Electronic Transactions on Numerical Analysis.
Volume 21, pp. 47-65, 2005.
Copyright © 2005, Kent State University.
ISSN 1068-9613.

ETNA
Kent State University
etna@mcs.kent.edu

COMMUNICATION BALANCING IN PARALLEL SPARSE MATRIX–VECTOR MULTIPLICATION[*]

ROB H. BISSELING[†] AND WOUTER MEESEN[‡]

Dedicated to Alan George on the occasion of his 60th birthday

Abstract. Given a partitioning of a sparse matrix for parallel matrix–vector multiplication, which determines the total communication volume, we try to find a suitable vector partitioning that balances the communication load among the processors. We present a new lower bound for the maximum communication cost per processor, an optimal algorithm that attains this bound for the special case where each matrix column is owned by at most two processors, and a new heuristic algorithm for the general case that often attains the lower bound. This heuristic algorithm tries to avoid raising the current lower bound when assigning vector components to processors. Experimental results show that the new algorithm often improves upon the heuristic algorithm that is currently implemented in the sparse matrix partitioning package Mondriaan. Trying both heuristics combined with a greedy improvement procedure solves the problem optimally in most practical cases. The vector partitioning problem is proven to be NP-complete.

Key words. vector partitioning, matrix–vector multiplication, parallel computing, sparse matrix, bulk synchronous parallel

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

1. Introduction.

Sparse matrices from emerging applications such as information retrieval, linear programming, and Markov modelling of polymers may have a highly irregular structure and no underlying three-dimensional physical structure, in contrast to many finite-element matrices, and this irregularity poses a challenge to parallel computation. Often, the sparse matrix must be repeatedly multiplied by a vector, for instance in iterative linear system solvers and eigensystem solvers. In recent years, hypergraph partitioning has become the tool of choice for partitioning the sparse matrix, delivering good balance of the computation load and a minimal communication volume during parallel sparse matrix–vector multiplication. The problem of partitioning the input and output vectors is just as important as the matrix partitioning problem, since it affects the balance of the communication load, but it has received much less attention.

Assume that we have an $m \times n$ sparse matrix A, which must be multiplied by an input vector $\mathbf{v}$ of length n, to give an output vector $\mathbf{u}$ of length m, using p processors of a parallel computer with distributed memory. The natural parallel algorithm for sparse matrix–vector multiplication with an arbitrary distribution of matrix and vectors consists of the following four phases:

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

[*]Received October 5, 2004. Accepted for publication June 14, 2005. Recommended by B. Hendrickson.

[†]Mathematical Institute, Utrecht University, PO Box 80010, 3508 TA Utrecht, The Netherlands (`Rob.Bisseling@math.uu.nl`). Part of the work was done while this author was a visiting scientist at the Computer Science Research Institute, Sandia National Laboratories, PO Box 5800, Albuquerque, NM 87185-1110, USA. Part of the research has been funded by the Dutch BSIK/BRICKS project. Computer time has been partially funded by the Dutch National Computer Facilities foundation (NCF).

[‡]Mathematical Institute, Utrecht University, PO Box 80010, 3508 TA Utrecht, The Netherlands (`woutermeesen@gmail.com`).

4. Each processor adds the contributions received for its components u_i, giving $u_i = \sum_{t=0}^{p-1} u_{it}$.

Processors are assumed to synchronise globally between the phases. In the language of the bulk synchronous parallel (BSP) model [21], the phases are called *supersteps*. This model motivated the present work, because it encourages balancing communication loads, besides balancing computation loads. Phases 1 and 3 are communication supersteps. Their costs include a fixed overhead l representing the global *latency*, which lumps together the time of the global synchronisation and the startup times of an all-to-all message exchange. The superstep approach allows combining data words destined for the same processor into one packet and also reordering of packets, both for the purpose of communication optimisation. Sometimes, we call l just the *synchronisation cost*. The BSP model also assumes a cost of g time units per data word sent or received by a processor. The processor with the maximum number h of data sent or received determines the overall communication cost. The cost of a communication superstep can thus be expressed as

$$T_{\text{superstep}} = hg + l. \tag{1.1}$$

A natural time unit for g and l in scientific computation is the time of a floating-point operation.

The synchronisation cost l does not grow with the problem size and it has to be paid only twice for the matrix–vector multiplication algorithm. The communication cost, however, grows with the problem size, and depends very much on the data partitioning chosen. We are mainly interested in large, highly irregular problems, and for this reason we will exclusively be concerned with the communication cost hg, aiming to minimise it.

We assume that the matrix A has already been partitioned for p processors and that we have to find a suitable vector partitioning. Thus, each nonzero a_{ij} has been assigned to a processor and we need to assign vector components v_j and u_i to processors. We assume that the vectors can be partitioned independently, which is usually the case for rectangular matrices, but also for square matrices if the output vector is used as the input for multiplication by A^T. We only treat the partitioning problem for the input vector, because the partitioning of the output vector is similar and can be done by running the same algorithm applied to A^T. If the matrix A is symmetric and it has been partitioned symmetrically, i.e., with nonzero elements a_{ij} and a_{ji} assigned to the same processor, then we can partition the input vector and use the same solution for the output vector as well.

The preceding matrix partitioning can be done by using any of the currently available matrix partitioners, either based on hypergraph partitioning or graph partitioning. The result can be: one-dimensional [5], e.g. a row distribution or column distribution; two-dimensional Cartesian [8], each processor obtaining a submatrix defined by a partitioning of the matrix rows and columns; two-dimensional non-Cartesian with Mondriaan structure [23], defined by recursively bipartitioning the matrix in either the row or column direction; or completely arbitrary [7], each nonzero having been assigned individually to a processor. In all these cases, it may be beneficial to partition the vector by the methods presented here.

Let $\mathcal{P}_j$ be the set of processors that own nonzeros in matrix column j and $\lambda_j = |\mathcal{P}_j|$ be the number of such processors, for $0 \leq j < n$. We make a number of assumptions, all without loss of generality, to facilitate the exposition. We assume that $\lambda_j \geq 2$, because columns with $\lambda_j = 0, 1$ do not cause communication so that they can be removed from the problem. We assume that all processors are involved in communication, i.e., occur in at least one matrix column; it is easy to remove the other processors. Furthermore, we assume that *duplicate nonzeros* have been removed, i.e., nonzeros $a_{i'j}$ that are in the same column and are owned by the same processor as a nonzero a_{ij}, where $i' > i$; a duplicate nonzero does not cause extra

ETNA
Kent State University
etna@mcs.kent.edu

communication because v_j has to be sent only once to the owner of a_{ij}. Since the processor that owns a nonzero a_{ij} in matrix column j is important, but the row number i is irrelevant, we can transform the matrix into a $p \times n$ sparse matrix A', which contains a nonzero in position (s, j), $0 \le s < p$, $0 \le j < n$, if and only if processor s has a nonzero in column j of A. (The matrix A' is introduced in [20] as the *communication matrix* corresponding to a partitioned matrix A.) We assume without loss of generality that A has already been transformed into A', so that we can drop the prime. In Section 4, however, where we present our experimental results, we will distinguish between the original matrix A and its communication matrix A'.

Our aim is to assign each input vector component v_j to a processor $\phi(j) \in \mathcal{P}_j$, such that we minimise the *communication cost* C_ϕ of the partitioning ϕ. This cost is defined as

$$C_\phi = \max_{0 \le s < p} \max\{N_{\text{send}}(s), N_{\text{recv}}(s)\}, \tag{1.2}$$

where

$$N_{\text{send}}(s) = \sum_{j:\ \phi(j)=s} (\lambda_j - 1), \tag{1.3}$$

and

$$N_{\text{recv}}(s) = |\{j : 0 \le j < n \ \wedge \ s \in \mathcal{P}_j \ \wedge \ \phi(j) \ne s\}|. \tag{1.4}$$

Sometimes we refer to the *communication cost of a processor* s, which is $C_\phi(s) = \max\{ N_{\text{send}}(s), N_{\text{recv}}(s)\}$. Note that the *number of sends* expressed by $N_{\text{send}}(s)$ is the number of data words sent, and not the number of messages in which they are packaged. We will use the terms ‘send’ or ‘receive’ to denote the communication of a single data word, irrespective of the way this is done (the data word is most likely sent as part of a larger packet). The total communication volume is

$$V_\phi = \sum_{s=0}^{p-1} N_{\text{send}}(s) = \sum_{s=0}^{p-1} N_{\text{recv}}(s). \tag{1.5}$$

We will drop the subscript ‘ϕ’ from C_ϕ, $C_\phi(s)$, and V_ϕ if the partitioning involved is clear from the context.

