Parallel Sparse PLUQ Factorization modulo p

Charles Bouillaguet
Université de Lille, CRIStAL
Cité Scientifique
59650, Villeneuve d'Ascq
charles.bouillaguet@univ-lille1.fr

Claire Delaplace
Université de Lille, CRIStAL
Université de Rennes, IRISA
263 Avenue Général Leclerc
35000, Rennes
claire.delaplace@irisa.fr

Marie-Emilie Voge
Université de Lille, CRIStAL
Cité Scientifique
59650, Villeneuve d'Ascq
marie-emilie.voge@univ-lille1.fr

ABSTRACT

In this paper, we present the results of our experiments to compute the rank of several large sparse matrices from Dumas's Sparse Integer Matrix Collection, by computing sparse PLUQ factorizations.

Our approach consists in identifying as many pivots as possible before performing any arithmetic operation, based solely on the location of non-zero entries in the input matrix. These "structural" pivots are then all eliminated in parallel, in a single pass. We describe several heuristic structural pivot selection algorithms (the problem is NP-hard).

These algorithms allows us to compute the ranks of several large sparse matrices in a few minutes, versus many days using Wiedemann's algorithm. Lastly, we describe a multi-thread implementation using OpenMP achieving 70% parallel efficiency on 24 cores on the largest benchmark.

KEYWORDS

Sparse Linear Algebra, Gaussian Elimination, Structural Pivots, Parallel Pivots Selection Algorithm

ACM Reference format:
Charles Bouillaguet, Claire Delaplace, and Marie-Emilie Voge. 2017. Parallel Sparse PLUQ Factorization modulo p. In *Proceedings of PASCO '17,*
, July 23–24, 2017, Kaiserslautern, Germany, 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The PLUQ factorization of an $n \times m$ matrix A is the product $A = PLUQ$. There, P, Q are permutation matrices, L is an $n \times r$ lower-trapezoidal matrix with all non-zero diagonal coefficients, and U is an $r \times m$ upper-trapezoidal matrix with unit diagonal, where r denotes the rank of A. This decomposition reveals the rank of A and allows to solve $Ax = y$ efficiently. PLUQ factorizations are typically computed by some form of gaussian elimination: while it is possible, choose a pivot and eliminate it.

When the input matrix A is sparse, then a sparse PLUQ should be computed, producing factors L, U that are as sparse as possible. A good pivot selection strategy is critical in maintaining sparsity during the echelonization process. For example, in the following matrix, choosing the top-left entry as pivot and performing an elimination step results in a fully dense submatrix, while choosing the bottom-right entry leads to no fill-in at all.

$$\begin{pmatrix} \otimes & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \otimes \end{pmatrix}$$

Let us write $|M|$ the number of non-zero entries in the sparse matrix M. Finding the sequence of pivots that minimizes the amount of fill-in $|L| + |U| - |A|$ is NP-hard [31], so sparse matrix factorization algorithms use heuristics to select pivots. If too much fill-in accumulates during the factorization, the computation may come to a grinding halt, or simply fail if not enough memory is available.

1.1 Pivot Selection in Numerical Algorithms

Many sparse matrix factorization algorithms have been designed to deal with square, invertible, floating-point matrices. When A is symmetric positive definite, a sparse Cholesky factorization $A = {}^tLL$ can be computed. In that case, the pivots are on the diagonal, and many algorithms have been designed to select an *a priori* order on the pivots, *i.e.* a permutation such that ${}^tPAP$ is easier to factorize than A. Approximate Minimum Degree [1] or Nested Dissection [14] are such algorithms. They only exploit the structure of A (the locations of non-zero entries) and disregard the numerical values.

When the input matrix is square and invertible but not symmetric, a PLU factorization can be computed. In that case, pivots are chosen to maintain both sparsity and numerical accuracy. As such, the actual choice of pivots will ultimately depend on numerical values that are not known in advance, and choosing an *a priori* sequence of pivots is much more difficult. One option is to look at the stucture of $A + {}^tA$ or A^tA (which are symmetric), and use symmetric pivot-selection algorithms to define an order in which the columns of A will be processed. Then, during the numerical factorization, choose inside each column the pivot that maximizes accuracy (this is called "partial pivoting").

Alternatively, pivots can be chosen "on-line" during the numerical factorization. Markowitz pivoting [24] chooses the next pivot in a greedy way, to minimize fill-in caused by the next elimination step (excluding those that would lead to numerical instability). Of course, this requires the previous elimination steps to have already been performed. This strategy has been implemented in LinBox for exact sparse elimination.

When the coefficients of the input matrix live in an exact field such as $\mathbb{Q}$ or $\mathbb{Z}_p$, numerical accuracy is no longer a problem. However, numerical cancellation becomes much more likely (and *does*

PASCO '17,

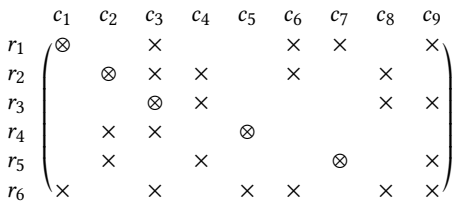
DOI: 10.1145/nnnnnnn.nnnnnnn

occur if A is rectangular or rank-deffective). *A priori* pivot choice is not made much easier because an entry selected beforehand to be a pivot may vanish in a subsequent elimination step.

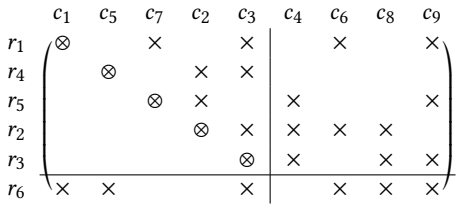
1.2 Structural Pivots

Entries that can be used as pivots can sometimes be identified based solely on the pattern of non-zero entries. In the following matrix, circled entries can be chosen as pivots regardless of the actual values. We call these *structural pivots*.

Example 1.1.



The rows and column can be permuted as follows:



A set of k structural pivots has been identified if the rows and columns of the matrix can be permuted such that the top-left $k \times k$ submatrix is upper-triangular, as shown in this example.

We claim that being able to find many structural pivots is desirable. In the extreme case, if *all* pivots can be found based on the structure of the matrix, then the echelonization process can be carried over without any arithmetic operation, just by permuting the rows and columns — and without any fill-in.

If a large number of structural pivots can be identified, then the coefficients below them can all be eliminated in one fell swoop and the result (the Schur complement) can be easily computed in parallel. If the number of pivots found is close to the rank, then the Schur complement will have small rank, and will be easily manageable.

Unfortunately, finding the maximum number of structural pivots in a sparse matrix is NP-hard (cf. section 3.1).

1.3 Exact Linear Algebra Modulo p

A series of work considered the problem of computing the rank of sparse matrices modulo a small prime p. Sparse PLUQ factorizations can be used to this end, and computing the rank is not much easier than computing a PLUQ, so both problems are more-or-less equivalent.

