Sparse Gaussian Elimination modulo p: an Update

Charles Bouillaguet, Claire Delaplace

Sparse Gaussian Elimination modulo p: an Update

Charles Bouillaguet[1] and Claire Delaplace[1,2]

[1] Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
charles.bouillaguet@univ-lille1.fr

[2] Université de Rennes-1 / IRISA
claire.delaplace@irisa.fr

Abstract. This paper considers elimination algorithms for sparse matrices over finite fields. We mostly focus on computing the rank, because it raises the same challenges as solving linear systems, while being slightly simpler.
We developed a new sparse elimination algorithm inspired by the Gilbert-Peierls sparse LU factorization, which is well-known in the numerical computation community. We benchmarked it against the usual right-looking sparse gaussian elimination and the Wiedemann algorithm using the Sparse Integer Matrix Collection of Jean-Guillaume Dumas.
We obtain large speedups (1000× and more) on many cases. In particular, we are able to compute the rank of several large sparse matrices in seconds or minutes, compared to days with previous methods.

1 Introduction

There are essentially two families of algorithms to perform the usual operations on sparse matrices (rank, determinant, solution of linear equations, etc.): *direct* and *iterative* methods.

Direct methods (such as gaussian elimination, LU factorization, etc.) generally produce an echelonized version of the original matrix. This process often incurs *fill-in*: the echelonized version has more non-zero entries than the original. Fill-in increases the time and space needed to complete the echelonization. As such, the time and space requirements of direct methods are usually unpredictable; they may fail if not enough storage is available, or become excruciatingly slow.

Iterative methods such as the Wiedemann algorithm [29] only perform matrix-vector products and only need to store one or two vectors in addition to the matrix. They do not incur any fill-in. On rank-r matrices, the number of matrix-vector products that must be performed is $2r$. Also, when a matrix M has $|M|$ non-zero entries, computing the matrix-vector product $\mathbf{x} \cdot M$ requires $\mathcal{O}(|M|)$ operations. The time complexity of iterative methods is thus $\mathcal{O}(r|M|)$, which is fairly easy to predict, and the space complexity is essentially that of keeping

the matrix in memory. These methods are often the only option for very large matrices, or for matrices where fill-in makes direct methods impractical.

In a paper from 2002, Dumas and Villard [15] surveyed and benchmarked algorithms dedicated to rank computations for sparse matrices modulo a small prime number p. In particular, they compared the efficiency of sparse gaussian elimination and the Wiedemann algorithm on a collection of benchmark matrices that they collected from other researchers and made available on the web [16]. They observed that while iterative methods are fail-safe and can be practical, direct methods can sometimes be much faster. This is in particular the case when matrices are almost triangular, so that gaussian elimination barely has anything to do.

It follows that both methods are worth trying. In practical situations, a possible workflow could be : "*try a direct method ; if there is too much fill-in, abort and restart with an iterative method*".

We concur with the authors of [15], and strengthen their conclusion by developing a new sparse elimination algorithm which outperforms all other techniques in some cases, including several large matrices which could only be processed by an iterative algorithm.

Lastly, it is well-known that both methods can be combined: performing one step of elimination reduces the size of the "remaining" matrix (the Schur complement) by one, while increasing the number of non-zeros. This may decrease the time complexity of running an iterative method on the Schur complement. This strategy can be implemented as follows: "*While the product of the number of remaining rows and remaining non-zeros decreases, perform an elimination step ; then switch to an iterative method*". For instance, one phase of the record-setting factorization of a 768-bit number [20] was to find a few vectors on the kernel of a 2 458 248 361 $\times$ 1 697 618 199 very sparse matrix over $\mathbb{F}_2$ (with about 30 non-zero per row). A first pass of elimination steps (and discarding "bad" rows) reduced this to a roughly square matrix of size 192 796 550 with about 144 non-zero per row. A parallel implementation of the block-Wiedemann [7] algorithm then finished the job. The algorithm presented in this paper lends itself well to this hybridization.