2. Related work.

Çatalyürek and Aykanat [5] present a one-dimensional matrix partitioning method for square matrices based on hypergraphs. The method is also applicable to rectangular matrices, see [4]. They partition the input and output vectors conformally with the matrix partitioning; for a rowwise matrix partitioning, this means that matrix row i, v_i, and u_i are all assigned to the same processor. Thus, no specific effort is made to balance the communication by finding a better vector partitioning. This may be beneficial for the output vector $\mathbf{u}$, but for the input vector $\mathbf{v}$ this leaves no choice in the partitioning, and it might increase the communication volume V. Çatalyürek and Aykanat [5] do report results, however, on their vector partitioning. For instance, experiments with rowwise partitioning by the HCM variant of their partitioner PaToH [6] on a set of square, structurally nonsymmetric matrices show for $p = 64$, on average, a scaled volume of 0.92 and a maximum scaled volume per processor of 0.025 (only counting the sends, not the receives), which is not too far from the theoretical optimum of 0.0143. The scaling is by matrix size, i.e., the scaled volume is V/n. The maximum scaled volume per processor would probably be somewhat higher if receives were included in the metric as well.

Vastenhouw and Bisseling [23] pose the problem of minimising the cost in the metric we use here. They present a vector partitioning algorithm which is the default in version 1.0 of the Mondriaan package. The algorithm works as follows. First, it handles the components with $\lambda_j > 2$ in random order, trying to minimise $\max_{0 \le s < p}(N_{\rm send}(s) + N_{\rm recv}(s))$. The algorithm assigns v_j to the processor with the current lowest sum. The sum of processor s is initialised at $ncols(s) = |J(s)|$, where $J(s) = \{j : 0 < j < n \wedge s \in \mathcal{P}_j\}$, in an attempt to give the greedy algorithm at least a partial view of the future. The initial sum can be seen as the number of inevitable communication operations: a processor must either send or receive at least one data word if it occurs in a column. If v_j is assigned to processor s, this increases the sum of s by $\lambda_j - 2$. Second, the algorithm handles the components with $\lambda_j = 2$ in an arbitrary order, trying to balance the number of sends with the number of receives. The components with $\lambda_j = 2$ do not increase the sums any more. Let s and s' be the two processors in a column j. The algorithm chooses s as owner of column j if $N_{\rm send}(s) + N_{\rm recv}(s') \le N_{\rm send}(s') + N_{\rm recv}(s)$; otherwise, it chooses s'. This gives rise to one data word being sent in the least busy direction. The numerical experiments in [23] for the five rectangular matrices show reasonable communication balance, with the largest problem instance (matrix `tbdlinux`, $p = 64$) showing a normalised cost of 3.06 (relative to the average volume V/p). A disadvantage of this algorithm is that matrix partitionings with few two-processor columns have little opportunity to optimise the send/receive balance; in the worst case, this may double the communication cost. In the present work, we investigate the performance of this algorithm further and try to improve it; in the remainder of the paper, we denote the original Mondriaan vector partitioning algorithm by ‘Mon’.

Uçar and Aykanat [20] treat the problem of vector partitioning given a certain matrix partitioning, with the objective of minimising the total number of messages and hence the sum of the message latencies while satisfying a balancing constraint on the maximum communication volume per processor, measured in sends. This enables a trade-off between latency and maximum volume. To find a solution, they formulate the problem in terms of a hypergraph with p hyperedges and n vertices, and then try to partition the vertices into p sets using a multilevel hypergraph partitioner, in this case PaToH. Each vertex gets a weight of $\lambda_j - 1$, representing the number of sends for the corresponding vector component v_j. A cut hyperedge s, $0 \le s < p$, with vertices on different processors, means that processor s has to receive a message from these processors, except from itself. For comparison, Uçar and Aykanat also implemented a method which they call the *naive method*, which assigns components in order of decreasing λ_j, trying to balance the sends. The hypergraph method significantly reduces the total number of messages, by about a factor of two compared to the naive method, but it doubles the maximum number of sends per processor and it also increases the total volume by up to 41%. (An increase in communication volume may occur if a component v_j is moved to a processor outside $\mathcal{P}_j$ to reduce the number of messages.) The time of the vector partitioning by the hypergraph method is usually less than that of the matrix partitioning, but it is much more than that of the naive method. The difference between the present work and the work of Uçar and Aykanat is that we try to achieve the utmost in communication balance, ignoring latency, and that we also try to balance the receives of the processors.

A different area, related to the present work, is that of computation load balancing. Viewing communication as just another type of work that has to be balanced among the processors, we may be able to benefit from methods developed to balance computation work. Pinar and Hendrickson [17] propose a general framework for balancing work in complex situations, such as overlapping subdomains in domain decomposition, or computation and communication without synchronisation in between (where work represents the sum of computation time and communication time). They diffuse data and associated work from overloaded processors to others, starting from an initial partitioning, and improving the balance until it is satisfactory. In different work, Pinar and Hendrickson [18] present a method for assigning

ETNA
Kent State University
etna@mcs.kent.edu

computation tasks to processors in situations where this can be done flexibly, i.e., without consequences for the communication in a parallel computation. Each task is of unit size and can be assigned to one processor from a set of processors. An optimal task assignment is found by solving a maximum network-flow problem. In principle, such a method can also be applied to assign communication tasks. The restrictions of the problem formulation mean that the method can be applied to balance the sends or the receives, but not both simultaneously, in the case that $\lambda_j = 2$ for all j.

The best sparse matrix partitioning methods for parallel sparse matrix–vector multiplication are those based on hypergraph partitioners, since these reflect the communication volume accurately in their objective function; graph partitioners only approximate the volume. Currently, a variety of hypergraph partitioners are available: PaToH [4, 5, 6], which was the first hypergraph partitioner to be used for sparse matrix partitioning; hMetis [16], a hypergraph version of the Metis graph partitioner; Mondriaan [23], a two-dimensional hypergraph-based sparse matrix partitioner, which can also be used to solve hypergraph partitioning problems by running it in one-dimensional mode; MONET [15], a hypergraph-based matrix ordering package, which permutes the rows and columns of a matrix to obtain a bordered block-diagonal form; Zoltan [10], a dynamic load balancing library for a wide range of parallel applications, which has recently been extended by a serial hypergraph partitioner [2]; MLpart [3], a multilevel hypergraph partitioner developed for circuit design, a traditional application area of hypergraph partitioning. Par$k$way [19] is a parallel k-way hypergraph partitioner that has been developed for very large Markov transition matrices ($n = \mathcal{O}(10^7)$) from voting models.

3. Algorithms for vector partitioning.

3.1. Special case: two processors per matrix column.

A special case arises if $\lambda_j = 2$ for all j. This will happen if $p = 2$, but it can also happen for larger p. For instance, if the matrix partitioning first splits the columns into two sets of columns, and then splits each set independently in the row direction, the resulting matrix partitioning over four processors satisfies the condition both for rows and columns.

In the special case, every assignment of a vector component v_j to one of the two processors in matrix column j causes one processor to send a data word and the other to receive one. Let $ncols(s)$ be the number of columns in which processor s occurs. Therefore, processor s will have to perform a total of $ncols(s)$ send and receive operations. A lower bound on the communication cost of processor s is thus $\lceil ncols(s)/2 \rceil$. This bound is attained if s is assigned (nearly) half of the vector components corresponding to the columns it shares. A lower bound on the overall communication cost is

$$L_2 = \max_{0 \le s < p} \left\lceil \frac{ncols(s)}{2} \right\rceil . \tag{3.1}$$

We now present an algorithm, Opt2, that assigns the vector components corresponding to the matrix columns in three phases. For brevity, we say that the algorithm assigns matrix columns. Let B_{st} be the number of columns shared by processors s and t, where $0 \le s, t < p$; note that $B_{ss} = 0$. In Phase 1, the algorithm assigns columns shared by s and t in pairs, assigning one column to s and the other to t, until no such pair of columns is left. After these assignments, the remaining number of columns shared by processors s and t is $B'_{st} = B_{st} \bmod 2$. We can view B' as the adjacency matrix of the graph $G = (\mathcal{V}, E)$, where $\mathcal{V} = \{0, \ldots, p-1\}$ and $E \subset \mathcal{V} \times \mathcal{V}$ is defined by $(s,t) \in E$ if and only if $B'_{st} = 1$. Fig. 3.1(b) presents this graph; Fig. 3.1(a) presents a weighted graph corresponding to B.

