Efficient Matching for Column Intersection Graphs

B. O. FAGGINGER AUER and R. H. BISSELING, Utrecht University

To improve the quality and efficiency of hypergraph-based matrix partitioners, we investigate high-quality matchings in column intersection graphs of large sparse binary matrices. We show that such algorithms have a natural decomposition in an integer-weighted graph-matching function and a neighbor-finding function and study the performance of 16 combinations of these functions. We improve upon the original matching algorithm of the Mondriaan matrix partitioner: by using PGA', we improve the average matching quality from 95.3% to 97.4% of the optimum value; by using our new neighbor-finding heuristic, we obtain comparable quality and speedups of up to a factor of 19.6.

Categories and Subject Descriptors: G.2.2 [**Discrete Mathematics**]: Graph Theory—*Hypergraphs; Matchings and factors; Graph algorithms; Approximation algorithms*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Column intersection graphs, heuristic algorithms, weighted graph matching

ACM Reference Format:
B. O. Fagginger Auer and R. H. Bisseling. 2014. Efficient matching for column intersection graphs. ACM J. Exp. Algor. 19, 1, Article 3 (May 2014), 22 pages.
DOI: http://dx.doi.org/10.1145/2616587

1. INTRODUCTION

In this article, we investigate efficient matching algorithms that can be used for high-quality coarsening of hypergraphs to improve the performance and quality of matrix partitioners such as Mondriaan [Vastenhouw and Bisseling 2005], PaToH [Çatalyürek and Aykanat 1999], and Zoltan [Devine et al. 2006]. Internally, these matrix partitioners represent the matrix that is to be partitioned as a hypergraph. To partition this hypergraph, they use a multilevel coarsening strategy based on merging matched vertices, together with refinement during uncoarsening (e.g., [Kernighan and Lin 1970; Fiduccia and Mattheyses 1982]). Generating a matching within the hypergraph for coarsening is a costly operation[1] and the quality of the matching has a direct influence on the quality of the partitioning, which is why we are interested in investigating this problem.

[1]When partitioning the sparse matrix `rhpentium` from our test set into two parts with imbalance 0.03 for the row-net hypergraph model, the Mondriaan 3.11 hypergraph bipartitioner can spend as much as 91% of its time on generating matchings for the coarsening; the remaining 9% is then spent on coarsening and refinement.

Authors' addresses: R. H. Bisseling, Mathematics Institute, Utrecht University, PO Box 80010, 3508 TA Utrecht, the Netherlands; email: r.h.bisseling@uu.nl.
(Current address) B. O. Fagginger Auer, Development and Engineering Department, ASML, PO Box 324, 5500 AH Veldhoven, the Netherlands; email: basfaggingerauer@gmail.com.

DOI: http://dx.doi.org/10.1145/2616587

The presented algorithms can be viewed as methods to efficiently generate heavy graph matchings if the graph's weighted adjacency matrix can be factorized as $A^T A$ for a sparse binary matrix A. Even though maximum-weight graph matching can be solved in polynomial time [Edmonds 1965a, 1965b], the size of the matrices that need to be partitioned in practice precludes the use of exact solution methods.

An *(undirected) weighted graph* G is a triplet (V, E, ω) with *vertices* V, *edges* E (every $e \in E$ is of the form $e = \{u, v\}$ for $u, v \in V$ with $u \neq v$), and *edge weights* $\omega(e) > 0$ for all $e \in E$. We consider only *integer edge weights*, $\omega : E \to \mathbf{Z}_{>0}$. The symmetric $|V| \times |V|$ matrix B with entries

$$B_{uv} = \begin{cases} \omega(\{u, v\}) & \{u, v\} \in E \\ 0 & \{u, v\} \notin E \end{cases}$$

is the *weighted adjacency matrix of* G.

A *matching of* G is a set $M \subseteq E$ of edges that are mutually disjoint (i.e., for all $d, e \in M$, $d \cap e \neq \emptyset \to d = e$). The *weight of* M is defined as the sum of the weights of all edges in the matching:

$$\Omega_G(M) := \sum_{e \in M} \omega(e).$$

PROBLEM 1.1 (WGM). *Let $G = (V, E, \omega)$ be a weighted graph with integer edge weights. We call finding a matching M of G with maximum weight $\Omega_G(M)$ the* weighted graph-matching problem.

We say that a graph-matching algorithm is an *α-approximation algorithm*, for $\alpha \in [0, 1]$, if for every graph G the algorithm generates a matching M such that $\Omega_G(M) \geq \alpha\, \Omega_G(M^*)$, where M^* is a matching of G with the largest possible weight. We call an algorithm *optimal* or *exact* if it is a 1-approximation algorithm. There is a large number of exact and approximation algorithms available to solve WGM; for an overview, we refer the reader to Duan et al. [2011, Tables I–IV].

We now turn to the primary problem that we want to solve: $A^T A$-matching. Let $A \in \{0, 1\}^{m \times n}$ be a binary matrix (i.e., with only zeros and ones as entries) with m rows and n columns. We consider only binary matrices because we are interested in partitioning the matrix nonzeros, which is only influenced by the row-column position of the nonzeros and not their numerical values. There is a direct correspondence between binary $m \times n$ matrices A and hypergraphs with n vertices and m nets (hyperedges), where each column of A corresponds to a vertex in the hypergraph, and each row of A corresponds to a net containing all vertices of which the corresponding column possesses a nonzero in that row. This is the row-net hypergraph model [Çatalyürek and Aykanat 1999] and we view A both as a matrix and as a hypergraph with respect to this model.

Denote the columns of A by a_j for $1 \leq j \leq n$. We call two columns a_j, a_k ($j \neq k$) *neighbors* if $\langle a_j, a_k \rangle > 0$, where for arbitrary vectors $x, y \in \mathbf{R}^n$, we denote their *inner product* by $\langle x, y \rangle := \sum_{i=1}^{n} x_i\, y_i$. A *column matching of* A is a collection M of unordered pairs of different column indices that are mutually disjoint and satisfy $\langle a_j, a_k \rangle > 0$ for all $\{j, k\} \in M$. The *weight of* M is defined as the sum of these inner products:

$$\Omega_A(M) := \sum_{\{j,k\} \in M} \langle a_j, a_k \rangle. \tag{1}$$

Note that we have the following upper bound for $\Omega_A(M)$:

$$\Omega_A(M) \leq \left\lfloor \frac{\mathrm{nz}(A)}{2} \right\rfloor, \tag{2}$$

where $\mathrm{nz}(A)$ is the number of nonzero entries in A.

PROBLEM 1.2 (ATAM). *Let $A \in \{0,1\}^{m\times n}$ be a binary matrix. We call finding a column matching M of A with maximum weight $\Omega_A(M)$ the A^TA-*matching problem.

The inner products from (1) are the origin of the name "A^TA-matching," since

$$A^TA = \begin{pmatrix} — & a_1^T & — \\ & \vdots & \\ — & a_n^T & — \end{pmatrix} \begin{pmatrix} | & & | \\ a_1 & \cdots & a_n \\ | & & | \end{pmatrix} = \begin{pmatrix} \langle a_1, a_1\rangle & \cdots & \langle a_1, a_n\rangle \\ \vdots & \ddots & \vdots \\ \langle a_n, a_1\rangle & \cdots & \langle a_n, a_n\rangle \end{pmatrix},$$

so solving ATAM amounts to solving WGM in the graph with n vertices that has weighted adjacency matrix A^TA with zeros along the diagonal. Such a graph is called the *column intersection graph* of the matrix A [Gebremedhin et al. 2005].

Solving ATAM is essential for the coarsening step of matrix partitioning, which we will illustrate with a small example. The matrix partitioning problem for parallel sparse matrix–vector multiplication is finding the distribution of the nonzeros of a sparse matrix over a fixed number of processors such that, within a specified imbalance, each processor is assigned the same number of nonzeros, and the communication between different processors is minimized during the parallel sparse matrix–vector multiplication. This can be formulated as a hypergraph partitioning problem, where the quality of a partitioning is measured by the $(\lambda - 1)$-metric. Such a problem is NP hard and is therefore usually solved using multilevel heuristics.

For our example, we will assume that the matrix we wish to partition has already been converted to a hypergraph, represented by the binary matrix A via the row-net model.

PROBLEM 1.3 (HP). *Let $A \in \{0,1\}^{m\times n}$ be a matrix with column weights $w_j \geq 0$ for $1 \leq j \leq n$, $k \in \mathbf{Z}_{\geq 2}$ the desired number of parts, and $\epsilon \geq 0$ the maximum permitted imbalance. The* hypergraph partitioning problem *is finding a partitioning of A's columns, that is, a surjective map $\Pi : \{1,\ldots,n\} \to \{1,\ldots,k\}$, such that all parts are ϵ-balanced:*

$$\sum_{\substack{1\leq j\leq n \\ \Pi(j)=l}} w_j \leq \frac{1+\epsilon}{k} \sum_{1\leq j\leq n} w_j, \qquad (1 \leq l \leq k),$$

and the communication volume *as measured by the $(\lambda - 1)$-metric is minimal:*

$$\mathrm{CV}_A(\Pi) := \sum_{i=1}^{m} (|\Pi(\{1 \leq j \leq n \mid a_{ij} \neq 0\})| - 1). \tag{3}$$

The column weight w_j in the general hypergraph partitioning problem can in principle take any nonnegative value, not necessarily related to the column a_j itself, but in many applications $w_j = \mathrm{nz}(a_j)$. The communication volume is obtained by summing, for each row, the number of different parts to which nonzeros from this row belong minus one. This measures the number of data words that need to be communicated during a parallel sparse matrix–vector multiplication. We would like to refer the reader to Çatalyürek and Aykanat [1999] and Bisseling [2004] for an in-depth treatment of this problem.