1.1 Our contribution

Our original intention was to check whether the conclusions of [15] could be refined by using more sophisticated sparse elimination techniques used in the numerical world. To do so, we developed the SpaSM software library (SPArse Solver Modulo p). Its code is publicly available in a repository hosted at:

`https://github.com/cbouilla/spasm`

Our code is heavily inspired by CSPARSE ("*A Concise Sparse Matrix Package in C*"), written by Davis and abundantly described in his book [9]. We modified it to work row-wise (as opposed to column-wise), and more importantly to deal with non-square or singular matrices. At its heart lies a sparse LU factorization

algorithm. It is capable of computing the rank of a matrix, but also of solving linear systems (and, with minor adaptations, of computing the determinant, finding a basis of the kernel, etc.).

We used this as a playground to implement the algorithms described in this paper —as well as some other, less successful ones. This was necessary to test their efficiency in practice. We benchmarked them using matrices from Dumas's collection [16], and compared them with the algorithms used in [15], which are publicly available inside the LinBox library. This includes a right-looking sparse gaussian elimination, and the Wiedemann algorithm.

There are several cases where the algorithm described in section 3 achieve a 1000× speedup compared to previous algorithms. In particular, it systematically outperforms the right-looking sparse gaussian elimination implemented in LinBox.

It is capable of computing the rank of several of the largest matrices from [16], where previous elimination algorithms failed. In these cases, it vastly outperforms the Wiedmann algorithm. In a striking example, two computations that required two days with the Wiedemann algorithm could be performed in 30 minutes and 30 seconds respectively using the new algorithm. More complete benchmark results are given in section 4.

We relied on three main ideas to obtain these results. First, we built upon GPLU [19], a left-looking sparse LU factorization algorithm. We then used a simple pivot-selection heuristic designed in the context of Gröbner basis computation [18], which works well with left-looking algorithms. On top of these two ideas, we designed a new, hybrid, left-and-right looking algorithm.

Our intention is not to develop a competitor sparse linear algebra library; we plan to contribute to LinBox, but we wanted to check the viability of our ideas first.

1.2 Related Work

Sparse Rank Computation mod p. Our starting point was [15], where a right-looking sparse gaussian elimination and the Wiedemann algorithm are compared on many benchmark matrices. [23,24] consider the problem of large dense matrices of small rank modulo a very small prime, while [25] shows that most operations on extremely sparse matrices with only two non-zero entries per row can be performed in time $\mathcal{O}(n)$. [17] discusses sparse rank computation of the largest matrices of our benchmark collection by various methods. The largest computation were performed with a parallel block-variant of the Wiedemann algorithm.

Direct Methods in the Numerical World. A large body of work has been dedicated to sparse direct methods by the numerical computation community. Direct sparse numerical solvers have been developped during the 1970's, so that several software packages were ready-to-use in the early 1980's (`MA28`,

SPARSPAK, YSMP, ...). For instance, `MA28` is an early "right-looking" (cf. section 2) sparse LU factorization code described in [12,13]. It uses Markowitz pivoting [22] to choose pivots in a way that maintains sparsity.

Most of these direct solvers start with a symbolic analysis phase that ignores the numerical values and just examines the pattern of the matrix. Its purpose is to predict the amount of fill-in that is likely to occur, and to pre-allocate data structures to hold the result of the numerical computation. The complexity of this step often dominated the running time of early sparse codes. In addition, an important step was to choose *a priori* an order in which the rows (or columns) were to be processed in order to maintain sparsity.

Sparse linear algebra often suffers from poor computational efficiency, because of irregular memory accesses and cache misses. More sophisticated direct solvers try to counter this by using *dense* linear algebra kernels (the BLAS and LAPACK) which have a much higher FLOP per second rate.

The supernodal method [6] does this by clustering together rows (or columns) with similar sparsity pattern, yielding the so-called *supernodes*, and processing them all at once using dense techniques. Modern supernodal codes include CHOLDMOD [5] (for Cholesky factorization) and SuperLU [11]. The former is used in Matlab on symmetric positive definite matrices.

In the same vein, the multifrontal method [14] turns the computation of a sparse LU factorization into several, hopefully small, *dense* LU factorizations. The starting point of this method is the observation that the elimination of a pivot creates a completely dense submatrix in the Schur complement. Contemporary implementations of this method are UMFPACK [8] and MUMPS [1]. The former is also used in Matlab in non-symmetric matrices.