Iterative methods such as the Wiedemann algorithm [30] can be used in exact linear algebra to perform the usual operations on sparse matrices (computing the rank, solving linear systems, etc.). They basically work by performing a series of matrix-vector products and only need to store one or two vectors in addition to the matrix. Their time complexity is essentially $O(r|A|)$ and their running is easy to predict. Iterative methods are sometimes slower than sparse gaussian elimination, but they do not suffer from fill-in, and thus "cannot fail". As such, they are sometimes the only option for matrices on which fill-in makes direct methods impractical.

A survey [10] on sparse rank computation mod p compared the performance of the Wiedemann algorithm with that of a sparse gaussian elimination algorithm, both implemented in the LinBox library. It is proposed to parallelize the matrix-vector products in the Wiedemann algorithm: a speed-up of ≈ 3 on 4 processors is reported.

In [9], the authors were interested in large matrices originating from algebraic K-theory (these are the matrices from the GL7d folder of Dumas's Sparse Integer Matrices Collection). Computing the rank of these matrix, of size up to $\approx 2 \cdot 10^6$ with density 10^{-5}, reportedly took up to 35 days on 50 Itanium2 processors, using a parallel implementation of the block-Wiedemann algorithm [7]. The authors estimated that the sequential computation would have taken 321 days.

Faugère and Lachartre [12] designed an algorithm to compute the row-echelon form of sparse matrices produced during Gröbner basis computations. They observed that the leftmost entry of each row can be a structural pivot if no other pivot has previously been chosen on the same column. Because the Gröbner-basis matrices are nearly triangular, this strategy often finds 99.9% of the maximum number of pivots. This algorithm has been implemented in the GBLA [4] software package.

In [3], the present authors proposed an implementation of a left-looking elimination algorithm that often perform better than its more well-known right-looking counterpart. Combined with the Faugère-Lachartre structural pivot selection algorithm, this enabled us to compute the rank of some sparse matrices up to 1000× faster than the Wiedemann algorithm. However, some of the largest matrices (notably those from algebraic K-theory) remained out of the reach of sparse elimination.

1.4 Our Contribution

We present new algorithms to find structural pivots in a sparse matrix A, and use them to compute the ranks of some large sparse matrices much faster than what could be done before.

A rectangular sparse matrix can be associated with a bipartite graph: there is an edge $i \leftrightarrow j$ when $A_{ij} \neq 0$. We exploit the old observation that the edges of a *uniquely restricted matching* on this graph are structural pivots, and vice-versa. A matching is said to be uniquely restricted if the graph does not contain alternating cycles with respect to the matching. More details are given in section 3.1.

In this graph-theoretic framework, we propose two heuristic algorithms. The first one (in section 3.3) starts with a spanning tree and build a matching restricted to this tree, by removing the vertices that could potentially create an alternating cycle. It is almost linear, and can, in some cases, find more pivots than the Faugère-Lachartre heuristic.

The other algorithm we propose uses the following greedy strategy: for each $A_{ij} \neq 0$, that may be a pivot, we add A_{ij} to the pivots list, then we check if this induces an alternating cycle in the graph. If so, then A_{ij} cannot be a structural pivot (see section 3.4). This

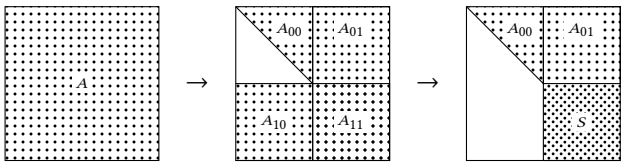


Figure 1: Structural pivots and Schur complement computation

algorithm allow us to find up to 99.9% of the total number of pivots in some cases, allowing us very fast rank computations.

A parallel implementation of this greedy algorithm is discussed in section 4, and its parallel efficiency is reported in section 5.

All in all, these pivot selection algorithms enabled us to compute the rank of several large sparse matrices from Dumas Sparse Integer Matrices Collection (SIMC), especialy the largest matrices from the GL7d folder, in a less than 5 minutes using 36 cores, where the previous best algorithm took days.

These algorithms have been implemented in C using OpenMP in a small library called SpaSM (for Sparse Solver Modulo p). Its code is publicly available at :

https://github.com/cbouilla/spasm

We are currently working with the LinBox team to incorporate these algorithms in the LinBox library.

2 SPARSE PLUQ FACTORIZATION

We used the following algorithm to compute PLUQ factorizations (more details are in [3]):

(1) Find a set of structural pivots.
(2) Eliminate them; this yields a Schur complement S.
(3) Estimate the density of S :
   (a) If S is sparse, compute its sparse PLUQ recursively.
   (b) If S is dense, compute its dense PLUQ.

An illustration of the first two steps is given in Figure 1. The main idea underlying this algorithm is essentially a greedy strategy to minimize fill-in in L and U: rows in which a structural pivot has been found in step 1 are copied as-is into U, with no fill-in at all. It follows that finding the largest possible set of structural pivots is usually beneficial.

Structural pivot selection strategies are discussed in section 3. Other aspects of the algorithm are outlined below.

2.1 Schur Complement Computation

If k structural pivots are found in step 1, then we are given permutations P and Q such that the $k \times k$ upper-left block of PAQ is upper-triangular with non-zero diagonal, and A can be decomposed as follow:

$$PAQ = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} I & \\ A_{10}A_{00}^{-1} & I \end{pmatrix} \begin{pmatrix} I & \\ & S \end{pmatrix} \begin{pmatrix} A_{00} & A_{01} \\ & I \end{pmatrix}$$

where S denotes the Schur complement of PAQ with respect to A_{00} that remains to be factorized. It follows that:

$$S = A_{11} - A_{10}A_{00}^{-1}A_{01}.$$

If k structural pivots have been found, then S has size $(n-k) \times (m-k)$. Denote by $(\mathbf{a}_{i0}\ \mathbf{a}_{i1})$ the i-th row of $(A_{10}\ A_{11})$, and consider the following triangular system :

$$(\mathbf{x}_0\ \mathbf{x}_1) \cdot \begin{pmatrix} A_{00} & A_{01} \\ & I \end{pmatrix} = (\mathbf{a}_{i0}\ \mathbf{a}_{i1}). \quad (1)$$

One can check that $\mathbf{x}_1 = \mathbf{a}_{i1} - \mathbf{a}_{i0}A_{00}^{-1}A_{01}$ and then $\mathbf{x}_1$ is the i-th row of S.

A row of S can thus be computed by solving a sparse triangular system with a sparse right-hand side. This can be done efficiently using the Gilbert-Peierls algorithm [15]. The rows of S can be computed independently and in parallel.

2.2 Dealing with the Schur Complement S

The density of S can be estimated with good accuracy by computing a small random subset of its rows, and measuring its density.

If S is sparse, we obtain it by solving system (1) for all values of i, and we use our algorithm recursively to obtain a sparse PLUQ of S.