Because HP is NP hard, we use a multilevel heuristic, where we reduce the number of vertices of the hypergraph to obtain a problem that is easier to solve. During the coarsening phase, we want to coarsen a matrix A to a matrix A' by merging columns, based on a column matching M of A. We merge every pair of columns a_j and a_k of A for

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}, \qquad A' = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}, \qquad A'' = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}.$$

Fig. 1. Example of a matrix $A \in \{0,1\}^{5\times 4}$ coarsened to A' or A'' by merging matched columns in $M' = \{\{1,2\},\{3,4\}\}$ and $M'' = \{\{1,4\},\{2,3\}\}$, respectively.

which $\{j,k\} \in M$ to a single column a'_l of A' with weight $w'_l = w_j + w_k$:

$$\begin{array}{cc} a_j & a_k \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{array} \quad \rightarrow \quad \begin{array}{c} a'_l \\ 1 \\ 1 \\ 1 \\ 0 \end{array}.$$

This yields a matrix A' with fewer columns and nonzeros than A.

During uncoarsening, for any partitioning Π' of the coarser matrix A', we construct a partitioning Π of A by setting $\Pi(j)$ equal to $\Pi'(l)$, where l is the index of the column in A' into which j has been merged. For such a choice of Π, we have that

$$\mathrm{CV}_A(\Pi) = \mathrm{CV}_{A'}(\Pi')$$

by construction. Furthermore, if Π' is ϵ-balanced, then so is Π. After copying Π, this partitioning can be further refined by using, for example, the Kernighan–Lin heuristic [Kernighan and Lin 1970]. Coarsening recursively yields a multilevel heuristic to solve the hypergraph partitioning problem. To obtain a high-quality Π, it is therefore important to coarsen A to A' such that the minimal $\mathrm{CV}_{A'}(\Pi')$ for a balanced partitioning[2] Π' is as small as possible (this minimum depends on A' and therefore on the coarsening we perform).

Consider, for example, the matrix A in Figure 1, coarsened to A' or A'' via the matchings M' and M''. The best (and only balanced) bipartitionings Π' and Π'' of A' and A'' map column 1 to one part and column 2 to the other. Note, however, that $\mathrm{CV}_{A'}(\Pi') = 4$, while $\mathrm{CV}_{A''}(\Pi'') = 1$, which is much lower. This is due to the higher weight of M'' compared to M': $\Omega_A(M') = 2 < 5 = \Omega_A(M'')$. Because we merge columns that share many nonzeros together in M'', we ensure a lower communication volume.

In general, the number of nonzeros we eliminate by merging a_j and a_k is precisely equal to the number of nonzeros that both columns share and hence is equal to the inner product $\langle a_j, a_k \rangle$. Therefore, the number of nonzeros of A' satisfies

$$\mathrm{nz}(A') = \mathrm{nz}(A) - \Omega_A(M). \tag{4}$$

Every eliminated nonzero is one less possibility of a nonzero causing communication in parallel sparse matrix–vector multiplication, leading to higher-quality partitionings of the coarsened matrix. Furthermore, elimination of all but one nonzero in a row allows for deleting that row from the partitioning problem, because it cannot cause communication.

Suppose every row of A contains at least one nonzero. If we would coarsen A until only two columns remain, keeping rows with a single nonzero in the matrix, then we

[2]The Mondriaan partitioner, and also other partitioners, prohibit columns j with very large weights w_j from merging with any other column to ensure that balanced partitionings exist for the coarsened matrix.

reach the final $m \times 2$ matrix $\hat{A}$. We can partition this matrix trivially into $k = 2$ parts by $\Pi(1) = 1$ and $\Pi(2) = 2$, and then we have the relation

$$\mathrm{CV}_{\hat{A}}(\Pi) = \mathrm{nz}(\hat{A}) - m \tag{5}$$

between the communication volume and the number of nonzeros. In practice, we stop coarsening earlier to enable a better tradeoff between load balance and communication. Thus, the ultimate goal is to minimize the number of nonzeros, but we perform only a limited number of coarsening steps, each time greedily minimizing the number of nonzeros for the next coarsening via Equation (4).

Reducing the number of nonzeros decreases the expected communication volume for block partitionings in random sparse matrices as well. Consider a random sparse matrix $A \in \{0,1\}^{m \times n}$, where $a_{ij} = 1$ with a fixed probability $0 < d < 1$, and $a_{ij} = 0$ otherwise. Let $k \in \mathbf{Z}_{\geq 2}$ be the desired number of parts, which we assume to divide n. We create a block partitioning Π of the columns of A into k parts by assigning columns $1, \ldots, \frac{n}{k}$ to part 1, columns $\frac{n}{k} + 1, \ldots, 2\frac{n}{k}$ to part 2, and so on. This partitioning is expected to be balanced. The probability that there exists a nonzero $a_{ij} \neq 0$ in row i with $1 \leq j \leq \frac{n}{k}$ equals $1 - (1-d)^{\frac{n}{k}}$, independent of i. Therefore, in an analysis similar to Bisseling [2004, Section 4.7], the expected number of different parts that contain nonzeros in a row i equals $k(1 - (1-d)^{\frac{n}{k}})$, which means that, via Equation (3),

$$\mathrm{E}(\mathrm{CV}_A(\Pi)) = m(k\left(1 - (1-d)^{\frac{n}{k}}\right) - 1) = m\left(k\left(1 - \left(1 - \frac{\mathrm{E}(\mathrm{nz}(A))}{mn}\right)^{\frac{n}{k}}\right) - 1\right),$$

which increases strictly as a function of $\mathrm{E}(\mathrm{nz}(A))$. For block partitionings of random sparse matrices, the expected communication volume can therefore be minimized by minimizing $\mathrm{nz}(A)$, which makes solving ATAM a good way to decrease the communication volume of these partitionings, via Equation (4).

In summary, we can view the use of matching for coarsening as a relatively cheap heuristic that reduces the number of nonzeros in a row at every level of the coarsening, making it easier to keep the nonzeros of a row together in the initial partitioning at the coarsest level, or even guaranteeing this in case all nonzeros of the row were merged into one. This makes solving ATAM relevant for hypergraph partitioning.

An alternative, more aggressive approach to coarsening is agglomerative clustering, where more than two matrix columns are allowed to be merged into a cluster. In Çatalyürek et al. [2012], matching-based and agglomerative algorithms for clustering vertices in the coarsening phase of hypergraph partitioning are compared. The authors also present shared-memory parallel algorithms aimed at multicore architectures, parallelizing the coarsening phase of the sequential package PaToH [Çatalyürek and Aykanat 1999]. The authors view the matching as on-the-fly matching of an implicitly formed graph with adjacency matrix $A^T A - \mathrm{diag}(A^T A)$. They found that using a $\frac{1}{2}$-approximation matching algorithm on average gave a better quality than using greedy matching (without a quality guarantee), but that this comes at the expense of a considerable increase in computation time. Their numerical experiments also show that matching-based coarsening algorithms are faster than agglomerative algorithms, but that agglomerative algorithms catch up during initial partitioning (with fewer vertices) and uncoarsening (with fewer levels). The authors observed a lower communication volume for agglomerative algorithms. Our approach to coarsening is exclusively based on matching, and algorithms of a different nature such as agglomerative clustering fall outside the scope of the present study.

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

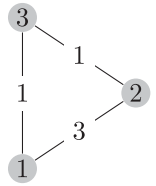


Fig. 2. Instances A and G of problems ATAM and WGM that have the same optimal solution, $M = \{1, 2\}$ with weight 3. G is the column intersection graph $G(A)$ of A.

2. THE RELATIONSHIP BETWEEN WEIGHTED GRAPH MATCHING AND A^TA-MATCHING

We now investigate the correspondence between the problems WGM and ATAM. In Figure 2, we see an example of instances of WGM and ATAM with the same solution.

Let $A \in \{0, 1\}^{m \times n}$ be an instance of ATAM. Construct the graph $G = G(A)$ with vertices $V = \{1, \ldots, n\}$ and edges $\{j, k\} \in E$ for all $1 \leq j < k \leq n$ with $\langle a_j, a_k \rangle > 0$. We define the weight $\omega(\{j, k\})$ to be equal to $\langle a_j, a_k \rangle$. Then, G's weighted adjacency matrix is given by $A^TA - \text{diag}(A^TA)$ so that G is the column intersection graph of A. Let M be a matching of G, and then M is also a column matching of A. Furthermore, by choice of ω, $\Omega_A(M) = \Omega_G(M)$.

Let $G = (V, E, \omega)$ be an instance of WGM. Enumerate $V = \{v_1, \ldots, v_n\}$ and let $m = \sum_{e \in E} \omega(e)$. We construct $A = A(G) \in \{0, 1\}^{m \times n}$ by adding $\omega(\{v_j, v_k\})$ identical rows to A that have nonzeros only in columns j and k, for all edges $\{v_j, v_k\} \in E$. This ensures that $(A^TA)_{jk} = \omega(\{v_j, v_k\})$. Let M be a column matching of A and define the matching M' of G by letting $\{v_j, v_k\} \in M'$ if and only if $\{j, k\} \in M$. By construction of A,

$$\begin{aligned} \Omega_A(M) = \sum_{\{j,k\} \in M} \langle a_j, a_k \rangle = \sum_{i=1}^{m} \sum_{\{j,k\} \in M} a_{ij}\, a_{ik} &= \sum_{\{v_p, v_q\} \in E} \omega(\{v_p, v_q\}) \sum_{\{j,k\} \in M} \delta_{\{j,k\}=\{p,q\}} \\ &= \sum_{\{v_p, v_q\} \in E} \omega(\{v_p, v_q\})\, \delta_{\{v_p, v_q\} \in M'} = \Omega_G(M'). \end{aligned}$$

Here, $\delta_{\{j,k\}=\{p,q\}}$ equals 1 if $\{j, k\} = \{p, q\}$ and 0 otherwise. This permits us to use an algorithm that is capable of solving WGM to solve ATAM and vice versa, while preserving the objective function. It furthermore shows us that we can encounter *any* combination of integer weights in ATAM problems. We have proven the following lemma.