Finally, "left-looking" algorithms are those that do not explicitly compute the Schur complement during the sparse factorization. This is for instance the case of SPARSPAK cited earlier. A very interesting algorithm, referred to as GPLU [19], computes an LU factorization in time proportionnal to the number of arithmetic operations needed to compute the product $L \times U$ (assuming the zeros are not stored). In particular, the symbolic part of the factorization does not dominate the numerical part asymptotically. This algorithm is implemented in Matlab, and is used for very sparse unsymmetric matrices. It is also the heart of the specialized library called KLU [10], dedicated to circuit simulation.

Direct Methods Modulo p. The world of exact sparse direct methods is much less populated. Besides LinBox, we are not aware of many implementations of sparse gaussian elimination capable of computing the rank of a matrix modulo p. According to its handbook, the MAGMA [2] computer algebra system computes the rank of a sparse matrix by first performing sparse gaussian elimination with Markowitz pivoting, then switching to a dense factorization when the matrix becomes dense enough. The Sage [28] system uses LinBox.

Some specific applications rely on exact sparse linear algebra. All competitive factoring and discrete logarithms algorithms work by finding a few vectors in the kernel of a large sparse matrix. Some controlled elimination steps are usually

performed, which makes the matrix smaller and denser. This has been called "structured gaussian elimination" [21]. The process can be continued until the matrix is fully dense (after which it is handled by a dense solver), or stopped earlier, when the resulting matrix is handled by an iterative algorithm. The current state-of-the-art factoring codes, such as CADO-NFS [26], seem to use the sparse elimination techniques described in [4].

Modern Gröbner basis algorithms work by computing the reduced row-echelon form of particular sparse matrices. An ad hoc algorithm has been designed, exploiting the mostly triangular structure of these matrices to find pivots without performing any computation [18]. An open-source implementation, GBLA [3], is available.

2 Sparse LU Factorization

In this section we discuss algorithms to compute a sparse LU factorization of a matrix A of size $n \times m$ over a finite field. The techniques described in this paper could in principle work over any field; for the sake of simplicity we focus on the case of integers modulo a prime p that fits into a machine integer.

2.1 Definitions and Notations

We recall here some usefull definitions. A is an n-by-m matrix and its (unknown) rank is denoted by r.

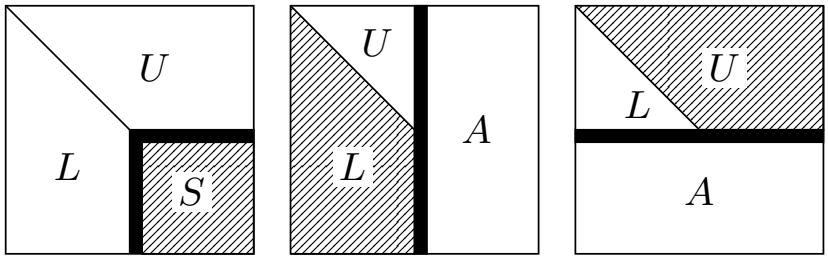
Because A is rectangular, we consider the $PLUQ$ factorization of A where L is n-by-r and lower-trapezoidal with a unit diagonal, U is r-by-m and upper-trapezoidal with a non-zero diagonal and P (resp. Q) is a permutation over the rows (resp. columns) of A. Computing a PLUQ factorization essentially amounts to performing gaussian elimination. We recall the usual (dense) algorithm: at each step i, choose a coefficient $a_{ik} \neq 0$ called the *pivot* and "eliminate" every coefficients under it by adding suitable multiples of row i to the rows below. This method is said to be *right-looking*, because at each step it accesses the data stored in the bottom-right of A (shaded aera in figure 1a). This contrasts with *left-looking* algorithms (or the *up-looking* row-by-row equivalent described in section 2.3), that accesses the data stored in the left of L (respectively in the top of U) represented by the shaded aera in figure 1b (1c).