If S is dense, but small enough to fit in memory, then we compute it entirely and obtain its dense PLUQ factorization, for instance using off-the-shelf code such as FFLAS-FFPACK [29].

If S is both dense and very large, the situation is more complicated. A favorable case arises when the number of structural pivots found in step 1 is close to the rank of the matrix. In this case, the rank of S is small. This makes it feasible to compute its dense PLUQ factorization: The L and U obtained will be very "thin", and we can hope to store them entirely in memory. In order to compute the rank of the original matrix, we actually implemented the following technique to compute the rank of S:

(1) Guess an upper-bound r on the rank of S.
(2) Choose a random $r \times (n-k)$ matrix M.
(3) Compute $S' = M \times S$ (dense but small).
(4) Compute a dense PLUQ factorization of S'.
(5) If S' has full row-rank, set $r \leftarrow 2r$ and restart.

S' has the same rank as S with overwhelming probability (this yields the rank of S, and thus of A). Solving $x \cdot S = y$ can be done by solving $x' \cdot S' = y$; $x' \cdot M$ is then the wanted solution.

More advanced algorithms are possible. For instance, Saunders and Youse [27] have proposed an algorithm, that can compute the rank of large $n \times n$ matrices whose rank r is very small, in $\tilde{O}(n^2 + r^3)$ time, and requiring only $O\left(r^2\right)$ storage, where the notation $\tilde{O}$ means that logarithmic factors are ignored.

Lastly, when S has both large dimensions and large rank, then the computation will most likely fail: computing the (dense) PLUQ of S will not be feasible.

3 STRUCTURAL PIVOTS SEARCH

Sturctural pivots finding algorithms are combinatorial by nature, since they ignore the values of non-zero entries in the matrix. As such, the matrix is often associated with a sparse graph. When the matrix is symmetric, then it can be seen as the *adjacency matrix* of an undirected graph. When it is square but unsymmetric, it can be seen as the adjacency matrix of a directed graph. Finally, when it is not square, it can be seen as the *occurence matrix* of an undirected bipartite graph: $A_{ij} \neq 0$ means that there is an edge between the row node R_i and the column node C_j.

3.1 Pivots and Matchings

A row (resp. a column) in which a pivot has been selected is a *pivotal row* (resp. column). Because two pivots cannot be on the same row or column, a set of pivots induces a *matching* on the bipartite graph. Recall that a matching is a set of edges without common vertices. This holds true for any set of pivots, be they chosen *a priori* or during the factorization.

As such, the size of a maximal matching gives an upper-bound on the rank of a sparse matrix. This bound is however not tight. Consider the matrix:

$$\begin{pmatrix} \textcircled{1} & 1 & 1 \\ 1 & \textcircled{1} & 1 \end{pmatrix}$$

The two circled entries form a maximal matching between rows and column, yet the matrix has rank 1. The elimination of the first entry makes the second candidate pivot disappear.

In terms of matching, the definition of structural pivots implies the absence of *alternating cycles* on the graph, with respect to the matching defined by the pivots (a cycle is alternating if, for every two consecutive edges, one is in the matching, cf. fig. 2). A matching that does not induce alternating cycles is said to be *uniquely restricted* [17], because it is the single matching between these vertices in the graph.

Theorem 3.1. *There exist row and column permutations P and Q such that the $k \times k$ principal submatrix of PAQ is upper-triangular if and only if there exist a uniquely restricted matching of size k in the bipartite graph associated to A.*

This theorem is not original (it is already mentionned in [17, 19]). We will prove it again for completeness. In alternating paths, every other edge belongs to the matching. Therefore, an alternating path is fully specified by its extremities and its non-matching edges. This "path compression" is useful both in theory and in practice.

Given a matching $\mathcal{M} = \{(r_1, c_1), \ldots, (r_k, c_k)\}$ over a bipartite graph A, we form the directed graph $G_{\mathcal{M}}$ as follows: its vertices are $\{1, \ldots, k\}$ and $i \xrightarrow{G} j$ if and only if $r_i \xleftrightarrow{A} c_j$. The adjacency matrix of $G_{\mathcal{M}}$ is formed by extracting rows $r_1, \ldots, r_k$ and columns $c_1, \ldots, c_k$ of the occurence matrix of A.

The point is that there is a one-to-one correspondance between alternating paths of A w.r.t $\mathcal{M}$ starting by a row vertex and in which all vertices are matched on the one hand, and (directed) paths of $G_{\mathcal{M}}$ on the other hand.

- Let $u_1 \to \cdots \to u_k$ be a directed path in $G_{\mathcal{M}}$. Then $r_{u_1} \leftrightarrow c_{u_2} \leftrightarrow r_{u_2} \leftrightarrow c_{u_3} \leftrightarrow \cdots \leftrightarrow c_{u_k}$ is an alternating path in A: the edges $r_{u_i} \leftrightarrow c_{u_i}$ belong to the matching, and the edges $r_{u_i} \leftrightarrow c_{u_{i+1}}$ exist in A by definition of $G_{\mathcal{M}}$.
- Let $R_1 \leftrightarrow C_2 \leftrightarrow R_2 \leftrightarrow \ldots \leftrightarrow C_\ell$ be an alternating path in A. We assume, without loss of generality, that $\{R_i, C_i\} \in \mathcal{M}$ for all i. This means that there is a sequence of indices (u_i) such that the path can be written $r_{u_0} \leftrightarrow c_{u_1} \leftrightarrow r_{u_1} \leftrightarrow \cdots \leftrightarrow c_{u_\ell}$. We ignore the edges belonging to the matching: the edges $r_{u_i} \leftrightarrow c_{u_{i+1}}$ connect two matched vertices in A, but they do not belong to the matching. This implies that there are edges $u_i \xrightarrow{G} u_{i+1}$ in $G_{\mathcal{M}}$, forming a directed path.

This correspondance makes it easy to prove theorem 3.1.

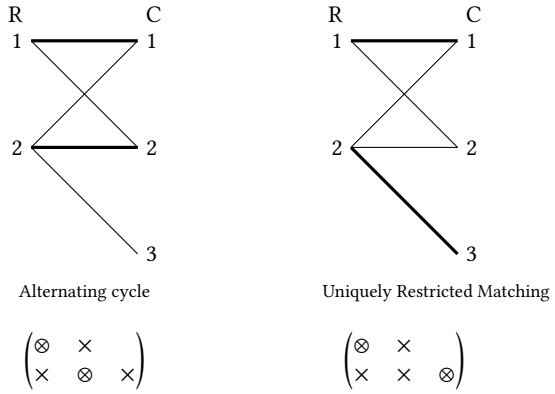


Figure 2: Matchings in bipartite graphs

Proof of theorem 3.1. If PAQ has an upper-triangular $k \times k$ principal submatrix, then we form the matching

$$\mathcal{M} = \{\{P(i), Q(i)\} \mid 1 \le i \le k\}.$$

The associated directed graph $G_{\mathcal{M}}$ is acyclic: its adjacency matrix is the $k \times k$ upper-triangular principal submatrix of PAQ. There are no alternating in A: by the correspondance above, this would imply a cycle in $G_{\mathcal{M}}$.