LEMMA 2.1. *Let $G = (V, E, \omega)$ be an integer-weighted graph and B its weighted adjacency matrix. Then, there exists an $A \in \{0, 1\}^{m \times n}$ with $m = \sum_{e \in E} \omega(e)$ and $n = |V|$, such that*

$$B_{ij} = (A^TA)_{ij} \qquad \textit{for all } 1 \leq i < j \leq n.$$

The maps $A \mapsto G(A)$ and $G \mapsto A(G)$ are not inverses of each other. For example, consider the matrices

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix},$$

which both map to the same column intersection graph. Thus, $A \mapsto G(A)$ is not a bijection. In particular, consider a sequence of graphs with n vertices and a single edge with weight n^n: $G_n = (\{1, \ldots, n\}, \{\{1, 2\}\}, \omega_n)$ with $\omega_n(\{1, 2\}) = n^n$. Then, translating this to ATAM gives us matrices $A(G_n)$ with n columns and n^n rows, requiring superexponential storage.

3. BUILDING BLOCKS OF THE A^TA-MATCHING ALGORITHM

As shown by Edmonds [1965a, 1965b], by using linear programming duality, WGM can be solved optimally in polynomial time. Hence, ATAM can also be solved optimally in polynomial time: we can calculate the symmetric matrix A^TA in $\mathcal{O}(n^2\,m)$ time and run Edmonds' algorithm on the weighted graph represented by this matrix. With Gabow and Tarjan's version of Edmonds' algorithm for bounded integer edge weights [Gabow and Tarjan 1991, Theorem 10.1], an optimal matching can be found in $\mathcal{O}(\sqrt{n\,\alpha(n^2,n)\,\log(n)}\,n^2\,\log(n\,m))$ time[3] and $\mathcal{O}(n^2)$ space.

However, this strategy is infeasible in practice: for current sparse problems, values as $m, n \sim 10^6$ are common [Davis and Hu 2011]. Therefore, even storing the matrix A^TA is problematic (it is often much less sparse than the matrix A), as is keeping track of dual variables associated with all of $G(A)$'s edges to solve ATAM optimally. In fact, should A possess a single dense row (almost-dense rows occur often in practice, see, e.g., the matrix `rhpentium`), then all columns of A have a mutual inner product of at least one. Viewed as a WGM problem, this means that we need to perform matching on a weighted clique graph, making exact matching algorithms very slow.

Apart from this problem, Lemma 2.1 shows us that A^TA-factorization does not impose any constraints on the integer edge weights that we can encounter, so we cannot exploit the fact that we can write G's weighted adjacency matrix as A^TA if we were to view this purely as a graph-matching algorithm. This motivates us to separate the ATAM algorithm into two building blocks:

(1) A known graph-matching algorithm to solve WGM (e.g., one of the algorithms listed in Duan et al. [2011]), where we incrementally build the graph $G(A)$ for which we generate the matching as the algorithm progresses. We refer to this algorithm as the *graph-matching function* of the ATAM algorithm.
(2) An algorithm that finds unmatched neighbors of a column in A and that calculates or approximates the inner products with these neighbors. We refer to this algorithm as the *neighbor-finding function* of the ATAM algorithm.

Due to Lemma 2.1, it is in the second part that we expect to be able to improve the ATAM algorithm's performance.

4. MONDRIAAN'S A^TA-MATCHING ALGORITHM

The matching algorithm used in the Mondriaan software package (a similar algorithm is also available in PaToH and Zoltan) is described by Algorithm 1. It considers each column j of a given matrix $A \in \{0,1\}^{m\times n}$ one by one and, should j not yet be matched, finds a neighbor column $k \neq j$ that is not yet matched, such that the inner product $\langle a_j, a_k\rangle$ is as large as possible, and then matches j to k.

For convenience, we define for row $1 \leq i \leq m$ in our matrix A the set of *nonzero column indices* as

$$J_i := \{1 \leq j \leq n \mid a_{ij} \neq 0\} \tag{6}$$

and for $1 \leq j \leq n$ the set of *nonzero row indices* in column j as

$$I_j := \{1 \leq i \leq m \mid a_{ij} \neq 0\}. \tag{7}$$

Let $\mathrm{nz} = \mathrm{nz}(A)$ denote the number of nonzeros of A, and nz_r (nz_c) the maximum number of nonzeros in all rows (columns) of A. Furthermore, instead of considering a matching as a collection M of pairs $\{j,k\}$ of column indices, we describe M in terms of a map $\mu : \{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$ such that $\{j,k\} \in M$ if and only if $j \neq k$, $\mu[j] = k$, and

[3]Here, $\alpha(m,n)$ denotes the inverse Ackermann function.

ALGORITHM 1: ATAM algorithm used in Mondriaan 3.11.

Data: Matrix $A \in \{0,1\}^{m \times n}$.
Result: Column matching μ of A.

```
Initialize S[j] ← 0 for all 1 ≤ j ≤ n;
for j ← 1 to n do
    if μ[j] = j then
        V ← ∅;
        foreach i ∈ I_j do
            foreach k ∈ J_i do
                if μ[k] = k and k ≠ j then
                    if S[k] = 0 then V ← V ∪ {k};
                    S[k] ← S[k] + 1;
        k ← argmax_{k∈V} S[k];
        foreach l ∈ V do S[l] ← 0;
        μ[j] ← k;
        μ[k] ← j;
        Remove columns j and k from A;
```

$\mu[k] = j$. We set $\mu[j] = j$ to indicate that j is *unmatched* and always start with an empty matching where all vertices are unmatched.

The best unmatched neighbor of an unmatched column j of A is determined as follows. We start with an array S containing the inner product of column j with all other columns k of A, initialized to 0. Then, we traverse all rows $i \in I_j$, where for each column $k \in J_i$, we increment the value $S[k]$ in the inner product array. If we encounter a new column for the first time, it is added to a visited list V (as not all columns are necessarily neighbors of j). This will result in the inner product array containing the values $\langle a_j, a_k \rangle$ for all $1 \leq k \leq n$ and a visited list containing all column indices k for which $\langle a_j, a_k \rangle > 0$. Then, from all visited columns, we select the one having the highest inner product with j and match j to this column.

Algorithm 1 requires $\mathcal{O}(n \operatorname{nz}_c \operatorname{nz}_r)$ time and $\mathcal{O}(n)$ storage. It is possible for the matrix A to possess (nearly) dense rows, in which case Algorithm 1 takes a lot of time because each unmatched column j considers all other unmatched columns k as matching candidates.

A second issue is that Algorithm 1 provides solutions to ATAM that can be arbitrarily bad. Consider the matrix $A \in \{0,1\}^{2(k-1) \times 3}$ with $a_{i1} = 1$ if $i \leq k$ and $a_{ij} = 1$ for $j = 2, 3$ if $i \geq k$, and 0 otherwise. For $k = 4$, this gives

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}. \tag{8}$$

Note that the columns of A are ordered by decreasing number of nonzeros (which is beneficial; see Section 7). Still, Algorithm 1 yields a matching $\{\{1, 2\}\}$ with weight 1, instead of the optimal solution $\{\{2, 3\}\}$ with weight $k - 1$. Therefore, Algorithm 1 is not a $\frac{1}{k}$-approximation algorithm for ATAM, for every $k \geq 1$.

5. GRAPH-MATCHING FUNCTIONS

To improve Algorithm 1, we will separate it into the graph-matching and neighbor-finding functions mentioned before. The graph-matching function of Algorithm 1 visits

the vertices of $G(A)$ in order, adding the heaviest edge originating from the current vertex to the matching; we call this function `Greedy` (Algorithm 2). We consider three other graph-matching algorithms, apart from `Greedy`, to improve solution quality.

ALGORITHM 2: `Greedy` matching algorithm as used by Algorithm 1.

Data: Matrix $A \in \{0,1\}^{m \times n}$ and a function `FindHeavyUnmatchedNeighbors()`.
Result: Column matching μ of A.

Initialize $S[j] \leftarrow 0$ for all $1 \leq j \leq n$;
for $j \leftarrow 1$ **to** n **do**
  if $\mu[j] = j$ **then**
    $V \leftarrow \emptyset$;
    `FindHeavyUnmatchedNeighbors`(j, μ, V, S);
    $k \leftarrow \text{argmax}_{k \in V} S[k]$;
    foreach $l \in V$ **do** $S[l] \leftarrow 0$;
    if $k \neq j$ **then**
      $\mu[j] \leftarrow k$;
      $\mu[k] \leftarrow j$;
      Remove columns j and k from A;

The neighbor-finding function of Algorithm 1 is described by `Mondriaan`, Algorithm 3. We denote this function by `FindHeavyUnmatchedNeighbors` and consider three other implementations, apart from `Mondriaan`, to improve performance.

In general, the function `FindHeavyUnmatchedNeighbors`(j, μ, V, S) determines a collection of columns $V \subseteq \{1, \ldots, n\} \setminus \{j\}$ that are heavy (i.e., having large inner product with a_j) unmatched neighbors of j. It also calculates a collection of scores $S : \{1, \ldots, n\} \to \mathbf{Z}$ such that $S[k] \approx \langle a_j, a_k \rangle$ for all $k \in V$.

ALGORITHM 3: `Mondriaan` implementation of `FindHeavyUnmatchedNeighbors`(j, μ, V, S) (cf. Algorithm 1). Note that the calculated scores satisfy $S[k] = \langle a_j, a_k \rangle$ for $k \in V$.

Data: Matrix $A \in \{0,1\}^{m \times n}$, column index j, matching μ, empty set of neighbors V, and an array of scores S, set to zero.
Result: A set of unmatched neighbors V of j, together with their scores S.

foreach $i \in I_j$ **do**
  foreach $k \in J_i$ **do**
    if $\mu[k] = k$ ***and*** $k \neq j$ **then**
      if $S[k] = 0$ **then** $V \leftarrow V \cup \{k\}$;
      $S[k] \leftarrow S[k] + 1$;

We indicate which combination of graph-matching and neighbor-finding algorithms we are using to solve ATAM by the notation `Matching-Neighbor_finding`. Mondriaan 3.11's ATAM algorithm, Algorithm 1, is thus described by `Greedy-Mondriaan`.