In Phase 2, the algorithm walks paths in the graph G, each time starting at a vertex with odd degree. After an edge (s,t) is traversed in the direction from s to t, it is removed from

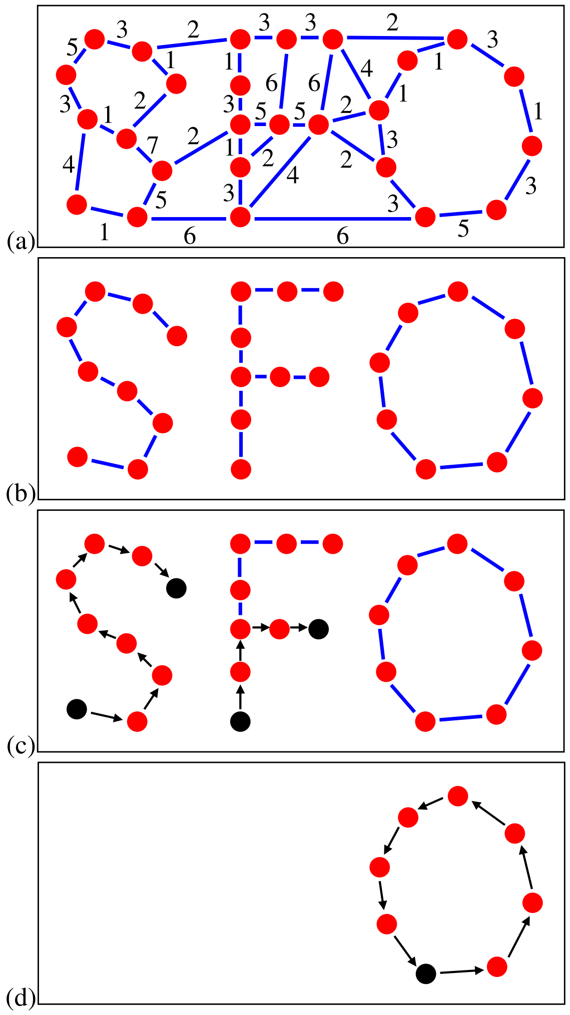

FIG. 3.1. *Path-walking algorithm for the graph that represents the communication of the input vector in parallel sparse matrix–vector multiplication. (a) Weighted undirected graph, where each vertex s represents a processor, each edge (s,t) a set of columns shared by processors s and t, and each edge weight B_{st} the number of columns shared. (b) The unweighted graph obtained by assigning pairs of shared columns to processors, each time one column to s and the other to t. The adjacency matrix of this graph is B', defined by $B'_{st} = B_{st} \bmod 2$. (c) The graph is traversed by walks starting at an odd-degree vertex, walking along the edges while removing them on the way, until a dead end is reached. This phase contains three such walks; the first two are shown. Start and end points are shown in black. (d) The graph is traversed by walks starting at an even-degree vertex.*

E, and the corresponding column is assigned to s. A path is finished when the walk reaches a dead end, i.e., when the vertex reached has no more edges left. After the degree of a vertex becomes zero, the vertex is removed. This phase ends when all remaining vertices have even degree. Phase 2 is illustrated by Fig. 3.1(c). In Phase 3, the algorithm walks paths in the remaining graph until the graph is empty. Phase 3 is illustrated by Fig. 3.1(d). We will prove optimality of the algorithm by using the following lemma.

LEMMA 3.1. *Let $G = (\mathcal{V}, E)$ be a graph and $(s_0, s_1, \ldots, s_r)$ be a path in G with all edges distinct. Let $\deg(s_0)$, the degree of s_0, be odd. Assume that the path cannot be extended any more by adding an edge from E that is distinct from the edges already contained in the* *path. Then:* $s_0 \neq s_r$ *and a traversal of the path with edge removal changes* $\deg(s_0)$ *from odd to even and* $\deg(s_r)$ *from odd to zero, and it does not change the parity of* $\deg(s)$ *for* $s \neq s_0, s_r$.

Proof. The removal of the first edge (s_0, s_1) makes $\deg(s_0)$ even. If $\deg(s_0)$ becomes zero, the walk cannot reach s_0 any more; otherwise, if the walk enters s_0, it will leave. Thus, the end vertex s_r must differ from the start vertex s_0 and $\deg(s_0)$ remains even. Furthermore, $\deg(s_r)$ becomes zero since no edges remain. The parity of $\deg(s_i)$, $i = 1, \ldots, r-1$, does not change when the path reaches an intermediate vertex s_i, because one edge is removed on entry and one on exit. Note that vertices may occur more than once on a path; our statements still hold then. Just before the last edge is removed, we have $\deg(s_r) = 1$; at the start of the walk, $\deg(s_r)$ must have been odd. Vertices not on the path are not affected by the traversal. ☐

THEOREM 3.2. *Let* $p, n \in \mathbf{N}$, *with* $p \geq 2$, $n \geq 1$. *Let* $\mathcal{P}_j \subset \{0, \ldots, p-1\}$ *with* $|\mathcal{P}_j| = 2$ *represent the pair of processors that share column* j, *for* $j = 0, \ldots, n-1$. *Then algorithm Opt2 produces a vector partitioning* $\phi : \{0, \ldots, n-1\} \to \{0, \ldots, p-1\}$ *with minimal cost* C_ϕ.

Proof. The algorithm terminates because the number of columns and edges is finite and because each walk in Phases 2 and 3 removes at least one edge. Each column is assigned to a processor at some time during the algorithm, either in Phase 1 or when its corresponding edge is removed in Phase 2 or 3. At the end of the algorithm, a complete vector partitioning ϕ has been obtained.

After Phase 1, $N_{\text{send}}(s) - N_{\text{recv}}(s) = 0$ for all vertices s. The value $N_{\text{send}}(s) - N_{\text{recv}}(s)$ does not change for an intermediate vertex s in a walk in Phase 2. The value can change, however, if s is a start or end vertex in Phase 2. This can happen only once for every s, $0 \leq s < p$, because a vertex can only once be a start or end vertex of a walk. This is because a start vertex s_0 has an odd degree, which becomes even after the walk, see Lemma 3.1. Therefore, s_0 cannot become a start vertex again in this phase. Furthermore, it cannot become an end vertex of a walk, because of its even degree. An end vertex has no edges left, is removed, and hence does not occur again in further walks.

In Phase 3, the first walk starts at a vertex s_0 with even degree. After traversal and removal of the first edge (s_0, s_1), the two vertices s_0 and s_1 have odd degree, but all other vertices have even degree. We can apply Lemma 3.1 to the path $(s_1, \ldots, s_r)$ representing the remainder of the first walk. The lemma says that $s_r \neq s_1$ and that $\deg(s_r)$ is odd at the start of the remaining walk, so $s_r = s_0$ must hold. Thus, the path $(s_0, \ldots, s_r)$ must be a cycle. After walking a complete path, all vertices have even degree again, and further walks can be carried out in the same way. Because each path is a cycle, the value $N_{\text{send}}(s) - N_{\text{recv}}(s)$ does not change for the vertices s on the paths.

As a result of the algorithm, $|N_{\text{send}}(s) - N_{\text{recv}}(s)| \leq 1$ for all s, and equality $|N_{\text{send}}(s) - N_{\text{recv}}(s)| = 1$ only happens if $\deg(s)$ is odd at the start of Phase 2. Because $\deg(s) = \sum_{t \neq s} B'_{st}$, this is equivalent to $ncols(s) = \sum_{t \neq s} B_{st}$ being odd. This proves that the resulting assignment is optimal. ☐

3.2. General lower bounds on the communication cost. A simple lower bound on the communication cost based on the communication volume is

$$L_{\text{vol}} = \left\lceil \frac{V}{p} \right\rceil . \tag{3.2}$$

This bound occurs because not all processors can have a communication cost below the average V/p, so that in every vector partitioning there must exist a processor s with $C(s) \geq V/p$.

Because costs are integers, we even have $C(s) \geq \lceil V/p \rceil$. Therefore, $C \geq \lceil V/p \rceil$, which shows that the cost C must be at least $L_{\rm vol}$.

A different lower bound can be obtained by considering the vector components that a processor would like to possess in order to minimise its cost. Assume that a processor s can freely choose its components. Every vector component v_j that the processor obtains will decrease its number of receives by one, irrespective of the size of the corresponding matrix column. At the same time, this will increase the number of sends by $\lambda_j - 1$. Thus, a processor would prefer to obtain vector components corresponding to small columns, i.e., with low λ_j. If given a free choice, a processor would take components in order of increasing column size, stopping when adding another component would give a larger number of sends than receives. This leads to the lower bound expressed by the following theorem.