Conversely, let $\mathcal{M} = \{(r_1, c_1), \ldots, (r_k, c_k)\}$ be a uniquely restricted matching on A. By the correspondance above, this implies that the associated directed graph $G_{\mathcal{M}}$ is acyclic. Let M denote its adjacency matrix (it is a submatrix of A, as argued above). $G_{\mathcal{M}}$ can be topologically sorted: with a simple DFS, we find a permutation P of its vertices such that PM^tP is upper-triangular. This is enough for our purpose: we build row and column permutations P' and Q' such that the first k rows (resp. columns) are $r_{P(1)}, \ldots, r_{P(k)}$ (resp. $c_{P(1)}, \ldots, c_{P(k)}$). It follows that the principal $k \times k$ submatrix of $P'A^tQ'$ is PM^tP, which is upper-triangular. □

Computational Aspects. To our knowledge the interest in maximum matching without alternating cycle first appears in [6] in connection with the *Jump-number* problem for partially ordered set. The size of such a matching is shown to equal the *number of steps* in a bipartite DAG and is directly connected to the number of arcs to add to create an hamiltonian path but no circuits. Later in [26] the maximum alternating cycle-free matching problem is shown to be polynomial for —bipartite or not— distance hereditary graphs but NP-hard for chordal bipartite graphs and strongly chordal split graphs of diameter 2.

The problem was then introduced in [17] under the name Maximum Uniquely Restricted Matching problem, following the work of [19]. Recognition algorithms and NP-completness results for general and bipartite and some other special classes of graphs were given. In [18] this work is pursued for sub-classes of interval graphs with polynomial time algorithms. Recently a polynomial time algorithm to solve the problem in interval graphs was presented in [13] as well as linear time algorithms for proper interval graphs and bipartite permutation graphs.

Part of the work related to Uniquely Restricted Matchings concerns structural properties of graphs or characterization of classes of graphs having some special properties. A characterization of unicycle graphs in which a maximum matching and a maximum uniquely restricted matching have the same size is given in [22]. In [20, 21, 23] links between UR Matching and Local Maximum Stable Set greedoids are investigated while [16] explores the properties of generalized subgraph-restricted matchings to which belongs UR Matchings. A characterization of graphs in which every maximum matching is uniquely restricted is proposed in [8].

The approximability properties of the URM problem in bipartite graphs are studied in [25]. This problem is APX-complete in bipartite graphs of degree at most 3, but is not approximable within a factor of $O(n^{\frac{1}{3}-\varepsilon})$ $\forall \varepsilon > 0$ unless NP = ZPP. The 2-approximation given for 3-regular bipartite graphs is improved in [2] with a $\frac{9}{5}$-approximation algorithm for subcubic bipartite graphs.

These approximations cannot be used directly in our context, where the bipartite graphs induced by sparse matrices need not have degree at most three.

3.2 The Faugère-Lachartre Algorithm

We now discuss several heuristic algorithms to find structural pivots. The Faugère-Lachartre (FL) algorithm selects the first non-zero coefficient of each row, if it is not already under another pivot (in the example given by equation 1.1, the first three pivots of the matrix are of this kind). This can be done in $O(|A|)$.

This simple procedure was applied in the particular case of matrices originating from Gröbner basis computation. These matrices are nearly triangular and it follows that nearly 99% of the pivots can be identified *a priori* using this method.

The matrix can be permuted to make the principal submatrix triangular by pushing pivotal rows to the top, pivotal columns to the left, and finally sorting pivotal rows according to the column index of their leftmost non-zero entry.

The FL strategy is not restricted to Gröbner basis matrices. We used it to compute the rank of some large sparse matrices [3]. It often finds a large number of structural pivots when the matrix is mostly triangular.

In certain cases, it can be interesting to permute the columns of the matrix first, to increase the number of structural pivots found by the FL algorithm. For instance, in the cases of the GL7d matrices from Dumas's collection, we figured out that flipping them horizontally (permuting the first column with the last column, the second first with the second last, and so) allow us to find up to 50% more *a priori* pivots with the FL algorithm. This is for instance the case on the GL7d19 matrix shown fig. 3.

3.3 A quasi-linear Heuristic Algorithm

This algorithm is based on the works of [5] and [28] which are related to connectivity and ear decomposition of graphs. They use depth-first search respectively to compute st-orderings in 2-connected graphs and test 2- and 2-edge-connectivity. Our algorithm works as follows:

(1) Compute a spanning tree $\mathcal{T}$ of the input graph.
(2) Let A' be the subgraph formed by the edges not in $\mathcal{T}$.
(3) Let $\mathcal{I}$ be an Independent Set of A'.

Figure 3: Pattern of the GL7d19 matrix. The Faugère-Lachartre heuristic finds much more pivots after a horizontal flip.

(4) Let $\mathcal{T}'$ be the subgraph of $\mathcal{T}$ induced by $\mathcal{I}$.
(5) Compute a maximum matching $\mathcal{M}$ on $\mathcal{T}'$.

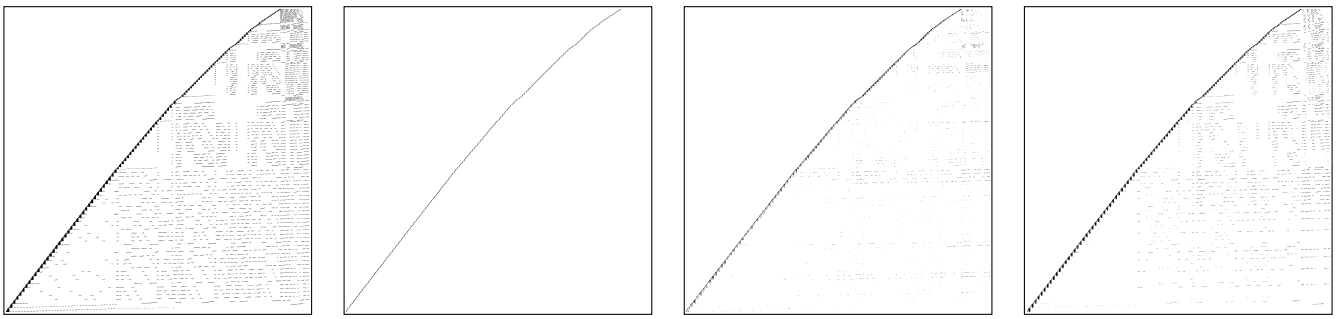
This procedure yields a UR matching of A. Indeed, consider a cycle in A. It cannot be contained inside $\mathcal{T}$, because $\mathcal{T}$ is a tree. Therefore, there is an edge $u \leftrightarrow v$ of the cycle in A'. Because $\mathcal{I}$ is an independent set of A', it cannot contain both u and v. This implies that u and v cannot both be matched in $\mathcal{M}$. This shows that the cycle cannot be alternating (all its vertices would have to be matched).