If we assume that we have performed some steps of the PLUQ decomposition, then we have

$$PAQ = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix},$$

such that A_{00} is square, nonsingular and can be factored as $A_{00} = L_{00} \cdot U_{00}$. It leads to the following factorization of A :

$$PAQ = \begin{pmatrix} L_{00} & \\ L_{10} & I \end{pmatrix} \cdot \begin{pmatrix} U_{00} & U_{10} \\ & S_{11} \end{pmatrix},$$



(a) right-looking (b) left-looking (c) up-looking

Fig. 1: Data access pattern of several PLUQ factorization algorithms. The dark area is the echelonization front. While the algorithm processes, they only access data in the shaded area.

where $L_{10} = A_{10}U_{00}^{-1}$, $U_{01} = L_{00}^{-1}A_{01}$ and $S_{11} = A_{11} - A_{10}A_{00}^{-1}A_{01}$ is the *Schur Complement* of A_{11} in A. It is represented by S in figure 1a.

2.2 The Classical Right-Looking Algorithm

LinBox implements a sparse gaussian elimination that only computes U (and not L) which is a straightforward adaptation of the classical (dense) algorithm to sparse data structures: find a pivot, permute the rows and columns to move the pivot to the diagonal, eliminate all the entries below the pivot, repeat. In LinBox's gaussian elimination code, a sparse matrix is an array of sparse vectors; a sparse vector is a dynamic array of (column index, coefficient) pairs, sorted by column indices. This dynamic array is essentially a `std::vector` object from the C++ STL.

This is very similar to the early `MA28`, except that in `MA28`, sparse vectors are unordered. Also, `MA28` implements full Markowitz pivoting (greedily choosing the pivot that minimises fill-in in the Schur complement), while LinBox implements a faster relaxation thereof. Note that choosing the next pivot takes time $\Omega(n)$, so that the complexity of the whole process is lower-bounded by $\Omega(n^2)$. This is suboptimal in some cases (e.g. tridiagonal matrices).

2.3 The Left-Looking GPLU Algorithm.

The GPLU Algorithm was introduced by Gilbert and Peierls in 1988 [19]. It is also abundantly described in [9], up to implementation details. During the k-th step, the k-th column of L and U are computed from the k-th column of A and the previous columns of L. The algorithm thus only accesses data on the left of the echelonization front, and the algorithm is said to be "left-looking". It does not compute the Schur complement at each stage. We implemented a row-by-row (as opposed to column-by-column) variant of this method which is then "up-looking".

The main idea behind the algorithm is that the next row of L and U can both be computed by solving a triangular system. If we ignore row and column permutations, this can be derived from the following 3-by-3 block matrix expression :

$$\begin{pmatrix} L_{00} & & \\ \mathbf{l}_{10} & 1 & \\ L_{20} & \mathbf{l}_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{00} & \mathbf{u}_{01} & U_{02} \\ & u_{11} & \mathbf{u}_{12} \\ & & U_{22} \end{pmatrix} = \begin{pmatrix} A_{00} & \mathbf{a}_{01} & A_{02} \\ \mathbf{a}_{10} & a_{11} & \mathbf{a}_{12} \\ A_{20} & \mathbf{a}_{21} & A_{22} \end{pmatrix}.$$

Here $(\mathbf{a}_{10}\ a_{11}\ \mathbf{a}_{12})$ is the k-th row of A. L is assumed to have unit diagonal. Assume we have already computed $(U_{00}\ \mathbf{u}_{01}\ U_{02})$, the first k rows of U. Then we have:

$$\left(\mathbf{l}_{10}\ u_{11}\ \mathbf{u}_{12}\right) \cdot \begin{pmatrix} U_{00} & \mathbf{u}_{01} & U_{02} \\ & 1 & \\ & & Id \end{pmatrix} = \left(\mathbf{a}_{01}\ a_{11}\ \mathbf{a}_{21}\right). \tag{1}$$

Thus, the whole PLUQ factorization can be performed by solving a sequence of n sparse triangular systems.