THEOREM 3.3. *Let $p, n, s \in \mathbf{N}$, with $p \geq 2$, $n \geq 1$, and $0 \leq s < p$. Let $\mathcal{P}_j \subset \{0, \ldots, p-1\}$ and $\lambda_j = |\mathcal{P}_j|$, for $j = 0, \ldots, n-1$. Assume that $\lambda_j \geq 2$ for all j, and $\lambda_j \leq \lambda_{j'}$ if $j < j'$. Define*

$$J_k = \{j : 0 \leq j < k \wedge s \in \mathcal{P}_j\},$$

for $k = 0, \ldots, n$. Let k be the largest integer with $k \leq n$ for which

$$\sum_{j \in J_k} (\lambda_j - 1) \leq |J_n \backslash J_k|.$$

Define $L(s) = |J_n \backslash J_k|$. Then for every vector assignment ϕ with $\phi(j) \in \mathcal{P}_j$ for all j, we have

$$C_\phi \geq L(s).$$

Proof. Let J_k be the set defined in the theorem and let $J = J_n$ be the set of all indices j with $s \in \mathcal{P}_j$. Let ϕ be a vector assignment with $\phi(j) \in \mathcal{P}_j$ for all j, and $J' \subset J$ be the set of indices j with $\phi(j) = s$. We will transform J' into J_k by changes that do not increase the cost for processor s. This will prove that the cost of J_k is less than or equal to that of J'. First, we transform J' into a J_r by adding $j_{\rm min}$, the smallest $j \in J \backslash J'$ to J', and removing $j_{\rm max}$, the largest j from J', and repeating this procedure as long as $j_{\rm min} < j_{\rm max}$. Each change reduces the cost, or keeps it the same, because it replaces an index j with a larger size λ_j by one with a smaller size, or a lower numbered one of equal size. The changes fill the holes in J' until it contains all the indices from J up to a certain size, i.e., $J' = J_r$ for a certain r. If $J_k \subsetneq J_r$, we have $\sum_{j \in J_r} (\lambda_j - 1) > |J \backslash J_r|$, by the definition of k. Thus, we can remove the largest j from J_r while not increasing the cost. (This is because the number of sends decreases, and the number of receives increases by one, but it will not exceed the number of sends before the change.) We can repeat this until $r = k$. If $J_k \subsetneq J_r$ does not hold, we have $J_k = J_r$, or $J_r \subsetneq J_k$. If $J_k = J_r$, we are done, but otherwise we can add the smallest element of $J \backslash J_r$ to J_r. This does not increase the cost, because the number of receives decreases and this number determines the cost. We have proven that the cost for processor s in an assignment ϕ is at least the cost incurred by assigning all components from J_k to s, which is $L(s) = |J \backslash J_k|$. □

By using Theorem 3.3 we can determine a local lower bound $L(s)$ for each processor s. The resulting lower bound for the overall communication cost is

$$L = \max_{0 \leq s < p} L(s). \tag{3.3}$$

We call this bound the *local lower bound* because it is based on a local assignment of vector components, not taking other processors into account. In contrast, the volume-based bound L_{vol} represents a global view, but it does not take local details into account. Combining the two lower bounds, we have $\max\{L_{\mathrm{vol}}, L\}$ as the best lower bound.

The bound $L(s)$ for processor s can be generalised to the situation where a processor does not start from scratch, but instead already has some communication obligations (for instance for vector components that have already been assigned). Let N_{send} be the number of data words the processor has to send already, and N_{recv} the number it has to receive. Choose k as large as possible, while ensuring that

$$N_{\mathrm{send}} + \sum_{j \in J_k} (\lambda_j - 1) \le N_{\mathrm{recv}} + |J \backslash J_k|. \tag{3.4}$$

The lower bound for processor s then equals

$$L(s) = N_{\mathrm{recv}} + |J \backslash J_k|. \tag{3.5}$$

The number of data words the processor has to send to attain the lower bound is

$$L_{\mathrm{send}}(s) = N_{\mathrm{send}} + \sum_{j \in J_k} (\lambda_j - 1). \tag{3.6}$$

The number of data words it has to receive to attain the lower bound equals $L(s)$.

3.3. Vector partitioning by the local-bound algorithm. We can use the local lower bound as the basis for a heuristic algorithm. The heuristic is to choose the processor $s_{\max}$ that has the highest local bound $L(s)$ for its current index set $J(s)$, and let this processor choose a vector component v_j. It chooses a component with minimal λ_j, and thus will not increase its lower bound. This heuristic tries to avoid increasing the highest lower bound $L(s)$ and hence raising L. The algorithm then updates the number of sends and receives incurred, removes j from all the index sets $J(s)$, and updates the lower bounds $L(s)$ and the associated number of sends $L_{\mathrm{send}}(s)$ needed to attain the lower bound.

A processor stops accepting new components when its number of sends $N_{\mathrm{send}}(s)$ equals the current optimal number of sends $L_{\mathrm{send}}(s)$. (By definition, $N_{\mathrm{send}}(s) \le L_{\mathrm{send}}(s)$.) Accepting more components would only increase $L(s)$, so that instead it is better to stop and let other processors accept components. The algorithm terminates when no processor is willing any more to accept new components. It may be possible that some (large) components are left over at the end of the algorithm; these can be handled by greedy assignment, see the next subsection. Algorithm 3.1 presents the details of the local-bound based partitioning. The notation 'argmax' used in Algorithm 3.1 means an index for which the maximum is obtained, and similarly for 'argmin'. An efficient implementation of the algorithm would use the compressed column storage (CCS) data structure (see e.g. [1]) for the communication matrix A' to facilitate access to the processors in each column, and compressed row storage (CRS) with the nonzeros of each row stored in increasing order to enable direct access for processor s to the next local component v_j with minimal λ_j.

3.4. Greedy assignment. Vector partitioning by *greedy assignment* (GA) is done by handling vector components in an arbitrary order, each time assigning a component v_j to a processor $\phi(j)$ that would have the minimum current cost if it were to obtain the component. This assignment is determined by first incrementing $N_{\mathrm{recv}}(s)$ for all $s \in \mathcal{P}_j$, and then finding a processor s from $\mathcal{P}_j$ with minimal value $\max\{N_{\mathrm{send}}(s) + \lambda_j - 1, N_{\mathrm{recv}}(s) - 1\}$. The current send and receive counts are then updated by decrementing $N_{\mathrm{recv}}(\phi(j))$ and adding $\lambda_j - 1$ to $N_{\mathrm{send}}(\phi(j))$. GA can also be applied when part of the components have already been assigned by a different method.

```
input:  $A$ is a $p \times n$ sparse matrix,
        $\mathcal{P}_j = \{i : 0 \leq i < p \wedge a_{ij} \neq 0\}$, for $0 \leq j < n$.
output: $\phi = \mathrm{distr}(\mathbf{v})$: vector distribution over $p$ processors,
        such that $\phi(j) \in \mathcal{P}_j$, for $0 \leq j < n$.

for all $s : 0 \leq s < p$ do
        $N_{\mathrm{send}}(s) := 0$;
        $N_{\mathrm{recv}}(s) := 0$;
        $J(s) := \{j : 0 \leq j < n \wedge s \in \mathcal{P}_j\}$;
        $k := \max\{k : 0 \leq k \leq n \wedge \sum_{j \in J_k(s)} (\lambda_j - 1) \leq |J(s) \backslash J_k(s)|\}$,
                where $J_k(s) = \{j : j \in J(s) \wedge 0 \leq j < k\}$;
        $L(s) := |J(s) \backslash J_k(s)|$;
        $L_{\mathrm{send}}(s) := \sum_{j \in J_k(s)} (\lambda_j - 1)$;
        if $L_{\mathrm{send}}(s) = 0$ then active$(s)$ := false;
                        else active$(s)$ := true;

while $(\exists s : 0 \leq s < p \wedge \mathrm{active}(s))$ do
        $s_{\max} := \mathrm{argmax}\{L(s) : 0 \leq s < p \wedge \mathrm{active}(s)\}$ ;
        $j := \mathrm{argmin}\{\lambda_j : j \in J(s_{\max})\}$;
        $\phi(j) := s_{\max}$;
        $N_{\mathrm{send}}(s_{\max}) := N_{\mathrm{send}}(s_{\max}) + \lambda_j - 1$;
        for all $s : s \in \mathcal{P}_j \wedge s \neq s_{\max}$ do
                $N_{\mathrm{recv}}(s) := N_{\mathrm{recv}}(s) + 1$;
        for all $s : s \in \mathcal{P}_j$ do
                $J(s) := J(s) \backslash \{j\}$;
        for all $s : s \in \mathcal{P}_j \wedge s \neq s_{\max}$ do
                $k := \max\{k : N_{\mathrm{send}}(s) + \sum_{j \in J_k(s)} (\lambda_j - 1) \leq N_{\mathrm{recv}}(s) + |J(s) \backslash J_k(s)|\}$,
                        where $J_k(s) = \{j : j \in J(s) \wedge 0 \leq j < k\}$;
                $L(s) := N_{\mathrm{recv}}(s) + |J(s) \backslash J_k(s)|$;
                $L_{\mathrm{send}}(s) := N_{\mathrm{send}}(s) + \sum_{j \in J_k(s)} (\lambda_j - 1)$;
        for all $s : s \in \mathcal{P}_j \wedge N_{\mathrm{send}}(s) = L_{\mathrm{send}}(s)$ do
                active$(s)$ := false;
```

ALGORITHM 3.1
Local-bound based vector partitioning.