Computing the largest possible Independent Set is NP-hard, and therefore we used the following greedy algorithm: start with an empty $\mathcal{I}$; If there remain more row vertices than column vertices, add the row vertex of smallest degree to $\mathcal{I}$ and remove it from the graph. Otherwise, do the same with the column vertex of smallest degree. This can be implemented in time $O(n \log |A|)$ using two binary heaps to store remaining vertices of each kind.

There are cases where this strategy outperforms the Faugère-Lachartre heuristic (see fig. 4). However, in most cases it yields disappointingly bad results. We believe that it might only work well on very sparse matrices where $\mathcal{T}$ contains a non-negligible fraction of all non-zero entries. Indeed, note that $|\mathcal{T}| = n + m - 1$, while $|A'| = |A| - n - m + 1$.

3.4 A Simple Greedy Algorithm

We had more success with a simple algorithm that starts with an empty UR matching and grows it in a greedy way: examine each candidate edge $r_i \leftrightarrow c_j$, where r_i and c_j are unmatched rows/columns; check whether adding it to the matching would create an alternating cycle; if not, add it to the matching.



Original matrix
rank 168310

Faugère-Lachartre (section 3.2)
65536 pivots

Tree-based (section 3.3)
73492 pivots

Greedy (section 3.4)
160740 pivots

Figure 4: The `Homology/shar_te.b3` matrix, and the structural pivots found by each algorithm.

Adding (r_i, c_j) to the matching creates an alternating cycle if and only if there is already an alternating path $r_i \leftrightarrow C \leftrightarrow R \leftrightarrow \cdots \leftrightarrow c_j$ in A w.r.t. $\mathcal{M}$. Because the the path is alternating, then R is the row matched to C. This makes it possible to check all candidate non-zero entries of the i-th row of A as follows :

(G1) [Initialization] For each non-zero A_{ij}, if column j is matched, then enqueue j, else mark j as "candidate".
(G2) [BFS] While the queue is not empty, remove its first element k. If column k is not matched, go back to step (G2). Otherwise, let u be the row matched to k. For each non-zero A_{uv}, if v is not marked as "visited", then mark v as "visited" and add v to the queue.
(G3) [Detection] For each non-zero A_{ij}, if j is still marked as "candidate", then add it to the matching and go to step (G4).
(G4) [Cleanup] For each j that has been enqueued, and each non-zero A_{ij}, remove the mark on j.

The worst case complexity of the method is $O(n|A|)$, which is the same complexity as the Wiedemann algorithm. However, we observed that, in practice, this is much faster. Our sequential implementation terminates in a few hours on our largest benchmark, versus many expected months for Wiedemann.

Most of the time is spent in the BFS step. We used the following early-abort strategy to speed it up: in step 1, we count the number of candidates. During the BFS, each time we mark a candidate as "visited", we decrement this counter. If it reaches zero, we jump to step 4. On our collection of benchmarks, this gives a noticeable improvement.

Note that this procedure can be started with a non-empty matching. In fact, one can first find a set of structural pivots using another technique (*e.g.* the F.-L. algorithm), and then enlarge the matching using this greedy algorithm. As a matter of fact, we figured out that the following strategy yields good results in many cases:

(1) Find the F.-L. pivots, and add them to the matching.
(2) For each non-pivotal column, if the upmost non-zero coefficient is on a non-pivotal row, add it to the matching.
(3) Complete the matching using the greedy procedure above

Case (2) is a particular case of the general algorithm given above. In the matrix from example 1.1, the first three pivots are found performing step 1 of the previous strategy. The fourth is selected performing step 2. The last one is selected performing step 3. We represent the candidates on the last row by black bullets. As it is shown below, none of them can be selected.

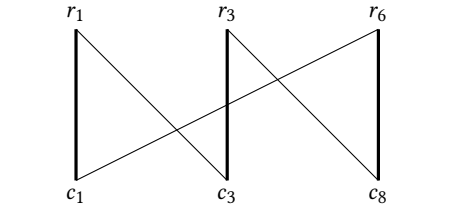
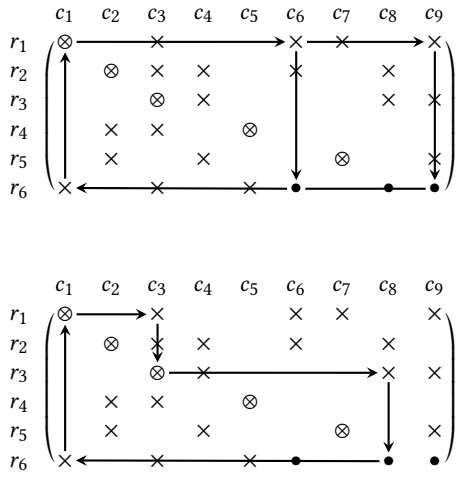


Figure 5: Example of a "bad" candidate, alternating cycle appears



In all the cases, an alternative cycle would appear. For instance, figure 5, gives an illustration of the alternating cycle obtained choosing candidate (r_6, c_8).

This greedy algorithm is capable of finding 99.9% of all the pivots in some matrices, when the other strategies we tried were not as good.

4 PARALLEL IMPLEMENTATION

Because structural pivot search dominates the running time of our rank computations, we chose to parallelize the greedy algorithm of section 3.4. This raises interesting problems.

Data Structures. We store sparse matrices using the Compressed Sparse Row (CSR) format. This means that given i, we can efficiently iterate over the set of non-zero A_{ij}. This is all that is needed to implement our greedy algorithm.

Rows and column indices are stored using `int` variables. Storing the matching requires an array `qinv` of m integer: `qinv[j]` gives the row matched to column j, or -1 if column j is not matched. Marking is done with an array of m bytes (only 2 bits would be necessary, because only 3 marks are used), with `mark[j]` containing the state of column j.

The queue is another array `queue` of m integers, along with two integer `first` (incremented when dequeuing) and `last` (incremented when enqueuing), both initially zero. The queue is empty when `first` = `last`. Re-walking the queue is then very easy: just read `queue` from 0 to `last`.

Parallelization Strategy. The basic approach is to process k rows on k threads simultaneously. The description of the matching `qinv` is shared between threads, as well as `npiv`, an integer counting the size of the matching.

The problem is that every time a thread adds an edge to the matching, new alternating paths are created in the graphs. Depending on the timing of operations, these paths may or may not be explored by concurrent BFS searches. More precisely, what can go wrong is that a thread T_1 finds that column k is not matched in the BFS step, and ignores the matched row. Immediately after, another thread T_2 matches column k to some row. Unaware of that, T_1 chooses an entry that creates an alternating cycle. This is illustrated by the example bellow.

$$\begin{array}{l} \\ T_1 \rightarrow \\ T_2 \rightarrow \end{array} \begin{pmatrix} \otimes & & \times & & & \\ & \otimes & & \times & & \\ & \times & \boxtimes & \times & \times & \\ \times & & \times & & \otimes & \times \end{pmatrix}$$

Simple Transactional Approach. To avoid this problem, and to avoid expensive synchronizations between threads, we used an optimistic approach based on transactions, as found in concurrent databases.