The quality of the matching can be improved by replacing `Greedy` by an algorithm that has a matching-quality guarantee. For this article, we focus on three such algorithms, which are all compatible with large matrices A in the sense that we do not have to keep track of variables associated with all edges of the graph $G(A)$.

The first of these algorithms is the Path Growing Algorithm (PGA), a $\frac{1}{2}$-approximation algorithm introduced in Drake and Hougardy [2003c] and further refined in Drake and Hougardy [2003b] to PGA'. The second $\frac{1}{2}$-approximation algorithm is the Global Paths Algorithm (GPA) introduced in Maue and Sanders [2007] and the

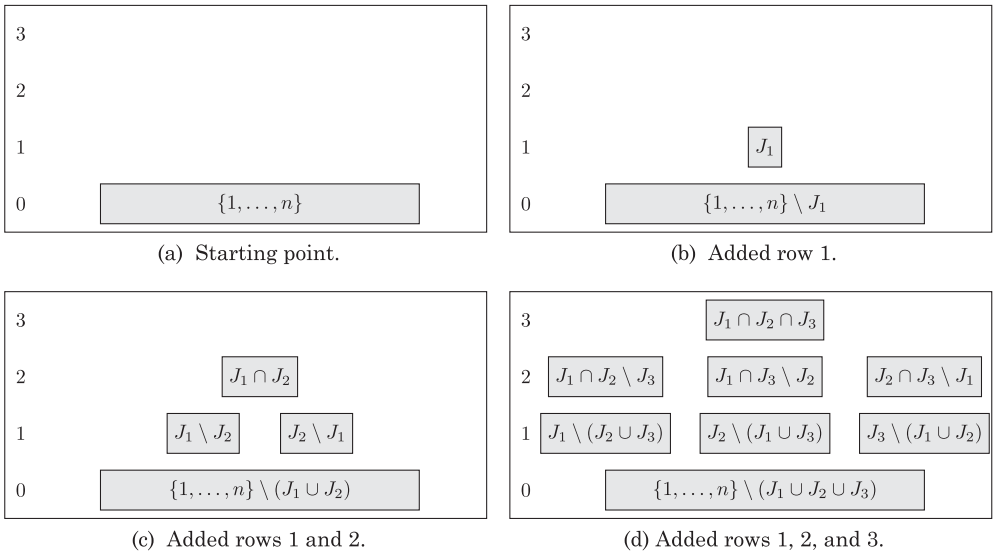
third algorithm is the Random Order Augmentation Matching Algorithm (ROMA) from Maue and Sanders [2007], which is based on the $(\frac{2}{3} - \epsilon)$-approximation algorithm from Pettie and Sanders [2004], which in turn was based on Drake and Hougardy [2003a].

The usual assumption for WGM that we have direct access to all edges or vertex neighbors and all edge weights does not hold for ATAM, which causes problems for matching algorithms that rely on this assumption. For example, Preis' $\frac{1}{2}$-approximation algorithm based on locally dominant edges [Preis 1999] is not suitable for ATAM, because it is necessary to maintain flags for all edges in $G(A)$.

Let $G = (V, E, \omega)$ be the column intersection graph $G(A)$ for a matrix A. The PGA' algorithm works by constructing a path $P \subseteq G$ as follows. Initially, P consists of a single vertex $v_1 \in V$. For the last vertex v_i added to P we find the edge $\{v_i, v_{i+1}\} \in E$ such that v_{i+1} is unmatched and not in P and $\omega(\{v_i, v_{i+1}\})$ is maximal, and extend P along this edge. This is done until the path P cannot be extended further. The original PGA algorithm would then add either all odd-numbered or all even-numbered edges in the path P to M, whichever has the highest total weight. The PGA' algorithm adds a maximum matching on P (constructed using dynamic programming) to M, which is what we do as well. Such a path is constructed for each unmatched vertex in G. For our implementation, we find the heaviest edge originating from a vertex using `FindHeavyUnmatchedNeighbors()`, flagging vertices that are already in the path P using μ. We furthermore choose the starting vertices v_1 in order of decreasing number of nonzeros in the corresponding columns of the matrix A; see Section 7. We denote this algorithm by `PGA`.

The GPA algorithm constructs disjoint paths and even cycles by adding all edges $e \in E$ of the graph in order of decreasing $\omega(e)$, and it uses each edge to either start a path, lengthen a path, or close an existing path to an even cycle (in all other cases the edge is discarded). Then maximum matchings are generated for each path and cycle using dynamic programming. Because it is infeasible for large matrices A to order all edges in $G(A)$ by weight, we look at columns $j = 1, 2, \ldots, n$ of A and select the neighbor k of j with the highest score determined by `FindHeavyUnmatchedNeighbors()` for which the edge $\{k, j\}$ can be added to the paths constructed thus far by the algorithm. This has an adverse effect on the quality of GPA's matchings. As done in Maue and Sanders [2007], we perform three GPA rounds and afterward sequentially match all unmatched columns to obtain a maximal matching. We denote this algorithm by `GPA`. In Holtgrewe et al. [2010], it is shown that using GPA yields better-quality graph partitionings than local greedy matching algorithms that do not offer any quality guarantees.

The ROMA algorithm visits all vertices v in V in a random order and finds the highest-gain 2-augmentation centered at v. If this 2-augmentation has positive gain, it is used to increase the weight of the matching. An *augmentation* with respect to a matching M of G is an alternating (successive edges belong to M and $E \setminus M$) path or cycle $P \subseteq E$ such that $(M \setminus P) \cup (P \setminus M)$ is a matching, and its gain is defined as $\Omega_G(P \setminus M) - \Omega_G(P \cap M)$. A 2-*augmentation* centered at a vertex v is an augmentation containing at most two edges in $E \setminus M$, which are incident either to v or to the vertex matched to v. If no 2-augmentations with positive gain exist for a matching M, then that matching is guaranteed [Drake and Hougardy 2003b, Theorem 1] to have weight at least $\frac{2}{3}$ of the maximum possible. We implemented the algorithm as indicated in Pettie and Sanders [2004], using the columns returned by `FindHeavyUnmatchedNeighbors()` as neighborhoods in which we search for 2-augmentations. The gains are calculated using the scores as determined by `FindHeavyUnmatchedNeighbors()` and may therefore be approximate. We use four ROMA rounds and start with a matching generated by `PGA`. We denote this algorithm by `ROMA`.



(a) Starting point. (b) Added row 1.

(c) Added rows 1 and 2. (d) Added rows 1, 2, and 3.

Fig. 3. Illustration of the stairway heuristic where $\{1,\ldots,n\}$ is the set of all columns of $A \in \{0,1\}^{m\times n}$ and $J_i = \{1 \le j \le n \mid a_{ij} \ne 0\}$ for $1 \le i \le m$. The numbers 0, 1, 2, and 3 on the left are lower bounds on the inner product between all pairs of columns in the same range on the right.

6. NEIGHBOR-FINDING FUNCTIONS

Matrices that possess dense rows slow down `Mondriaan` significantly. To be able to effectively deal with such matrices, we consider the neighbor-finding problem from the perspective of the matrix rows, instead of the columns. We partition the set of all columns $\{1,\ldots,n\}$ into *disjoint ranges* that contain at least a fixed number of $N \ge 2$ columns, such that all pairs of columns in a specific range are guaranteed to have a certain minimum inner product. Thus, the inner product of a range is a lower bound on all the mutual inner products of matrix columns in that range. To visualize this process, we can think of these ranges as sorted by their inner product, forming a stairway. This provides us with a fast `Stairway` heuristic for `FindHeavyUnmatchedNeighbors()`, which is outlined in Algorithm 4. We take a minimum range size of $N = 2$ such that each column has at least one other matching candidate (odd-length ranges, however, may occur).

ALGORITHM 4: `Stairway` implementation of `FindHeavyUnmatchedNeighbors(`j, μ, V, S`)`. Assumes that the stairway ranges for A have already been constructed (Algorithm 5) for $N = 2$.

Data: Matrix $A \in \{0,1\}^{m\times n}$, column index j, matching μ, empty set of neighbors V, and an array of scores S, set to zero.
Result: A set of unmatched neighbors V of j, together with their scores S.

$V \leftarrow$ { unmatched columns $k \ne j$ in the same range as j };
foreach $k \in V$ **do**
    $S[k] \leftarrow$ the inner-product lower bound of the range of j;

Construction of the stairway is illustrated in Figure 3. We start with a single range containing all columns $\{1,\ldots,n\}$ of A with lower bound 0 (Figure 3(a)). Then, we consider the first row with index set J_1. All columns $j,k \in J_1$ share at least one row

(namely, row 1), and therefore, $\langle a_j, a_k \rangle \geq 1$. So, we can move all columns in J_1 from $\{1, \ldots, n\}$ to a new range with lower bound 1. Next, we consider J_2. All columns that are in both J_1 and J_2 share at least two rows, and therefore they can be moved to the range $J_1 \cap J_2$ with lower bound 2. Both ranges in Figure 3(b) can have a nonempty intersection with J_2 and therefore both can be split, resulting in up to four ranges (Figure 3(c)). We continue adding rows 3, 4, $\ldots$, m and subdivide ranges until they contain N columns, moving columns that share a large number of rows up the stairway in the process.

To prevent the algorithm from terminating early because all columns are assigned to small ranges, we order the rows of A by decreasing number of nonzeros: $|J_1| \geq |J_2| \geq \cdots \geq |J_m|$. With this ordering, it may still happen early on that two large rows only share a small number of columns. We found that we could remedy this by only accepting intersections of a range $r \subseteq \{1, \ldots, n\}$ and row i with probability $|r \cap J_i| / \min\{|r|, |J_i|\}$.

The details of this procedure are described in Algorithm 5. Here, we generate a partitioning of all columns $\{1, \ldots, n\}$ into disjoint ranges by assigning an index to each column, stored in the *range* array, such that columns belong to the same range if and only if they have the same index. We keep track of the number of columns in each range, stored in the *size* array, as well as the lower bound on the inner products between all pairs of columns in the same range, stored in the *ip* array. We denote the number of ranges by R.