3.5. Greedy improvement of a given partitioning. A given partitioning ϕ can be improved by a simple *greedy improvement* (GI) procedure, as follows. A vector component v_j is chosen at random, and for each processor $s \in \mathcal{P}_j$, $s \neq \phi(j)$, the change in cost caused by reassigning v_j to s is computed. If there is a change that reduces the cost, it is carried out and the vector component is reassigned. In case of several possible changes, the best one is taken by using a secondary criterion. Often, the size of the cost reduction is the same for all cost-reducing reassignments, so the size is less suitable for breaking ties. (It could be used together with a ternary criterion.) We break ties by reassigning the component to the processor in $\mathcal{P}_j$ with the current least number of sends. The reason is that the number of sends of a processor is affected most by a reassignment, more than the number of receives. Any remaining ties are resolved arbitrarily. After that, another component j is randomly chosen, and the same procedure is executed. This is repeated until no more improvement can be obtained. Note that GI is based on moves, like the well-known Fiduccia-Mattheyses algorithm [13] which is at the heart of today's successful hypergraph partitioners, but GI is less sophisticated (and cheaper) since it does not accept cost increases and does not select a move with largest possible gain.

This procedure can be implemented efficiently by organising it in passes. A new pass starts every time a reduction has been obtained, or at the start of the GI algorithm. The indices j are stored in an array, initially in the natural order. The first part of the array stores the indices of the columns that have not been tried yet in the current pass and the second part the indices of the remaining columns. The number $N_{\mathtt{tried}}$ is used to keep track of the number of components tried in the current pass. If reassigning a vector component v_j cannot reduce the cost, the index j is swapped with the last untried index, and $N_{\mathtt{tried}}$ is incremented. If reassignment succeeds, $N_{\mathtt{tried}}$ is reset to 0. Thus, a random index can always be chosen from a contiguous subarray of untried indices.

4. Results. The algorithms presented in the previous section have been implemented within the sparse matrix partitioning package Mondriaan[1]. First, the matrices were partitioned on a 375 MHz Sun Enterprise 420 computer at Sandia National Laboratories in Albuquerque, NM, which has four Sparc-2 processors, 4 Gbyte RAM, 64-bit hardware arithmetic, and which runs the Solaris 8 operating system. Since the Mondriaan program itself is serial, only one processor is used per program run. Second, numerical experiments with different vector partitioning methods were performed for the partitioned matrices on an 867 MHz Apple PowerBook G4 computer with 768 Mbyte RAM, 32-bit hardware arithmetic, and a PowerPC G4 processor, which runs the Mac OS 10.2 operating system.

We have checked the quality of the vector partitioning by using a test set of sparse matrices from publicly available collections, supplemented with a few matrices from Sandia applications and a few matrices from our own applications. Table 4.1 presents the matrices; the matrix `bcsstk32` was obtained from the Rutherford–Boeing collection [11, 12]; the matrices `lhr34` and `nug30` were obtained from the University of Florida collection [9]. The matrix `rhpentium_new` provided by Robert Hoekstra represents a circuit simulation by the Sandia package Xyce for part of a Pentium processor. The matrix `polyDFT` originates in a polymer self-assembly simulation by the Density Functional Theory package Tramanto from Sandia. The matrix `tbdlinux` is a term-by-document matrix describing the documentation of the SuSE Linux 7.1 operating system. The matrix `cage13` [22] (available through [9]) is a stochastic matrix describing transition probabilities in the cage model of a DNA polymer of length 13 moving in a gel under the influence of an electric field. Each matrix has been partitioned using the Mondriaan matrix partitioner with default parameters, allowing a computational load imbalance of 3%.

Table 4.2 presents the partitioned sparse matrices. We can view the resulting vector partitioning problem (for a vector $\mathbf{v}$) as a sparse $m' \times n'$ matrix A', where the $m' = p_{\mathrm{comm}}$ rows represent the processors that are active in communication and the n' columns represent the columns from the original matrix with $\lambda_j \geq 2$. (For a vector $\mathbf{u}$, we can transpose the partitioned matrix A and proceed in the same way.) From now on, we distinguish between A and A', and between p_{comm} and p. Table 4.2 presents several properties of A' such as the communication volume, two lower bounds on the communication cost, the optimal solution for the special case $\lambda_j \leq 2$, if applicable, and a balance ratio, for the seven matrices from the test set and for $p = 4, 16, 64$. Note that

$$V \geq n', \tag{4.1}$$

[1]The new vector partitioning algorithms will be made available in version 2.0 of Mondriaan, see `http://www.math.uu.nl/people/bisseling/Mondriaan`, to be released soon.

TABLE 4.1
Properties of the test matrices. For each matrix, we give: the number of rows m, the number of columns n, the number of nonzeros $nz(A)$, and the origin.

Matrix	m	n	$nz(A)$	Application area
`rhpentium_new`	25187	25187	258265	circuit simulation
`lhr34`	35152	35152	764014	chemical engineering
`nug30`	52260	379350	1567800	linear programming
`bcsstk32`	44609	44609	2014701	structural engineering
`tbdlinux`	112757	20167	2157675	information retrieval
`polyDFT`	46176	46176	3690048	polymer simulation
`cage13`	445315	445315	7479343	DNA electrophoresis

because every column in A' has two or more processors and hence causes at least one communication. If $V = n'$, this means that all columns have two nonzeros, so that our optimal algorithm Opt2 from Subsection 3.1 is applicable. Furthermore, we have

$$V + n' = nz(A'), \tag{4.2}$$

because the number of communications caused by a column in A' equals its number of nonzeros minus 1. It is easily verified that Eqns (4.1) and (4.2) are indeed satisfied for the data given in Table 4.2. The bounds given in the table are the volume-based bound $L_{\mathrm{vol}} = \lceil V/p_{\mathrm{comm}} \rceil$ and the local lower bound L, see Eqn (3.3). The ratio $\max\{L_{\mathrm{vol}}, L\}/(V/p)$ gives the balance of the communication load in case we manage to attain the best lower bound. Perfect balance corresponds to a ratio of 1.

For each matrix, both the input vector $\mathbf{v}$ and the output vector $\mathbf{u}$ are partitioned. A problem instance is thus a triple `matrix`/p/vector. The instances `nug30`/p/$\mathbf{v}$ and `bcsstk32`/4/$\mathbf{v}$ are omitted because the preceding matrix partitioning produces a one-dimensional column partitioning, so that no communication is needed for $\mathbf{v}$. The number of communicating processors p_{comm} can be less than the available number of processors p: for instance, $p_{\mathrm{comm}} = 38$ for `rhpentium_new`/64/$\mathbf{v}$ and `lhr34`/64/$\mathbf{u}$. A processor does not communicate if all the components j that it owns satisfy $\lambda_j = 0$ or $\lambda_j = 1$; for $\lambda_j = 0$, the corresponding matrix column j must be empty, but for $\lambda_j = 1$ it may have many nonzeros. Table 4.2 shows that the lower bound L is the best (highest) bound for 34 out of 38 instances, that $L = L_{\mathrm{vol}}$ for three instances, and that the bound L_{vol} is best for only one instance, namely `nug30`/4/$\mathbf{u}$. The table also shows that applicability of Opt2 is not restricted to the case $p_{\mathrm{comm}} \leq 2$; e.g., `lhr34`/16/$\mathbf{v}$ with $p_{\mathrm{comm}} = 10$ can be solved by the optimal algorithm. The ratios given in the last column of Table 4.2 show that even if the balancing problem can be solved optimally, this does not mean that the resulting balance is perfect. For 17 problem instances out of 38, the ratio exceeds 2, meaning that one processor has to communicate at least twice the average amount based on V and p; the maximum ratio observed is 4.02 for `lhr34`/64/$\mathbf{u}$. This imbalance is inevitable and is caused by the preceding matrix partitioning.