In its simplest form, it works by processing each row, using the algorithm of section 3.4, in a transaction. Either the matching is not modified during the lifespan of the transaction, and it succeeds. If the matching has been modified, then the transaction rollbacks and has to be restarted. More often than not, examining a row does not reveal any new pivot, so that rollbacks are relatively rare. The redundant work that may occur is more than compensated by the nearly complete elimination of synchronizations. The procedure executed by each thread to process a row i is:

(1) [Start transaction] Copy `npiv` to a local variable `npiv'`.
(2) [Sequential process] Process row i. If no pivot has been found, stop.
(3) [Commit] Let A_{ij} be a potential pivot. Enter a critical section. If `npiv` > `npiv'`, set `bad` to `True`. Otherwise, increment `npiv`, and set `qinv[j]` to i. Exit the critical section. If not `bad`, stop.
(4) [Rollback] Restart.

The goal of this strategy is to minimize time spent in the critical section. In practice, we observed that time spent waiting to enter the critical section is negligible.

Refined Transactional Approach. It is possible to eliminate the potential redundant work created by rollbacks. To this end, we add another shared array of m integers, `journal`, such that `journal[k]` is the column of the k-th edge added to the matching.

When thread tries to commit a new pivot and fails, then it has "missed" pivots on columns:

`journal[npiv']`, ..., `journal[npiv-1]`.

The corresponding rows are given by `qinv`. For each such column j, several situations may occur:

- If j was not marked during the BFS, it can safely be ignored.
- If a pivot has been found on column j, then step (G3) has to be reattempted.
- If column j was marked "visited" then j must be re-enqueued and step (G2) must be restarted.

This eliminates almost all the extra work caused by rollbacks, while keeping the advantages.

Lock-Free Implemenation. Pushing things to the extreme, it is possible to implement both transactional strategies without locks nor critical sections, if a compare-and-swap (CAS) hardware instruction is available (the CMPXCHG instruction is available on x86 processors since the 1990's). The C11 standard specifies that this functionnality is offered by the `atomic_compare_exchange_strong` function of `<stdatomic.h>`. The `CAS(A,B,C)` instruction performs the following operation atomically: if $A = B$, then set $A \leftarrow C$ and return `True`, else return `False`.

If the journal is a linked list, then it can be updated atomically using a CAS. Each cell contain the column index of a pivot and a pointer to the next cell. `journal` is then a pointer to the first cell (or `NULL` initially). The simple strategy is then implemented as follows:

(1) [Start] Copy `journal` to a local variable `journal'`.
(2) [Sequential process] Process row i. If no pivot has been found, stop.
(3) [Commit] Let A_{ij} be a potential pivot. Allocate a new linked-list entry `new_journal` $\leftarrow$ (j, `journal'`). Set `OK` to `CAS(journal, journal', new_journal)`. If `OK`, set `qinv[j]` to i and stop.
(4) [Rollback] Walk the pivot linked list from `journal` to `journal'`; for each pivot (i, j), set `qinv[j]` $\leftarrow$ `i`. Free `new_journal` and go back to (1).

When a transaction rollbacks, the "missed" pivots are those found by walking the `journal` list until the next link points to `journal'`. This allows to implement the "refined" strategy easily. It is necessary to perform the (potentially redundant) update of `qinv` in step 4: another thread could have been interrupted in step 3 after updating `journal` but before setting `qinv[j]`.

GL7/GL7dk	$\|A\|$	n	m	rank
15	6,080,381	460,261	171,375	132,043
16	14,488,881	955,128	460,261	328,218
17	25,978,098	1,548,650	955,128	626,910
18	35,590,540	1,955,309	1,548,650	921,740
19	37,322,725	1,911,130	1,955,309	1,033,568
20	29,893,084	1,437,547	1,911,130	877,562
21	18,174,775	822,922	1,437,547	559,985
22	8,251,000	349,443	822,922	262,937

Table 1: Benchmark matrices

We did not implement this lock-free strategy, because the critical section did not appear to be a bottleneck. However, we have shown that it is possible to get rid of it.

5 EXPERIMENTS AND RESULTS

The algorithms described in this paper have been implemented in C (using OpenMP) and assembled in a small library called SpaSM. We focused on rank computation, because the output is a single `int`; it nevertheless requires a complete PLUQ factorization. Our code computing the rank modulo a word-size prime p in parallel can be reduced to a single C file of 1200 lines of code (including IO).

We used the matrices from algebraic K-theory [9] as benchmarks (GL7/GL7dk, where k ranges between 10 and 26). They are amongst the largest in Dumas's sparse matrix collection [11], and computing their rank is challenging. It was only feasible using iterative methods to this point — and it required many days. Table 1 gives standard information about them.

According to [9], for the considered matrices, the sequential running time of the Block-Wiedemann algorithm is estimated between 67 hours (GL7d15) and 322 days (GL7d19, the hardest case). A parallel implementation of the Block-Wiedemann algorithm on a SGI Altix 370 with 64 Itanium2 processors allowed the authors of [9] to compute the rank of GL7d15 in 2.25 hours using 30 processors and to compute the rank of GL7d19 in 35 days using 50 processors in parallel.

We tried to check how these results held the test of time by running the sequential computation using LinBox (sequential Wiedemann algorithm) on a single core of machine A. The rank of GL7d15 was found in 3 hours. We could not compute the rank of GL7d19: unfortunately, the cluster manager killed our job after only 16 days.

In strong contrast to these numbers, our code computed the rank of GL7d19 in 10 seconds and the rank of GL7d19 in 4 minutes (wall-clock time) using 36 cores (machine C, see below).

We benchmarked our implementation on three parallel machines[1] with different characteristics, summarized in table 2. They all contain Intel Xeon E5-xxxx CPUs with varying number of cores and amount of L3 cache. We focused performance measurement on the structural pivot selection phase, because its running time is dominating in practice, and because the way we compute the rank of the dense schur complement is rather naive.

[1] A belongs to the university of Lille-1's cluster; B was kindly made available to us by the ANR HPAC project and is located in the university Grenoble-Alpes; C is used for teaching HPC at the Paris-6 university

Name	CPU Type	# CPU	cores/CPU	L3 Cache/CPU
A	Xeon E5-2670 v3	2×	12	30MB
B	Xeon E5-4620	4×	8	16MB
C	Xeon E5-2695 v4	2×	18	45MB

Table 2: Benchmark Machines Specifications

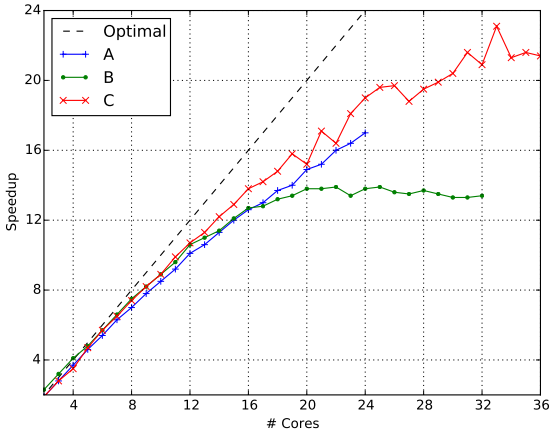


Figure 6: Structural pivot search, GL7d19

Table 3 shows the number of structural pivot found using the FL heuristic and the greedy algorithm of section 3.4, as well as the proportion of the rank that this represent (*i.e.* $100 \times$ #pivots/rank).