In Algorithm 5, we start out with a single range, with index 1, that contains all columns of A. Suppose that at some stage we process row i with column indices J_i. Algorithm 5 first determines the indices in V of all ranges that have a nonempty intersection with row i and stores the size of these intersections in the array *intersect*. To prevent adding the same range twice to V, we use the *visited* array. Then, each visited range $r \in V$ either remains intact or is split into two ranges.

Suppose $r \in V$ needs to be split. This means that we have to create two new ranges, one containing *intersect*$[r]$ columns and one containing *size*$[r]$ − *intersect*$[r]$ columns. The latter can overwrite r, so that we only need to create one new range, which is done using the *child* array. Initially, all indices in the *child* array are set to 0, indicating that no ranges need to be split. To indicate that we want to move the *intersect*$[r]$ columns shared by r and row i into a new range, we set *child*$[r]$ to the index of this new range. After doing this for all $r \in V$, we move all columns $j \in J_i$ to the new ranges *child*[*range*$[j]$], provided this value is not equal to zero, by modifying *range*$[j]$. Note that the maximum number of ranges is equal to n, because all ranges need to be disjoint and nonempty.

The advantage of Algorithm 5 is that it can generate the stairs in $\mathcal{O}(\text{nz} + m + n)$ time and using $\mathcal{O}(\text{nz} + m + n)$ storage, which makes `Stairway` very fast and not susceptible to slowdowns because of nearly dense rows in A. Note that Algorithm 5 only needs to be called once for the entire ATAM algorithm. However, the quality of `Stairway` (see Table II and Figure 4) is not very good compared to `Mondriaan`.

It was pointed out to us that Algorithm 5 is very similar to the supervariable algorithm [Duff and Reid 1996, Section 2.5] applied to A^T. Indeed, we can determine groups of columns that have exactly the same sets of row indices by choosing $N = 1$ and always splitting ranges instead of using randomization. In that sense, Algorithm 5 can be viewed as an adaptation of the supervariable algorithm to the ATAM problem.

To improve upon the quality, we combine the strengths of `Stairway` and `Mondriaan` into a new algorithm `StairwayM` (Algorithm 6). The main disadvantage of `Stairway` is that the algorithm often terminates before it can treat rows with a small number of nonzeros, while these rows can have a large influence on the column's inner products. The main disadvantage of `Mondriaan` is that the algorithm slows down when treating rows with a large number of nonzeros.

ALGORITHM 5: Building column ranges for a given matrix $A \in \{0,1\}^{m \times n}$.

Data: Matrix $A \in \{0,1\}^{m \times n}$ and minimum range size N.
Result: An array *range* of length n containing the range index of every column of A.

```
for j = 1 to n do // Create starting range.
    range[j] ← 1;
ip[1] ← 0 ;                                  // Inner product lower bound for range 1 is 0.
size[1] ← n ;                                          // Range 1 contains all n columns.
visited[1] ← false ;                                     // Range 1 has not been visited.
child[1] ← 0 ;                                  // Currently, range 1 has no child ranges.
intersect[1] ← 0;
R ← 1;
for rows i of A in order of decreasing |J_i|, with |J_i| ≥ N do
    V ← ∅;
    foreach j ∈ J_i do // Mark all ranges that contain columns in row i.
        if not visited[range[j]] then
            V ← V ∪ {range[j]};
            visited[range[j]] ← true ;
        intersect[range[j]] ← intersect[range[j]] + 1;
    foreach r ∈ V do // Split visited ranges in V if necessary.
        if size[r] = intersect[r] then
            ip[r] ← ip[r] + 1;
        else if intersect[r] ≥ N and intersect[r] ≤ size[r] − N then
            if random(0, 1) ≤ intersect[r] / min{size[r], |J_i|} then
                R ← R + 1 ;                                        // Create new range R.
                child[r] ← R ;                                  // Set r's child range to R.
                size[r] ← size[r] − intersect[r];
                ip[R] ← ip[r] + 1;
                size[R] ← intersect[r];
                visited[R] ← false ;
                child[R] ← 0;
        visited[r] ← false ;
    foreach j ∈ J_i do // Reassign columns in split ranges.
        if child[range[j]] ≠ 0 then
            range[j] ← child[range[j]];
    foreach r ∈ V do
        child[r] ← 0;
        intersect[r] ← 0;
```

This inspired us to split the matrix into top and bottom parts A_t (rows i with $|J_i| \leq N$) and A_b (rows i with $|J_i| > N$), for a certain size parameter N, and treat A_t with `Mondriaan` and A_b with `Stairway`. Doing so is an improvement but has two disadvantages: (i) the set of neighboring columns that is constructed for the column under consideration can grow very quickly if the sparsity patterns of A_t's rows are disjoint, and (ii) rows with only slightly more than N nonzero elements are likely to be rejected for splitting ranges during Algorithm 5, as the minimum range size is N.

To remedy both issues, we continue considering rows a little longer (going from A_t into A_b), as long as we do not discover too many new neighbors. This also requires us to only consider the first N nonzeros contained in rows i with $|J_i| > N$, because otherwise `StairwayM` would experience the same slowdown caused by dense rows as `Mondriaan`. Note that this causes the scores S calculated by `StairwayM` to be approximations, not necessarily lower bounds, of the actual inner products, because the same row may be counted more than once.

ALGORITHM 6: `StairwayM` combination of `Stairway` and `Mondriaan` to efficiently perform `FindHeavyUnmatchedNeighbors(`j, μ, V, S`)`. Assumes that the stairway ranges for A have already been constructed using Algorithm 5 for the given N.

Data: Matrix $A \in \{0,1\}^{m \times n}$, column index j, matching μ, empty set of neighbors V, and an array of scores S, set to zero.
Result: A set of unmatched neighbors V of j, together with their scores S.

$V \leftarrow$ { up to N unmatched columns $k \neq j$ in the same range as j };
foreach $k \in V$ **do**
    $S[k] \leftarrow$ the inner-product lower bound of the range of j;
for *rows* $i \in I_j$ *in order of increasing* $|J_i|$ **do**
    if $|V| \geq 4N$ **then** return;
    for *the first* $\min\{N, |J_i|\}$ *columns* $k \in J_i$ **do**
        if $\mu[k] = k$ ***and*** $k \neq j$ **then**
            if $S[k] = 0$ **then** $V \leftarrow V \cup \{k\}$;
            $S[k] \leftarrow S[k] + 1$;

`StairwayM` takes $\mathcal{O}(\mathrm{nz}_c\, N)$ time to construct a set of heavy unmatched neighbors for a given column j, which can offer significant benefits compared to `Mondriaan`'s $\mathcal{O}(\mathrm{nz}_c\, \mathrm{nz}_r)$ in the case of nearly dense rows. We use

$$N := \left\lceil \max\left\{ \sqrt{n},\ \frac{3\,\mathrm{nz}}{2\,m} \right\} \right\rceil, \tag{9}$$

which strikes a good balance between performance and quality (the second term was added to accommodate relatively dense matrices A).

7. MATRIX COLUMN ORDERING AND TIE BREAKING

The quality of the generated matchings can be improved by reordering the columns of $A \in \{0,1\}^{m \times n}$. Suppose that we have two columns, $1 \leq j < k \leq n$, and that the $|I_j|$ and $|I_k|$ nonzeros in both these columns are randomly distributed over the m rows of A with uniform probability. For $1 \leq i \leq m$, the probability that $a_{ij} \neq 0$ equals $|I_j|/m$ and similarly the probability that $a_{ik} \neq 0$ equals $|I_k|/m$. Since the distribution of the nonzeros is independent for both columns, the probability that $a_{ij} \neq 0$ and $a_{ik} \neq 0$ equals $|I_j||I_k|/m^2$, independent of i. This means that the expected number of rows $1 \leq i \leq m$ that satisfy $a_{ij}\, a_{ik} \neq 0$ (i.e., the expected value of $\langle a_j, a_k \rangle$) equals

$$\mathrm{E}(\langle a_j, a_k \rangle) = m \frac{|I_j||I_k|}{m^2} = \frac{|I_j||I_k|}{m}. \tag{10}$$

From Equation (10), we see that the expected inner product between two columns scales directly with the number of nonzeros contained in them. Therefore, ordering the columns of A by decreasing number of nonzeros (such that $|I_j| \geq |I_k|$ for all $1 \leq j \leq k \leq n$) and matching the columns in this order is expected to yield the highest inner products. Even though this is no guarantee (see Equation (8)), improved quality is confirmed by the experiments performed in Section 8 (see the last panel of Figure 4). Unless stated otherwise, we will order the columns of A by decreasing number of nonzeros.