Table 4.3 presents results of Mon, the algorithm from [23] implemented in the original Mondriaan package, version 1.0, which has been explained in Section 2, and of LB, the local-bound algorithm presented in Subsection 3.3, with and without the greedy improvement procedure presented in Subsection 3.5. The matrix partitioning was fixed by using Mondriaan once with a fixed random number seed, giving the partitioned matrices presented in Table 4.2. The vector partitioning was performed a hundred times, each time with a different seed. Like the Mondriaan matrix partitioning, the Mondriaan vector partitioning is a randomised algorithm. Still, there may be some dependence on the input ordering. The LB vector partitioning

TABLE 4.2
Properties of the $m' \times n'$ communication matrix A' for various processor numbers p and for both the input vector $\mathbf{v}$ *and the output vector* $\mathbf{u}$*. For each problem instance, we give: the communication volume V; the number of active processors $m' = p_{\mathrm{comm}}$; the number of columns n' that cause communication; the number of nonzeros of A'; the lower bounds L_{vol}, L on the communication cost; the ratio $\max\{L_{\mathrm{vol}}, L\}/(V/p)$. The values in boldface in the column of L show the cases where the optimal algorithm Opt2 is applicable; in all these cases, Opt2 attained the lower bound L.*

Matrix	p	Vector	V	m'	n'	$nz(A')$	L_{vol}	L	Ratio
`RHPentium_new`	4	**v**	13791	4	12656	26447	3448	5822	1.69
		u	5614	2	5614	11228	2807	**2807**	2.00
	16	**v**	9888	10	9215	19103	989	1722	2.79
		u	18519	16	14469	32988	1158	1941	1.68
	64	**v**	10108	38	7930	18038	266	517	3.27
		u	26223	64	17317	43540	410	621	1.52
`lhr34`	4	**v**	1172	4	1171	2343	293	381	1.30
		u	253	2	253	506	127	**127**	2.01
	16	**v**	1069	10	1069	2138	107	**234**	3.50
		u	2694	15	2512	5206	180	344	2.04
	64	**v**	8372	64	7813	16185	131	270	2.06
		u	2008	38	1946	3954	53	126	4.02
`nug30`	4	**u**	56348	4	39413	95761	14087	12435	1.00
	16	**u**	136712	16	47237	183949	8545	8823	1.03
	64	**u**	247032	64	51413	298445	3860	5816	1.51
`bcsstk32`	4	**u**	1986	4	1968	3954	497	718	1.45
	16	**v**	4872	16	4750	9622	305	741	2.43
		u	3106	12	3094	6200	259	456	2.35
	64	**v**	11839	64	10759	22598	185	366	1.98
		u	8001	56	7474	15475	143	378	3.02
`tbdlinux`	4	**v**	27800	4	15740	43540	6950	9149	1.32
		u	3263	2	3263	6526	1632	**1632**	2.00
	16	**v**	44179	16	15871	60050	2762	5906	2.14
		u	30299	14	15194	45493	2165	2900	1.53
	64	**v**	76914	64	16261	93175	1202	3407	2.83
		u	66564	58	21652	88216	1148	1832	1.76
`PolyDFT`	4	**v**	5352	4	5352	10704	1338	**1410**	1.05
		u	3636	4	3636	7272	909	**1135**	1.25
	16	**v**	19756	16	17571	37327	1235	2274	1.84
		u	16405	14	14820	31225	1172	2346	2.29
	64	**v**	41625	64	30083	71708	651	1281	1.97
		u	37541	64	28604	66145	587	1454	2.48
`cage13`	4	**v**	57054	4	57054	114108	14264	**19497**	1.37
		u	58804	4	58804	117608	14701	**16466**	1.12
	16	**v**	95032	16	86329	181361	5940	12313	2.07
		u	160761	16	139624	300385	10048	15152	1.51
	64	**v**	236759	64	194172	430931	3700	5456	1.47
		u	208314	64	172388	380702	3255	5457	1.68

is deterministic, since it uses the natural vector ordering. To remove any possible dependence on the input ordering, the columns of A' were randomly permuted before each run, both for Mon and LB. This makes the average results more meaningful, but it also enhances the chances of encountering a good solution in one of the runs. The greedy improvement procedure is terminated if a pass through all the vector components shows that no improvement can be found any more, or if a preset maximum number of tries is reached, $10n$ for partitioning of $\mathbf{v}$ and $10m$ for $\mathbf{u}$. Thus, the maximum number of tries is of the order of the size of the original matrix A.

The average communication costs given in Table 4.3 show that LB is better than Mon for 18 out of 38 problem instances, equal to Mon for 12 instances, and worse for 8 instances. With greedy improvement (GI) switched on, LB is better for 11 instances, equal for 24, and worse for 3. All instances with equal performance (with or without GI) correspond to cases where both algorithms solve the problem to optimality in *every run* of the algorithm. LB+GI solves the problem to optimality in every run for 34 instances; Mon+GI does this for 25 instances. We may conclude that LB is superior to Mon. Furthermore, it is clear that GI is useful, since it improves the average result in most cases for Mon, and in 9 cases for LB. Although LB is better than Mon, we need not discard the latter: Mon+GI performs better on average than LB+GI for `nug30`/4/$\mathbf{u}$, `cage13`/16/$\mathbf{u}$, and `cage13`/64/$\mathbf{v}$. This means that it is worthwhile to try both LB+GI and Mon+GI, instead of just using the better method LB+GI. For `nug30`/4/$\mathbf{u}$, with ratio 1.00 in Table 4.2, Mon+GI managed to achieve an optimal solution and hence a perfect balance in at least one run.

The best communication costs given in Table 4.3 show that all problems, except `nug30`/16/$\mathbf{u}$, can be solved to optimality by one of the methods in at least one of the runs. The cost 8935 of the best solution for `nug30`/16/$\mathbf{u}$ is 1.3% higher than the best bound $L = 8823$. We tried to improve the solution in 10,000 additional runs, but only obtained a marginal reduction of the cost to 8931.

In an actual application, it is possible to perform several runs of the vector partitioner and keep the best solution, because vector partitioning by our methods is cheap and takes much less time than the preceding matrix partitioning. Our largest problem `cage13`/64/$\mathbf{v}$ took about 9 s for both Mon+GI and LB+GI on the PowerBook computer, whereas the corresponding matrix partitioning took 1123 s on the same machine.

5. Conclusion and future work.

We have presented two new algorithms and a new lower bound for solving the vector partitioning problem for parallel sparse matrix–vector multiplication. We have concentrated on the input vector, since the partitioning of the output vector is a similar problem. The first algorithm, Opt2, is optimal and can be applied in the special case that every matrix column is owned by at most two processors. This situation occurs in particular for small p, but also if the preceding matrix partitioner has been highly successful, which often limits the number of processors that own a column. The second algorithm, LB, is a heuristic that in practice often finds the optimal solution, in particular if it is post-processed by greedy improvement. The lower bound L helps to steer the use of the different vector partitioners; in particular, it tells us when we can stop attempts at further improvement. The lower bound seems sharp, since it could be attained in all practical cases we tested, except one. Still, we have to realise that practical communication matrices A' are very special, since they typically contain many columns with a small number of nonzeros, and in most cases two nonzeros. For other types of matrices, other heuristics may be needed. The general vector partitioning problem presented in this paper is NP-complete, which is proven by Ali Pinar in Appendix A.