Sparse Triangular Solving. To make the above idea work, we need to solve efficiently $\mathbf{x} \cdot U = \mathbf{b}$, where U is a sparse matrix stored by rows, and $\mathbf{b}$ is a sparse vector (a row of A). The main trick of the GPLU algorithm is that it is possible to determine the sparsity pattern of $\mathbf{x}$ without performing any kind of numerical computation.

Theorem 1 (Gilbert and Peierls [19]). *Define the directed graph $G_U = (V, E)$ with nodes $V = \{1 \dots m\}$ and edges $E = \{(i, j) | u_{ij} \neq 0\}$. Let $\mathrm{Reach}_U(i)$ denote the set of nodes reachable from i via paths in G_U and for a set $\mathcal{B}$, let $\mathrm{Reach}_U(\mathcal{B})$ be the set of all nodes reachable from any nodes in $\mathcal{B}$. The non-zero pattern $\mathcal{X} = \{j | x_j \neq 0\}$ of the solution $\mathbf{x}$ to the sparse linear system $\mathbf{x} \cdot U = \mathbf{b}$ is given by $\mathcal{X} = \mathrm{Reach}_U(\mathcal{B})$, where $B = \{i | b_i \neq 0\}$, assuming there is no numerical cancellation.*

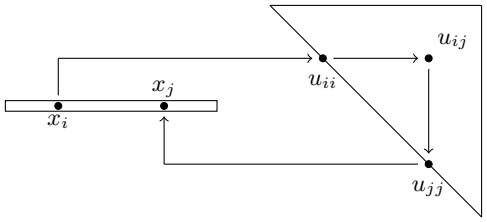


Fig. 2: The presence of x_i and of u_{ij} entails that of x_j in $\mathbf{x}$.

This is due to the fact that x_j can only be non-zero if either (1) b_j is non-zero, or (2) there is a node i such that $x_i \neq 0$ and $u_{ij} \neq 0$ as shown in figure 2.

Thus, $\mathcal{X}$ can be determined by performing a graph traversal in G_U starting from each node of $\mathcal{B}$. If we perform a *depth-first search*, then we will naturally obtain $\mathcal{X}$ sorted in *topological order*: if $u_{ij} \neq 0$, then i comes up before j in $\mathcal{X}$. This enables the following efficient procedure to compute $\mathbf{x}$:

1: Scatter $\mathbf{b}$ into a (dense) working array $\mathbf{w}$, initially filled with 0.
2: **for all** $i \in \mathcal{X}$ in topological order **do**
3: $\quad w_i \leftarrow w_i / u_{ii}$
4: $\quad$ **for all** $j > i$ such that $u_{ij} \neq 0$ **do**
5: $\quad\quad w_j \leftarrow w_j - w_i u_{ij}$
6: Gather $\mathbf{x}$ (get w_i for $i \in \mathcal{X}$), and reset $\mathbf{w}$.

A striking feature of this triangular solver is that it may terminate in *constant time* in some cases (for instance if U is bidiagonal, and $\mathbf{b}$ has only a constant number of entries). In addition, the complexity of finding $\mathcal{X}$ is of the same order as that of the numerical computation of $\mathbf{x}$ once $\mathcal{X}$ is known: each iteration of line 5 in the above pseudo-code corresponds to the crossing of an edge in G_U during the construction of $\mathcal{X}$. This holds well in our implementation: computing $\mathcal{X}$ consistenty requires 25% of the time needed to compute $\mathbf{x}$.

Selecting the Pivots. For the triangular systems to have a solution, we need to make sure that the diagonal coefficient of U (the pivots) are non-zero. However, while solving the system (1), we may find $u_{11} = 0$. There are two possible cases: (1) If $u_{11} = 0$ and $\mathbf{u}_{12} \neq \mathbf{0}$, then we can permute the column j_0 where u_{11} is with another one j_1, such that $j_0 < j_1$. (2) If $u_{11} = 0$ and $\mathbf{u}_{12} = 0$, then there is no way we can permute the columns of U to bring a non-zero coefficent on the diagonal. This means that the current row of A is a linear combinations of the previous rows. When the factorization is done, the number of non-empty rows of U is the rank r of the matrix A.