The quality can also be improved by better tie breaking. Instead of returning an arbitrary best-scoring column $k \in V$, we return the best-scoring column k for which the number of nonzeros, $|I_k|$, is smallest. By Equation (10), this is a sensible approach, because columns k with smaller $|I_k|$ have a smaller probability of reaching a higher inner product with columns other than the column j to be matched. This has a positive

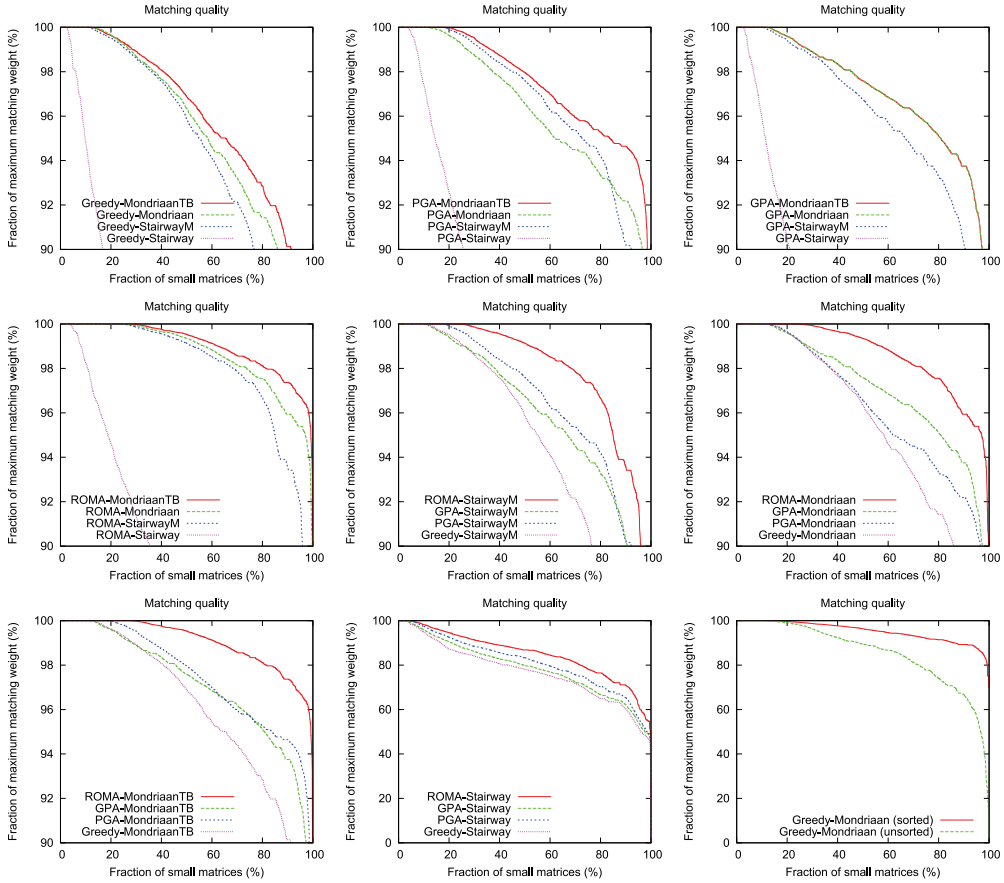


Fig. 4. The obtained matching weight, as percentage of the weight of an optimal solution of ATAM, as a function of the fraction of instances with at least this matching weight, for all sparse matrices from the small-matrix test set and their transposes. In the upper-left panel, for instance, about 20% of the matrices have at least 90% of the optimal matching weight for the Greedy-Stairway method. The image at the bottom right illustrates the influence of processing the columns of A in natural order (unsorted) or by decreasing number of nonzeros (sorted); see Section 7. Note that the vertical range of the last two images, $[0, 100]$, is different from the vertical range of the other images, $[90, 100]$.

effect on the matching quality, as can be seen in Table II for the tie-breaking (TB) variants.

8. EXPERIMENTAL RESULTS

We have implemented the `Greedy` (Algorithm 2), `PGA`, `GPA`, and `ROMA` matching algorithms from Section 5 in C++, as well as the implementations `Mondriaan` (Algorithm 3), `Stairway` (Algorithm 4), and `StairwayM` (Algorithm 6) of `FindHeavyUnmatchedNeighbors()` used by these matching algorithms. An implementation of `Mondriaan` with different tie breaking, Section 7, has been included as `MondriaanTB`. In total, this gives us 16 combinations of matching algorithms and neighbor finders to investigate for ATAM. Recall that `Greedy-Mondriaan` is Mondriaan 3.11's ATAM algorithm. It is important for the partitionings generated by Mondriaan that the quality of the generated matchings is as good as possible for a wide variety of matrices.

Table I. Statistics of the Large-Matrix Test Set

A	m	n	nz	$\lfloor \frac{\text{nz}}{m} \rfloor$	nz_r	nz_c
dfl001	6,071	12,230	35,632	5	228	14
rhpentium	25,187	25,187	258,265	10	22,818	22,818
cre_b	9,648	77,137	260,785	27	844	14
tbdmatlab	19,859	5,979	430,171	21	2,921	1,423
nug30	52,260	379,350	1,567,800	30	30	60
c98a	56,243	56,274	2,075,889	36	29,622	149
tbdlinux	112,757	20,167	2,157,675	19	7,121	3,721
stanford	281,903	281,903	2,312,497	8	38,606	255
cage13	445,315	445,315	7,479,343	16	39	39
stanfordberkeley	683,446	683,446	7,583,376	11	83,448	249
wikipedia-20051105	1,634,989	1,634,989	19,753,078	12	4,970	75,547

For each matrix A, the number of rows m, number of columns n, and number of nonzeros nz are given, as well as the (rounded) average number of nonzeros per row, the maximum number nz_r per row, and the maximum number nz_c per column.

The *small-matrix test set* consists of 1,681 matrices, of which the first 1,443 are consecutive, from the full list of 2,542 matrices available from Davis and Hu [2011], ordered by their number of nonzeros (accessed March 6, 2012). The number of rows and columns of these small matrices ranges from 1 to 129,164 and the number of nonzeros ranges from 1 to 714,241. Because of the Mondriaan matrix partitioning algorithm, which at each split chooses between row and column partitioning directions [Vastenhouw and Bisseling 2005], we generate matchings for both A and A^T for all matrices A in the small-matrix test set, as long as we can determine the weight of the optimal solution without running out of memory. We can determine this weight for 1,677 matrices without transposition (i.e., for A) and for 1,680 matrices with transposition (i.e., for A^T), resulting in 1,681 matrices for which we can find either optimal weight.

The *large-matrix test set* (see Table I) is drawn from a multitude of fields in scientific computing (cryptography, web search, information retrieval, circuit simulation, DNA electrophoresis, linear programming, etc.) and therefore provides a meaningful test set of real-world data. The test set consists of 10 matrices used in Bisseling et al. [2012], together with the matrix rhpentium from Sandia National Laboratories.

In order to compare the performance of all 16 algorithms, we normalize, per matrix A, the obtained ATAM weights $\Omega_A(M)$ for matchings M by the upper bound (Equation (2)) to obtain a value between 0 and 1 as a dimensionless indicator of M's quality. Apart from this, we also use the LEMON 1.2.3 template graph library [Dezsö et al. 2011] to find the optimal solution to WGM for $G(A)$ and normalize $\Omega_A(M)$ by the weight of this optimal solution. Unfortunately, the memory requirements of LEMON only permitted us to find optimal solutions for the small-matrix test set. The experiments were performed on an Intel Core i7 860 (2.8GHz) with 8GiB RAM and compiled using g++ 4.7.2.

Table II compares all ATAM algorithms with respect to both quality measures. We generate a matching 10 times for every matrix, as Algorithms 5 and 6 generate different results with different random seeds, and average the normalized matching weights over these 10 runs. Recorded timing results are also averaged over 10 runs and include the time required to sort the rows and columns of the matrix.

To get a good impression of the quality of the 16 algorithms under consideration, we have plotted, in Figure 4, the fraction of instances for which at least a certain fraction of the highest possible ATAM weight (as determined by LEMON) is obtained by the matching heuristic. The mean of all normalized ATAM weights can be found in Table II.

The effect of ordering A's columns by decreasing number of nonzeros on the matching quality (Section 7) is investigated in the last panel of Figure 4 for Greedy-Mondriaan,

Table II. Average Matching Quality Compared to the Weight of an Optimal Solution of ATAM (left) and to the Upper Bound $\lfloor \frac{nz(A)}{2} \rfloor$ (right) for the Different Matching Algorithms from Section 5, for the Small-Matrix Test Set

Method	Mean (%) (w.r.t. Opt.)	Std. dev. (%) (w.r.t. Opt.)	Mean (%) (w.r.t. $\lfloor \frac{nz(A)}{2} \rfloor$)
ROMA-MondriaanTB	99.0	1.3	56.6
ROMA-Mondriaan	98.6	1.8	56.4
ROMA-StairwayM	97.8	3.9	55.9
PGA-MondriaanTB	97.4	2.7	55.7
GPA-MondriaanTB	97.0	3.1	55.4
GPA-Mondriaan	97.0	3.1	55.4
PGA-StairwayM	96.4	4.2	55.2
PGA-Mondriaan	96.2	3.4	55.0
Greedy-MondriaanTB	95.9	4.1	54.9
GPA-StairwayM	95.5	5.3	54.5
Greedy-Mondriaan	95.3	4.5	54.6
Greedy-StairwayM	94.2	6.2	54.0
ROMA-Stairway	85.0	11.0	49.4
PGA-Stairway	81.3	12.5	47.4
GPA-Stairway	78.8	12.9	45.9
Greedy-Stairway	77.0	13.3	45.0

where we find that the quality of the generated matchings is improved significantly. Since ordering the matrix columns can be done with an $\mathcal{O}(n+m)$ counting sort, doing so is an efficient way to increase matching quality. (This is also the default option in Mondriaan 3.11.)

For the large-matrix test set, we have compared the different matching algorithms from Section 5 in Figure 5, where we use Mondriaan neighbor finding. Here, we see that ROMA is quite slow compared to the other matching algorithms, while it does offer the best quality. Based on the results from Table II, we have compared the performance of a selection of ATAM algorithms in Figure 6. GPA does not perform significantly better than PGA, which we suspect can be attributed to the fact that we do not process all edges in $G(A)$ in order of descending weight, as mentioned in Section 5. Since PGA is faster than GPA, this makes PGA a more suitable matching algorithm for ATAM.

From Table II, we see that ROMA-MondriaanTB yields the best results, as was to be expected. However, in Figure 5, we can see that the ROMA algorithm is very expensive to execute, because already matched columns can be reconsidered and hence may not be removed from the matrix. Therefore, for large matrices, the PGA-MondriaanTB combination with tie breaking as described in Section 7 is a better choice: the quality is significantly better than that of the original algorithm (Greedy-Mondriaan), it is guaranteed to be a $\frac{1}{2}$-approximation (while the original can yield arbitrarily bad results, Equation (8)), and it is only slightly slower than the original algorithm (Figure 6).