A good vector partitioning must have a good balance of the communication load, for instance in iterative linear system solvers. The balance is measured by the maximum amount of

TABLE 4.3

Communication cost for different vector partitioning methods. The value given is the percentage by which the cost exceeds the best known lower bound, $\max\{L_{\mathrm{vol}}, L\}$. *A value '0' means exactly 0, and hence an optimal solution. For each problem, we give the average over 100 runs of the vector partitioning by: Mon, the algorithm implemented in the original Mondriaan package, version 1.0; Mon+GI, which is Mon followed by greedy improvement; LB, the local-bound algorithm; LB+GI. Furthermore, we give the best result obtained in the runs by the methods with GI.*

Matrix	p	Vector	Average (%)				Best (%)	
			Mon	Mon+GI	LB	LB+GI	Mon+GI	LB+GI
`RHPentium_new`	4	$\mathbf{v}$	0	0	0	0	0	0
		$\mathbf{u}$	0	0	0	0	0	0
	16	$\mathbf{v}$	0	0	0	0	0	0
		$\mathbf{u}$	3.4	0.03	0	0	0	0
	64	$\mathbf{v}$	1.5	0.2	0	0	0	0
		$\mathbf{u}$	12.6	0.3	0	0	0	0
`lhr34`	4	$\mathbf{v}$	0	0	4.0	0	0	0
		$\mathbf{u}$	0	0	0	0	0	0
	16	$\mathbf{v}$	0.004	0	7.9	0	0	0
		$\mathbf{u}$	0.03	0	0.1	0	0	0
	64	$\mathbf{v}$	0	0	0	0	0	0
		$\mathbf{u}$	0.02	0	3.0	0	0	0
`nug30`	4	$\mathbf{u}$	0.003	0.003	0.6	0.6	0	0.6
	16	$\mathbf{u}$	15.7	7.6	8.4	1.4	7.2	1.3
	64	$\mathbf{u}$	32.7	12.0	0	0	11.6	0
`bcsstk32`	4	$\mathbf{u}$	0	0	0	0	0	0
	16	$\mathbf{v}$	0.001	0	0	0	0	0
		$\mathbf{u}$	0.02	0	0	0	0	0
	64	$\mathbf{v}$	0.006	0	0.7	0	0	0
		$\mathbf{u}$	0.07	0	0	0	0	0
`tbdlinux`	4	$\mathbf{v}$	31.8	0	0	0	0	0
		$\mathbf{u}$	0	0	0	0	0	0
	16	$\mathbf{v}$	34.9	5.6	0	0	5.3	0
		$\mathbf{u}$	6.0	1.7	0	0	1.5	0
	64	$\mathbf{v}$	22.5	7.3	0	0	6.8	0
		$\mathbf{u}$	30.0	9.8	0.09	0	9.0	0
`PolyDFT`	4	$\mathbf{v}$	0	0	0	0	0	0
		$\mathbf{u}$	0	0	0	0	0	0
	16	$\mathbf{v}$	1.0	0	0	0	0	0
		$\mathbf{u}$	0.003	0	0	0	0	0
	64	$\mathbf{v}$	14.1	0.8	0	0	0.5	0
		$\mathbf{u}$	3.7	0.4	0	0	0.1	0
`cage13`	4	$\mathbf{v}$	0	0	0	0	0	0
		$\mathbf{u}$	0	0	0	0	0	0
	16	$\mathbf{v}$	0	0	0	0	0	0
		$\mathbf{u}$	0.001	0	3.0	0.4	0	0.4
	64	$\mathbf{v}$	0.07	0.02	3.7	0.9	0	0.8
		$\mathbf{u}$	3.0	0	0	0	0	0

communication per processor, which is an important metric, perhaps second only to the metric of total communication volume. As communication will become more important when applications scale up to thousands of processors, balancing communication will become crucial. This gives rise to a class of new and interesting optimisation problems.

A promising avenue of research may be trying to generalise computation balancing methods, such as by Pinar and Hendrickson [18], so that they can be applied to communication balancing as well. Here, it is crucial that both sends and receives are balanced; to complicate matters, these two objectives depend on each other. Another issue is that communication tasks can vary in size; for the algorithm in [18], this would mean that the assumption of unit task size must be dropped.

Our problem originated in parallel sparse matrix–vector multiplication, but it has a wider range of applicability, within the field of parallel computing and beyond. One way of viewing the problem is as follows. We have to distribute items among a group of people. For each item, a specific subgroup is interested in owning it. One member of the subgroup wins and becomes the owner. This winner has to compensate all the losers, each with the same amount of money, say 1 dollar or euro. The problem is to distribute the items such that the maximum amount of money any person has to pay or receive is minimised. (Note that this problem is not about minimising the net amount of money paid or received.)

The problem for the special case that could be solved optimally by algorithm Opt2 may also have wider application, for instance in parallel molecular dynamics simulations. Because of Newton's Third Law of Motion, each 2-atom force at the boundary between two processors can be computed by either of the processors involved. The processor that computes the force must send the result to the other processor, which must receive it. Algorithm Opt2 can be used to balance such communication obligations.

Several open research questions remain, such as: can the general vector partitioning problem be solved optimally under less restrictive assumptions? Can the methods be generalised to the situation where the vector components have communication weights, so that sending component v_j to a processor costs w_j time units; this occurs sometimes in the context of hypergraph partitioning for circuit simulation. Can we extend the methods to the situation, not uncommon for square matrices, where we require the same partitioning for the input and output vectors? If the matrix is symmetric and has been partitioned symmetrically, this is easy, but otherwise it is a two-objective optimisation problem, which is much harder to solve. (The original Mondriaan package contains an option distr($\mathbf{u}$) = distr($\mathbf{v}$), but often the communication load is out of balance for this option, because there is not much choice in the assignment of vector components to processors.)

The existence of a lower bound $L > V/p$ points to unavoidable imbalance in the communication for a given matrix partitioning. An optimal solution to the vector partitioning problem may still have imperfect balance. Another indicator of such imbalance is $p_{\mathrm{comm}} < p$, meaning that not all processors participate in the communication. This leads to a bound $L_{\mathrm{vol}} = \lceil V/p_{\mathrm{comm}} \rceil > V/p$. To reduce this imbalance, we must modify the preceding matrix partitioning. The challenge is to do this without increasing the communication volume. We are currently investigating lower-bound based tie-breaking in the matrix partitioner as one means of achieving perfect communication balance.

Appendix. Vector partitioning is NP-complete. *Proof provided by Ali Pinar, Lawrence Berkeley National Laboratory.*

We can formulate the vector partitioning problem as the following decision problem, which we call the *Column Assignment* problem.

INSTANCE: Positive integers p, n, a $p \times n$ matrix A with elements $a_{ij} \in \{0, 1\}$ and with $\lambda_j = |\{i : 0 \le i < p \wedge a_{ij} = 1\}| \ge 2$ for $j = 0, \dots, n-1$, and a positive integer maximum

$$A = \left[\begin{array}{ccc|cccccccc}
1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & 1 & 1 & 1 \\
1 & \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot & 1 & 1 \\
\cdot & 1 & \cdot & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot & 1 \\
\cdot & 1 & \cdot & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & 1 & 1 & 1 & 1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & 1 & 1 & 1 & 1 & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 & 1 & 1 & 1 & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & 1 & 1 & 1 & 1
\end{array}\right]$$

FIG. A.1. 10×11 *matrix of column assignment problem obtained by reducing the bin packing problem with values* $|U| = 3$, $s(u_0) = 3$, $s(u_1) = 3$, $s(u_2) = 5$, $C = 4$, *and* $P = 2$. *Zero matrix elements are represented by a dot.*

communication load C.
QUESTION: Does there exist an assignment $\phi : \{0, \dots, n-1\} \to \{0, \dots, p-1\}$ of the columns of the matrix such that: (i) $a_{\phi(j),j} = 1$, for $j = 0, \dots, n-1$; (ii) $N_{\text{send}}(i) = \sum_{j:\ \phi(j)=i} (\lambda_j - 1) \le C$, for $i = 0, \dots, p-1$; and (iii) $N_{\text{recv}}(i) = |\{j : 0 \le j < n \wedge a_{ij} = 1 \wedge \phi(j) \ne i\}| \le C$?

The column assignment problem can be related to the *Bin Packing* problem, defined as follows by Garey and Johnson [14].
INSTANCE: A finite set U of items, a positive integer size $s(u)$ for each item $u \in U$, a positive integer bin capacity C, and a positive integer P.
QUESTION: Does there exist a partition of U into disjoint sets $U_0, U_1, \dots, U_{P-1}$ such that the sum of the sizes of the items in each U_i is at most C?

Bin packing is NP-complete in the strong sense, which means that there is no pseudo-polynomial time solution unless P=NP [14].

THEOREM A.1. *The column assignment problem is NP-complete.*

Proof. We will prove the NP-completeness of the column assignment problem by reduction from the bin packing problem. Our reduction is not polynomial, but pseudo-polynomial. The reduction still proves NP-completeness, since the bin packing problem is NP-complete in the strong sense. Observe that for any instance of the bin packing problem, multiplying the size of each item and the bound C by a constant will not change the original problem. Let $c = \lceil \max\{P, |U|\}/u_{min} \rceil$, where u_{min} is the minimum size among items in U. Multiply each size $s(u)$, u_{min}, and C by c. In the rest of the proof, we assume that $s(u)$, u_{min}, and C denote the numbers after this multiplication, so that $u_{min} \ge \max\{P, |U|\}$. Without loss of generality, we assume that $C \ge u_{min}$ and $u_{min} \ge 3$.