In order to maximize the number of pivots found by the FL heuristic, we first flipped the matrices horizontally (we swap the first and the last column, the second first and the second last,...). For the last matrices (GL7dk, with $k \geq 20$), we also transposed them as we figured out that this yields better results.

The table 4 gives the performances of the greedy algorithm. For each machine A, B, and C, the first column gives the running time (in seconds) of the algorithm using only one core, the second column gives the running time of the algorithm using all available cores. Finally the last column gives the speedup obtain using parallelization (running time using one core / running time using all cores).

The most interesting case is the largest matrix, GL7d19, whose rank computation is the toughest. It is the matrix where our parallel implementation scales worst (fig. 6). With the smaller GL7d16, on the other hand, we observe near-perfect scalability (fig. 7).

Figures 6 and 7 show that, given one matrix, our parallel implementation does not scale the same way on every benchmark machine. In fact, we noticed that our implementation does not scale very well on B, reaching a parallel efficiency (speedup/ #cores) of 63.01% in average using 32 cores. In fact, for the worst case (GL7d19) the parallel efficiency is 41.91%. On the other hand, the implementation scales better on C, where we reach a parallel efficiency of

GL7dk	# FL pivots	% of the rank	# greedy pivots	% of the rank
15	128,452	97.28	132,002	99.97
16	317,348	96.69	328,079	99.96
17	602,436	96.10	626,555	99.94
18	879,241	95.39	920,958	99.92
19	980,174	94.83	1,032,419	99.89
20	817,944	93.21	877,317	99.97
21	522,534	93.31	559,784	99.96
22	246250	93.65	262817	99.95

Table 3: Number of structural pivots found using the FL heuristic and using the new greedy algorithm

	Machine A			Machine B			Machine C		
GL7dk	1 core (s)	24 cores (s)	speedup	1 core (s)	32 cores (s)	speedup	1 core (s)	36 cores (s)	speedup
15	18	1	15.2	26	1	26.0	22	1	26.9
16	148	7	20.0	301	13	23.9	180	5	33.4
17	687	36	18.9	1502	105	14.4	897	29	30.9
18	2206	123	17.9	5649	388	14.6	2594	94	27.7
19	3774	222	17.0	9284	692	13.4	4212	197	21.4
20	1272	72	17.8	3657	197	18.5	1483	56	26.3
21	359	18	19.5	886	36	24.5	446	14	31.7
22	49	3	18.8	68	3	26.2	57	2	30.1

Table 4: Parallel efficiency of the greedy algorithm

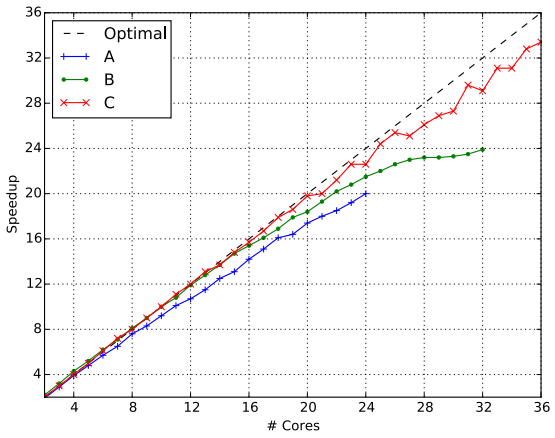


Figure 7: Structural pivot search, GL7d16

79.28% in average using 36 cores, and which can be up to 92.81% in the best case (GL7d16).

This phenomenon may be due to the size of the L3 Cache. The qinv array describing the matching is accessed frequently, in an unpredictable manner, by all the threads. Each thread also access frequently and unpredictably its own mark array (the queue of each thread, on the other hand, is accessed with good spatial and temporal locality).

We thus expect the L3 cache of each CPU to contain a copy of qinv plus one mark per thread. When k threads run on each CPU, this amount of data requires $(4 + k)m$ bytes.

This suggests that a strong performance degradation is likely to occur when this quantity goes beyond the size of the L3 cache. On machine B, for the GL7d19 matrix, this should happens when > 4 threads run on each CPU. And indeed, we observe a severe drop in parallel efficiency when using more than 16 threads on the 4 CPUs.

This might also explain why transposing the last matrices helps: they have much more columns than rows.

Lastly, this bottleneck could potentially be alleviated a little by using more efficient marks, using only 2 bits per column instead of 8. However, doing so would require more bit-twiddling, and its effect on the overall performance has yet to be evaluated.

6 CONCLUSION AND FUTURE WORK

We have designed and implemented an efficient algorithm for structural pivot selection in sparse matrices. This is turn allowed us to compute the rank of several large sparse matrices with a direct PLUQ factorization, in a tiny fraction of the time that was needed before using iterative methods. We have also demonstrated the parallel scalability of our implementation.

In conclusion, we feel compelled to highlight the limitations of our algorithms. They will not always work efficiently, and they will fail badly on matrices where fill-in is unavoidable.

Our experience suggests that a large number of structural pivot can often be found in matrices that are somewhat triangular, such as the GL7dk matrices. In this case, the algorithm we present are

more likely to be able to exploit this structure to allow much faster operations than the "slow-but-sure" iterative methods.

Acknowledgment. The first two authors are funded by the ANR BRUTUS project.

REFERENCES

[1] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. 1996. An Approximate Minimum Degree Ordering Algorithm. *SIAM J. Matrix Analysis Applications* 17, 4 (1996), 886–905. DOI: https://doi.org/10.1137/S0895479894278952

[2] Julien Baste, Dieter Rautenbach, and Ignasi Sau. 2016. Uniquely restricted matchings and edge colorings. *CoRR* abs/1611.06815 (2016).

[3] Charles Bouillaguet and Claire Delaplace. 2016. Sparse Gaussian Elimination modulo p: an Update. In *International Workshop on Computer Algebra in Scientific Computing*. Springer, 101–116.

[4] Brice Boyer, Christian Eder, Jean-Charles Faugère, Sylvian Lachartre, and Fayssal Martani. 2016. GBLA - Gröbner Basis Linear Algebra Package. *CoRR* abs/1602.06097 (2016). http://arxiv.org/abs/1602.06097

[5] Ulrik Brandes. 2002. Eager st-Ordering. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA '02)*. Springer-Verlag, London, UK, UK, 247–256. http://dl.acm.org/citation.cfm?id=647912.740674

[6] G. Chaty and M. Chein. 1979. Ordered matchings and matchings without alternating cycles in bipartite graphs. *Utilitas Mathematica* 16 (January 1979), 183 – 187.