The Stairway neighbor-finding heuristic performs quite badly, but this is improved significantly by using StairwayM: PGA-StairwayM offers slightly better quality than Greedy-Mondriaan according to Table II, while it is a lot faster for matrices with dense rows (such as rhpentium, c98a, stanford, tbdlinux, and stanfordberkeley), as shown in Figure 6 and Table I. On average, PGA-StairwayM is 5.3 times as fast as Greedy-Mondriaan for large matrices, with speedups as high as 19.6 for c98a. For the set of large matrices, the quality of PGA-StairwayM is 2% better, on average, than that of Greedy-Mondriaan.

We implemented PGA-StairwayM and PGA-MondriaanTB in version 4.0 of the Mondriaan matrix partitioner and compared the partitioning performance of Mondriaan 4.0

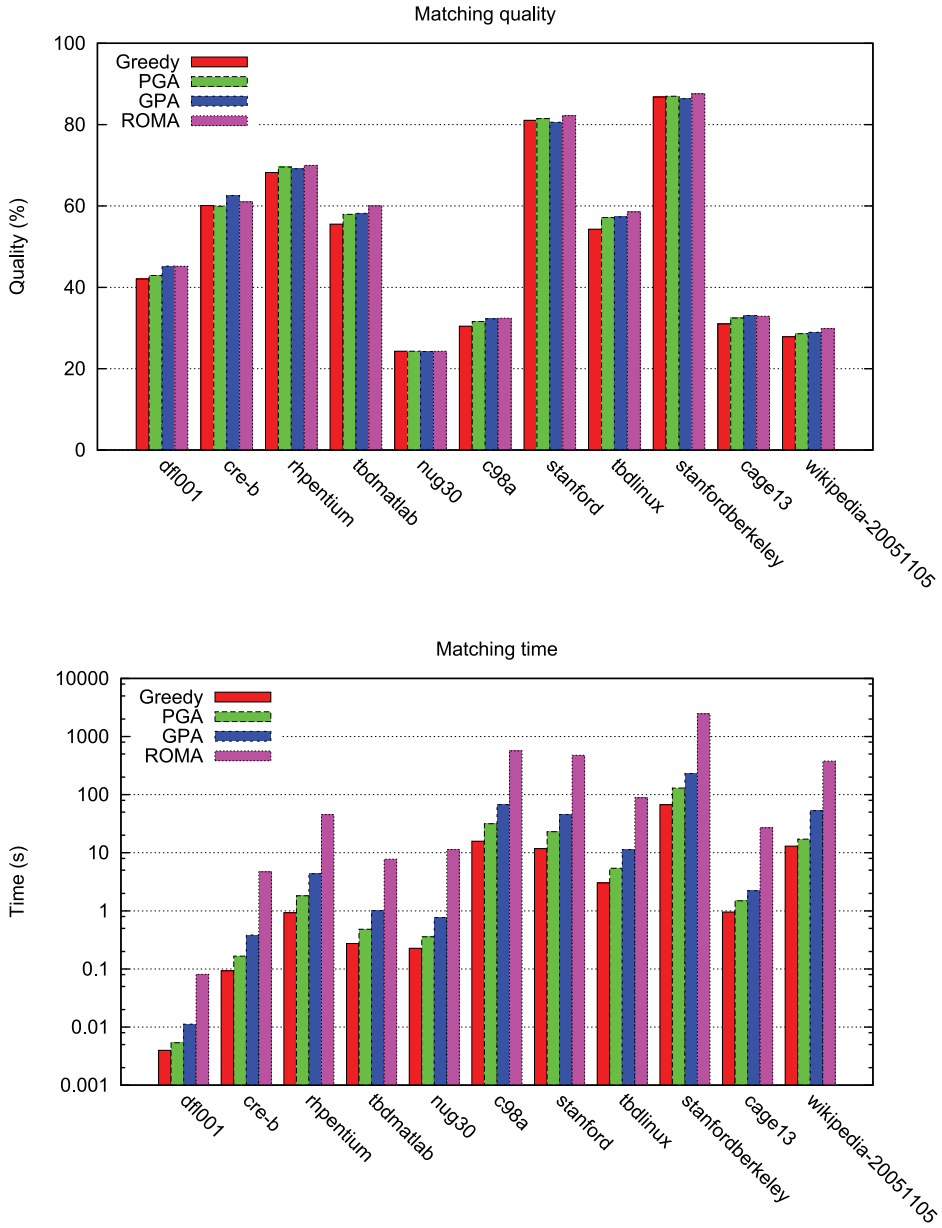


Fig. 5. Timings and quality (as percentage of the upper bound $\lfloor \frac{nz(A)}{2} \rfloor$) of the algorithms from Section 5 with `Mondriaan` neighbor finding, for the large-matrix test set.

with that of Mondriaan 3.11, which uses the `Greedy-Mondriaan` heuristic. To enable proper comparison, both the old and the new heuristics are tested within the new version 4.0 of the Mondriaan package. The communication volume (Equation (3)) for partitionings of matrices from the large-matrix test set into two parts with an imbalance of $\epsilon = 0.03$ is shown in Table III for the row-net hypergraph model (see Problem 1.3 in Section 1) and in Table IV for the medium-grain hypergraph method. The medium-grain hypergraph method (see Pelt and Bisseling [2014]) was included because it offers

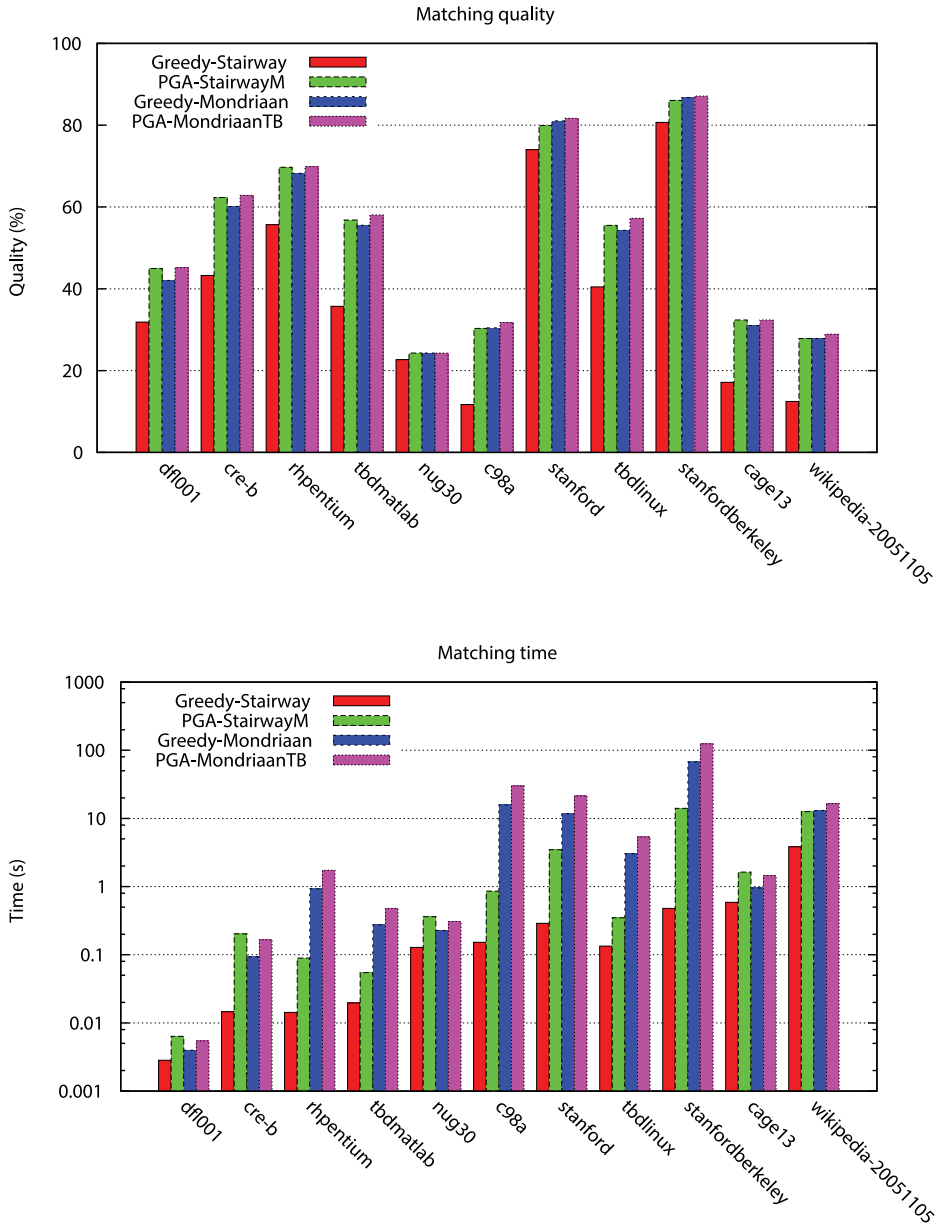


Fig. 6. Timings and quality (as percentage of the upper bound $\lfloor \frac{\mathrm{nz}(A)}{2} \rfloor$) of a selection of ATAM algorithms, for the large-matrix test set.

lower communication volumes than the row-net model by partitioning both rows and columns; it is also the default method used for Mondriaan 4.0.

Comparing Figure 6 with Tables III and IV yields a number of surprising results. First, note the 2.4 times higher communication volume of `PGA-StairwayM` for `rhpentium` in Table IV, which is unexpected given the nearly equal quality displayed in Figure 6. This difference in quality can be explained by the fact that the medium-grain method reduces the number of dense matrix rows and columns (see Pelt and Bisseling [2014]).