Given an instance of the bin packing problem, the column assignment problem will have $p = P + Q$ processors and $n = |U| + Q$ columns, where $Q = \sum_{u \in U} s(u) - |U|(P-1)$. The first P processors correspond to the bins of the bin packing problem, and the remaining Q are auxiliary processors. Similarly, the first $|U|$ columns correspond to items in U and the remaining Q are auxiliary columns.

Let $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ be the $p \times n$ matrix of the column assignment problem after reduction, where the $P \times |U|$ submatrix A_{11} is defined by the bin processors and item columns, and the $Q \times Q$ submatrix A_{22} is defined by the auxiliary processors and auxiliary items. The matrix A is defined as follows and is illustrated by Fig. A.1.

- All elements of A_{11} are 1, which means that each column representing an item can be assigned to any processor representing a bin.
- $A_{12} = 0$, which means that auxiliary columns cannot be assigned to a processor representing a bin.
- We define A_{22} such that each row and each column in A_{22} has C nonzeros. This can be achieved by assigning nonzeros to the C positions on and below the main diagonal in each column (continuing with the first row, if the last row is reached). In other words, $a_{ij} = 1$ in A_{22} for $i = j, (j+1) \mathbf{mod}\, Q, \ldots, (j+C-1) \mathbf{mod}\, Q$. All other elements are 0. We assume without loss of generality that $Q \geq C$. (Otherwise, Q can be suitably enlarged.)
- We define A_{21} such that each row has a single nonzero, and such that the ith column, which represents the ith item u_i, has $s(u_i) - P + 1$ nonzeros. This can be achieved by putting nonzeros in the first $s(u_0) - P + 1$ rows and the first column, the next $s(u_1) - P + 1$ rows and the second column, and so on. Altogether this requires $\sum_{u_i \in U} (s(u_i) - P + 1) = \sum_{u_i \in U} s(u_i) - |U|(P-1) = Q$ rows, which are exactly the auxiliary rows.

We claim that there is a solution to the reduced column assignment problem with no processor loaded more than C, if and only if there is a solution to the bin packing problem. The proof is based on the following observations on the reduced column assignment problem.

1. Each auxiliary column has to be assigned to an auxiliary processor. This will load every auxiliary processor with $C - 1$ sends and $C - 1$ receives for an auxiliary column. No auxiliary processor can obtain two (or more) auxiliary columns, since the number of sends $2C - 2$ would then exceed C. Since each of the remaining columns has at least $u_{min} + 1 \geq 4$ nonzeros, representing at least three sends, they cannot be assigned to an auxiliary processor without exceeding C. Therefore, each column representing an item needs to be assigned to a processor representing a bin.
2. The receiving load of every processor that represents a bin is at most $|U|$. Recall that $|U| \leq C$ because of the initial scaling operation. Every auxiliary processor has a receiving load of C, including one receive operation for the columns representing an item. Whether the communication load of a processor satisfies the bound C is thus solely determined by its sending load.
3. Each column representing an item u has $s(u) + 1$ nonzeros and hence assigns $s(u)$ send operations to its owner.

By the first observation, each item must be assigned to a bin for the communication load to remain below C. By the second observation, distribution of remaining receives does not affect the feasibility of the solution. Only the assignment of sends is important. By the final observation, each column representing an item assigns a number of sends equal to the size of the item to its processor (bin). By construction, any item-column can be assigned to any bin-processor. As a result, the problem reduces to the bin packing problem.

Based on these observations, there will be a solution to the column assignment problem, if and only if there is a solution to the bin packing problem. $\square$

Acknowledgements. We would like to thank the referees for their careful reading and useful comments on the initial version of this paper, which led to significant improvements. We thank Ali Pinar for providing the proof of NP-completeness of our problem, which is given in the appendix. The first author wishes to thank Erik Boman, Ümit Çatalyürek, Karen Devine, Bob Heaphy, Bruce Hendrickson, Alex Pothen, and Tony Skjellum for many fruitful discussions during his visit to the US, April–July 2004.

REFERENCES

[1] R. BARRETT, M. BERRY, T. F. CHAN, J. DEMMEL, J. DONATO, J. DONGARRA, V. EIJKHOUT, R. POZO, C. ROMINE, AND H. VAN DER VORST, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, PA, 1994.

[2] E. BOMAN, K. DEVINE, R. HEAPHY, B. HENDRICKSON, M. HEROUX, AND R. PREIS, *Parallel repartitioning for optimal solver performance*, Technical Report SAND2004-0365, Sandia National Laboratories, Albuquerque, NM, Feb. 2004.

[3] A. E. CALDWELL, A. B. KAHNG, AND I. L. MARKOV, *Improved algorithms for hypergraph bipartitioning*, in Proceedings Asia and South Pacific Design Automation Conference, ACM Press, New York, 2000, pp. 661–666.

[4] Ü. V. ÇATALYÜREK AND C. AYKANAT, *Decomposing irregularly sparse matrices for parallel matrix-vector multiplications*, in Proceedings Third International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 1996), A. Ferreira, J. Rolim, Y. Saad, and T. Yang, eds., vol. 1117 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1996, pp. 75–86.

[5] ———, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Trans. Parallel Distrib. Systems, 10 (1999), pp. 673–693.

[6] ———, *PaToH: A multilevel hypergraph partitioning tool, version 3.0*, technical report, Department of Computer Engineering, Bilkent University, Ankara, Turkey, 1999.

[7] ———, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, in Proceedings Eighth International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001), IEEE Press, Los Alamitos, CA, 2001, p. 118.

[8] ———, *A hypergraph-partitioning approach for coarse-grain decomposition*, in Proceedings Supercomputing 2001, ACM Press, New York, 2001, p. 42.

[9] T. A. DAVIS, *University of Florida sparse matrix collection*, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 1994-2004, online collection, http://www.cise.ufl.edu/research/sparse/matrices.

[10] K. DEVINE, E. BOMAN, R. HEAPHY, B. HENDRICKSON, AND C. VAUGHAN, *Zoltan data management services for parallel dynamic applications*, Computing in Science and Engineering, 4(2) (2002), pp. 90–97.

[11] I. S. DUFF, R. G. GRIMES, AND J. G. LEWIS, *Sparse matrix test problems*, ACM Trans. Math. Software, 15 (1989), pp. 1–14.

[12] ———, *The Rutherford-Boeing sparse matrix collection*, Technical Report TR/PA/97/36, CERFACS, Toulouse, France, Sept. 1997.

[13] C. M. FIDUCCIA AND R. M. MATTHEYSES, *A linear-time heuristic for improving network partitions*, in Proceedings of the 19th IEEE Design Automation Conference, IEEE Press, Los Alamitos, CA, 1982, pp. 175–181.

[14] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.

[15] Y. F. HU, K. C. F. MAGUIRE, AND R. J. BLAKE, *A multilevel unsymmetric matrix ordering algorithm for parallel process simulation*, Computers and Chemical Engineering, 23 (2000), pp. 1631–1647.

[16] G. KARYPIS AND V. KUMAR, *Multilevel k-way hypergraph partitioning*, in Proceedings 36th ACM/IEEE Conference on Design Automation, ACM Press, New York, 1999, pp. 343–348.

[17] A. PINAR AND B. HENDRICKSON, *Graph partitioning for complex objectives*, in Proceedings Eighth International Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001), IEEE Press, Los Alamitos, CA, 2001, p. 121.

[18] ———, *Exploiting flexibly assignable work to improve load balance*, in Proceedings 14th Annual ACM Symposium on Parallel Algorithms and Architectures, 2002, pp. 155–163.

[19] A. TRIFUNOVIC AND W. J. KNOTTENBELT, *A parallel algorithm for multilevel k-way hypergraph partitioning*, in Proceedings Third International Symposium on Parallel and Distributed Computing, Cork, Ireland, July 2004.

[20] B. UÇAR AND C. AYKANAT, *Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies*, SIAM J. Sci. Comput., 25 (2004), pp. 1837–1859.

[21] L. G. VALIANT, *A bridging model for parallel computation*, Communications of the ACM, 33(8) (1990), pp. 103–111.

[22] 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.

[23] B. VASTENHOUW AND R. H. BISSELING, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Rev., 47 (2005), pp. 67–95.