[7] Don Coppersmith. 1994. Solving homogeneous linear equations over $\mathbb{F}_2$ via block Wiedemann algorithm. *Math. Comp.* 62, 205 (1994), 333–350.

[8] L. Draque Penso, D. Rautenbach, and U. dos Santos Souza. 2015. Graphs in which some and every maximum matching is uniquely restricted. *ArXiv e-prints* (April 2015). arXiv:math.CO/1504.02250

[9] Jean-Guillaume Dumas, Philippe Elbaz-Vincent, Pascal Giorgi, and Anna Urbanska. 2007. Parallel computation of the rank of large sparse matrices from algebraic K-theory. In *Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007, University of Western Ontario, London, Ontario, Canada*. 43–52.

[10] Jean-Guillaume Dumas and Gilles Villard. 2002. Computing the rank of sparse matrices over finite fields. In *CASC'2002, Proceedings of the fifth International Workshop on Computer Algebra in Scientific Computing, Yalta, Ukraine*, Victor G. Ganzha, Ernst W. Mayr, and Evgenii V. Vorozhtsov (Eds.). Technische Universität München, Germany, 47–62. http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/Publications/sparseeliminationCASC2002.pdf

[11] J.-G. Dumas. Sparse Integer Matrices Collection. (????). http://hpac.imag.fr.

[12] J.-C. Faugère and Sylvain Lachartre. 2010. Parallel Gaussian elimination for Gröbner bases computations in finite fields. In *PASCO*, Marc Moreno Maza and Jean-Louis Roch (Eds.). ACM, 89–97.

[13] Mathew C. Francis, Dalu Jacob, and Satyabrata Jana. 2016. Uniquely Restricted Matchings in Interval Graphs. *CoRR* abs/1604.07016 (2016).

[14] Alan George. 1973. Nested Dissection of a Regular Finite Element Mesh. 10, 2 (April 1973), 345–363.

[15] John R. Gilbert and Tim Peierls. 1988. Sparse Partial Pivoting in Time Proportional to Arithmetic Operations. *SIAM J. Sci. Statist. Comput.* 9, 5 (1988), 862–874. DOI: https://doi.org/10.1137/0909058 arXiv:http://dx.doi.org/10.1137/0909058

[16] Wayne Goddard, Sandra M. Hedetniemi, Stephen T. Hedetniemi, and Renu Laskar. 2005. Generalized subgraph-restricted matchings in graphs. *Discrete Mathematics* 293, 1–3 (2005), 129 – 138. DOI: https://doi.org/10.1016/j.disc.2004.08.027 19th British Combinatorial Conference19th British Combinatorial Conference.

[17] M. C. Golumbic, T. Hirst, and M. Lewenstein. 2001. Uniquely Restricted Matchings. *Algorithmica* 31, 2 (2001), 139–154. DOI: https://doi.org/10.1007/s00453-001-0004-z

[18] Swapnil Gupta and C. Pandu Rangan. 2016. Computing Maximum Uniquely Restricted Matchings in Restricted Interval Graphs. *International Journal of Computer, Electrical, Automation, Control and Information Engineering* 10, 6 (2016), 950 – 959. http://waset.org/Publications?p=114

[19] Daniel Hershkowitz and Hans Schneider. 1993. Ranks of zero patterns and sign patterns. *Linear and Multilinear Algebra* 34, 1 (1993), 3–19. DOI: https://doi.org/10.1080/03081089308818204 arXiv:http://dx.doi.org/10.1080/03081089308818204

[20] V.E. Levit and Eugen Mandrescu. 2007. Triangle-free graphs with uniquely restricted maximum matchings and their corresponding greedoids. *Discrete Applied Mathematics* 155, 18 (2007), 2414 – 2425. DOI: https://doi.org/10.1016/j.dam.2007.05.039

[21] Vadim E. Levit and Eugen Mandrescu. 2003. Local maximum stable sets in bipartite graphs with uniquely restricted maximum matchings. *Discrete Applied Mathematics* 132, 1–3 (2003), 163 – 174. DOI: https://doi.org/10.1016/S0166-218X(03)00398-6 Stability in Graphs and Related Topics.

[22] Vadim E. Levit and Eugen Mandrescu. 2005. Unicycle graphs and uniquely restricted maximum matchings. *Electronic Notes in Discrete Mathematics* 22 (2005), 261 – 265. DOI: https://doi.org/10.1016/j.endm.2005.06.055

[23] VADIM E. LEVIT and EUGEN MANDRESCU. 2011. VERY WELL-COVERED GRAPHS OF GIRTH AT LEAST FOUR AND LOCAL MAXIMUM STABLE SET GREEDOIDS. *Discrete Mathematics, Algorithms and Applications* 03, 02 (2011), 245–252. DOI: https://doi.org/10.1142/S1793830911001115 arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S1793830911001115

[24] Harry M. Markowitz. 1957. The Elimination Form of the Inverse and Its Application to Linear Programming. *Manage. Sci.* 3, 3 (April 1957), 255–269. DOI: https://doi.org/10.1287/mnsc.3.3.255

[25] Sounaka Mishra. 2011. On the Maximum Uniquely Restricted Matching for Bipartite Graphs. *Electronic Notes in Discrete Mathematics* 37 (2011), 345 – 350. DOI: https://doi.org/10.1016/j.endm.2011.05.059

[26] Haiko Müller. 1990. Alternating cycle-free matchings. *Order* 7, 1 (1990), 11–21. DOI: https://doi.org/10.1007/BF00383169

[27] B. David Saunders and Bryan S. Youse. 2009. Large Matrix, Small Rank. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation (ISSAC '09)*. ACM, New York, NY, USA, 317–324. DOI: https://doi.org/10.1145/1576702.1576746

[28] Jens M. Schmidt. 2013. A simple test on 2-vertex- and 2-edge-connectivity. *Inform. Process. Lett.* 113, 7 (2013), 241 – 244. DOI: https://doi.org/10.1016/j.ipl.2013.01.016

[29] The FFLAS-FFPACK group. 2014. *FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package* (v2.0.0 ed.). http://linalg.org/projects/fflas-ffpack.

[30] Douglas H. Wiedemann. 1986. Solving sparse linear equations over finite fields. *IEEE Trans. Information Theory* 32, 1 (1986), 54–62. DOI: https://doi.org/10.1109/TIT.1986.1057137

[31] Mihalis Yannakakis. 1981. Computing the Minimum Fill-In is NP-Complete. *SIAM Journal on Algebraic Discrete Methods* 2, 1 (1981), 77–79. DOI: https://doi.org/10.1137/0602010 arXiv:http://dx.doi.org/10.1137/0602010