Table III. Partitioning Time and Communication Volume (Equation (3)) for Bipartitionings of Matrices from Table I Modeled as Row-net Hypergraphs with an Imbalance of $\epsilon = 0.03$, Generated Using the New Matching Heuristics Implemented in Mondriaan 4.0, Compared to the Old Matching Heuristic from Mondriaan 3.11

	Mondriaan 3.11		PGA-StairwayM		PGA-MondriaanTB	
A	Time (s)	CV	Time (s)	CV	Time (s)	CV
dfl001	1.4e−1	596	1.2e−1	612	1.6e−1	591
rhpentium	7.2e+0	11,265	9.5e−1	10,436	1.4e+1	10,700
cre_b	1.4e+0	620	1.1e+0	915	2.7e+0	546
tbdmatlab	2.0e+0	6,412	1.4e+0	6,442	3.4e+0	6,391
nug30	1.6e+1	30,491	1.3e+1	35,739	1.5e+1	35,006
c98a	1.0e+2	44,879	2.9e+1	45,321	2.4e+2	44,779
tbdlinux	2.1e+1	24,105	9.8e+0	24,212	4.3e+1	23,281
stanford	9.6e+1	279	3.7e+1	371	2.2e+2	426
cage13	4.9e+1	57,494	4.0e+1	62,283	5.0e+1	59,800
stanfordberkeley	2.8e+2	812	1.4e+2	671	5.9e+2	806
wikipedia-20051105	5.6e+2	458,108	2.6e+2	468,845	6.4e+2	454,647

Table IV. Partitioning Time and Communication Volume (Equation (3)) for Bipartitionings of Matrices from Table I Modeled as Medium-Grain Hypergraphs with an Imbalance of $\epsilon = 0.03$, Generated Using the New Matching Heuristics Implemented in Mondriaan 4.0, Compared to the Old Matching Heuristic from Mondriaan 3.11

	Mondriaan 3.11		PGA-StairwayM		PGA-MondriaanTB	
A	Time (s)	CV	Time (s)	CV	Time (s)	CV
dfl001	1.9e−1	583	2.2e−1	614	2.1e−1	592
rhpentium	1.6e+1	1,448	2.6e+0	3,460	3.9e+1	1,486
cre_b	1.7e+0	577	1.3e+0	963	2.9e+0	513
tbdmatlab	4.9e+0	3,877	2.2e+0	4,020	6.2e+0	3,763
nug30	2.0e+1	36,118	2.3e+1	28,345	2.7e+1	23,116
c98a	2.0e+2	35,823	3.4e+1	40,333	3.0e+2	36,105
tbdlinux	7.3e+1	7,977	1.6e+1	10,882	8.7e+1	7,950
stanford	1.2e+2	213	4.5e+1	346	2.1e+2	195
cage13	8.1e+1	44,920	6.4e+1	48,273	8.1e+1	46,457
stanfordberkeley	3.8e+2	516	1.5e+2	718	6.3e+2	480
wikipedia-20051105	9.7e+3	183,659	7.8e+2	189,662	9.6e+3	179,269

Doing so significantly reduces communication volume but also reduces the effectiveness of the PGA-StairwayM heuristic. For the row-net hypergraph model, the communication volume of PGA-StairwayM for rhpentium in Table III is indeed lower than for Mondriaan 3.11.

Second, PGA-StairwayM is only 3.5 times faster than Greedy-Mondriaan for c98a in Table III, instead of the 19.6 times speedup in Figure 6. This is due to the fact that for c98a, the Mondriaan partitioner spends a relatively small amount of time on matching, compared to the refinement of partitionings (whereas, for rhpentium, the majority of the time is spent on matching).

Note, however, that using PGA-MondriaanTB results in partitionings with a lower communication volume overall, compared with Greedy-Mondriaan, at the expense of a longer partitioning time. This is in line with the results from Table II . The combination PGA-MondriaanTB has, therefore, been made the default in version 4.0 of the Mondriaan package.

PGA-StairwayM solves the large problem wikipedia-20051105 in 13 minutes instead of 162 minutes (i.e., about 12 times faster), making partitioning practically feasible, at a modest 3% increase in communication volume. However, this heuristic can sometimes

give relatively bad results (see cre_b with 67% extra communication), but it may be the method to choose when computing times would become intractable otherwise.

9. CONCLUSION

In this article, we have introduced the column intersection graph or $A^T A$-matching problem ATAM in the context of matrix partitioning and established the relation of this problem to weighted graph matching. We have performed an investigation of 16 different algorithms to solve ATAM on a large set of real-world matrices and found a solution algorithm for ATAM that combines PGA' graph matching with Mondriaan's exact neighbor finding, which increases the quality of the generated matchings in the Mondriaan matrix partitioner and is guaranteed to be a $\frac{1}{2}$-approximation. Apart from an algorithm offering improved quality, we also provided a combined stairway/Mondriaan heuristic that in practice offers slightly better quality than Mondriaan's current ATAM algorithm, at much higher speed for matrices that possess dense rows: we attained speedups of up to 19.6 for the sparse test matrix c98a. We furthermore implemented both algorithms in the Mondriaan 4 matrix partitioner and showed that they indeed can be used to improve partitioning quality or reduce computation time.

REFERENCES

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

R. H. Bisseling, B. O. Fagginger Auer, A. N. Yzelman, T. van Leeuwen, and Ü. V. Çatalyürek. 2012. Two-dimensional approaches to sparse matrix partitioning. In *Combinatorial Scientific Computing*, U. Naumann and O. Schenk (Eds.). CRC Press, Taylor & Francis Group, Boca Raton, FL, 321–349. DOI:http://dx.doi.org/10.1201/b11644-13

Ü. V. Çatalyürek and C. Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Comput.* 10, 7 (1999), 673–693. DOI:http://dx.doi.org/10.1109/71.780863

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

Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar. 2012. Multithreaded clustering for multi-level hypergraph partitioning. In *Proc. IPDPS 2012*. IEEE, Los Alamitos, CA, 848–859. DOI:http://dx.doi.org/10.1109/IPDPS.2012.81

T. A. Davis and Y. Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1, Article 1, 25 pages. DOI:http://dx.doi.org/10.1145/2049662.2049663

K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. 2006. Parallel hypergraph partitioning for scientific computing. In *Proc. IPDPS 2006*. IEEE, Los Alamitos, CA. DOI:http://dx.doi.org/10.1109/IPDPS.2006.1639359

B. Dezsö, A. Jüttner, and P. Kovács. 2011. LEMON—An open source C++ graph template library. *Electron. Notes Theor. Comput. Sci.* 264, 5 (2011), 23–45. DOI:http://dx.doi.org/10.1016/j.entcs.2011.06.003

D. E. Drake and S. Hougardy. 2003a. Improved linear time approximation algorithms for weighted matchings. In *Proc. Approx/Random*. LNCS, Vol. 2764. Springer, Berlin, 14–23. DOI:http://dx.doi.org/10.1007/978-3-540-45198-3_2

D. E. Drake and S. Hougardy. 2003b. Linear time local improvements for weighted matchings in graphs. In *Proc. Workshop Exp. Alg. 2003*. LNCS, Vol. 2647. Springer, Berlin, 107–119. DOI:http://dx.doi.org/10.1007/3-540-44867-5_9

D. E. Drake and S. Hougardy. 2003c. A simple approximation algorithm for the weighted matching problem. *Inform. Process. Lett.* 85, 4 (2003), 211–213. DOI:http://dx.doi.org/10.1016/S0020-0190(02)00393-9

R. Duan, S. Pettie, and H.-H. Su. 2011. Scaling algorithms for approximate and exact maximum weight matching. *CoRR* abs/1112.0790 (2011).

I. S. Duff and J. K. Reid. 1996. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Software* 22, 2 (1996), 227–257. DOI:http://dx.doi.org/10.1145/229473.229480

J. Edmonds. 1965a. Maximum matching and a polyhedron with 0,1-vertices. *J. Res. Nat. Bur. Standards* 69B, 1/2 (1965), 125–130. DOI:http://dx.doi.org/10.6028/jres.069B.013

J. Edmonds. 1965b. Paths, trees, and flowers. *Canad. J. Math.* 17 (1965), 449–467. DOI:http://dx.doi.org/10.4153/CJM-1965-045-4

C. M. Fiduccia and R. M. Mattheyses. 1982. A linear-time heuristic for improving network partitions. In *Proc. 19th IEEE DAC*. IEEE, Los Alamitos, CA, 175–181. DOI:http://dx.doi.org/10.1109/DAC.1982.1585498

H. N. Gabow and R. E. Tarjan. 1991. Faster scaling algorithms for general graph matching problems. *J. ACM* 38, 4 (1991), 815–853. DOI:http://dx.doi.org/10.1145/115234.115366

A. H. Gebremedhin, F. Manne, and A. Pothen. 2005. What color is your jacobian? Graph coloring for computing derivatives. *SIAM Rev.* 47, 4 (2005), 629–705. DOI:http://dx.doi.org/10.1137/S0036144504444711

M. Holtgrewe, P. Sanders, and C. Schulz. 2010. Engineering a scalable high quality graph partitioner. In *Proc. IPDPS 2010*. IEEE, Los Alamitos, CA, 1168–1179. DOI:http://dx.doi.org/10.1109/IPDPS.2010.5470485

B. W. Kernighan and S. Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Techn. J.* 49, 2 (1970), 291–307.

J. Maue and P. Sanders. 2007. Engineering algorithms for approximate weighted matching. In *Proc. Workshop Exp. Alg. 2007*. LNCS, Vol. 4525. Springer, Berlin, 242–255. DOI:http://dx.doi.org/10.1007/978-3-540-72845-0_19

D. M. Pelt and R. H. Bisseling. 2014. A medium-grain method for fast 2D bipartitioning of sparse matrices. In *Proc. IPDPS 2014*. IEEE, Los Alamitos, CA, 529–539. DOI:http://dx.doi.org/10.1109/IPDPS.2014.62

S. Pettie and P. Sanders. 2004. A simpler linear time 2/3-ϵ approximation for maximum weight matching. *Inform. Process. Lett.* 91, 6 (2004), 271–276. DOI:http://dx.doi.org/10.1016/j.ipl.2004.05.007

R. Preis. 1999. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Proc. STACS 1999*. LNCS, Vol. 1563. Springer, Berlin, 259–269. DOI:http://dx.doi.org/10.1007/3-540-49116-3_24

B. Vastenhouw and R. H. Bisseling. 2005. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.* 47, 1 (2005), 67–95. DOI:http://dx.doi.org/10.1137/S0036144502409019

Received February 2013; revised February 2014; accepted April 2014