Vasco and O. Marques, *Private communication*, 1998.

[45] C. Walshaw, M. Cross, and M. Everett, *Mesh partitioning and load-balancing for distributed memory parallel systems*, in Parallel and Distributed Processing for Computational Mechanics: Systems and Tools, B. H. V. Topping, ed., Saxe-Coburg Publications, Edinburgh, 1999, pp. 110–123.

[46] W. Wang and D. P. O'Leary, *Adaptive Use of Iterative Methods in Interior Point Methods for Linear Programming*, Tech. Rep. CS-TR-3560, Department of Computer Science, University of Maryland, College Park, MD, 1995.