Useful Heuristics. Right-looking algorithms have the possibility to choose the next pivot by exploiting knowledge of the Schur complement (this is typically what Markowitz pivoting is about). On the other hand, the only information available to left-looking algorithms such as GPLU to choose pivots is the original matrix, and the part of U that has already been computed. It is possible to choose an *a priori* order in which to process the rows of the matrix in order to keep U as sparse as possible. Many different strategies have been designed to do so in the numerical world: the (approximate) minimum degree algorithm, nested dissection, etc. The problem is that these algorithms are mostly adapted to symmetric matrices, and a fortiori to square matrices. They are usually retrofitted to the unsymmetric case by applying them to $A^T A$, which is square and symmetric. However, when A is very rectangular, this becomes completely dense and provide little useful information.

We instead implemented a much simpler heuristic inspired from an echelonization procedure dedicated to Gröbner basis computation by Faugère and Lachartre [18]. We map each row to the column of its leftmost coefficient. When

several rows have the same leftmost coefficient, we select the sparsest row. We then move the selected rows before the others and sort them by increasing position of the left-most coefficient. As a result, the leftmost coefficient of each row cannot occur in any selected row below it. It follows that the selected rows are copied as-is into U, with zero fill-in. This is particularly well-suited to the GPLU algorithm, because this keeps U as sparse as possible; this in turn makes the triangular solver fast.

Also, we observed that it can be beneficial to compute the rank of the *transpose* of the matrix. Indeed, U has size $r \times m$, so when m is much larger than n, transposing the matrix before starting the computation ensures than U is smaller. More often than not this decreases the running time of the factorization (on the matrices from [16]). Lastly, we note that the running time of the right-looking algorithm implemented in LinBox is usually not the same on A and A^T.

We also implemented a simple early abort test which is performed when sufficiently many rows have been processed without finding any new pivot. A random linear combination of the remaining rows is computed; if it belongs to the row-space of U, then no new pivot will be found in the remaining rows with probability greater than $1 - 1/p$. This test can be repeated to increase its success probability exponentially.

2.4 Left or Right?

The two algorithms presented in this section are incomparable. We illustrate this by exhibiting two situations where each one consistently outperforms the other. We generated two random matrices with different properties (situations A and B in table 1). The entries of matrix A are identically and independently distributed. There are zero with probability 99%, and chosen uniformly at random modulo p otherwise. GPLU terminates very quickly in this case because it only process the first ≈ 1000 rows before stopping, having detected that the matrix has full column-rank.

In matrix B, one row over 1000 is random (as in matrix A), and the 999 remaining ones are sparse linear combination of a fixed set of 100 other random sparse rows (as in matrix A). The right-looking algorithm discovers these linear combinations early and quickly skip over empty rows in the Schur complement. On the other hand, GPLU has to work its way thoughout the whole matrix. Section 4 contains further "real" examples where an algorithm clearly outperforms the other.

Matrix	rows	columns	non-zero	rank	Right-Looking (LinBox)	GPLU (SpaSM)
A	100 000	1 000	995 076	1000	3.0	0.02
B	100 000	1 000	4 776 477	200	1.7	9.5

Table 1: Running time (in seconds) of the two algorithms on extreme cases.

3 A New Hybrid Algorithm

The right-looking algorithm enjoys a clear advantage in terms of pivot selection, because it has the possibility to explore the Schur complement. The short-term objective of pivot selection is to keep the Schur complement sparse.

In GPLU pivot selection has to be done "in the dark", with the short-term objective to keep U sparse (since this keeps the triangular solver fast). However, GPLU performs the elimination steps very efficiently.

In this section, we present a new algorithm that combines these two strong sides. It usually outperforms both the classical right-looking elimination and GPLU. The new algorithm works as follows:

1. Use the Faugère-Lachartre heuristic to find as many pivots as possible in A.
2. Compute the Schur complement S with respect to these pivots, using GPLU.
3. Compute the rank of S by any mean (including recursively).

Pivot Selection. As in the right-looking algorithm, we begin by exploring the matrix looking for pivots. However, instead of choosing only one, we try to pick as many as possible. For this, we use the Faugère-Lachartre heuristic described in section 2.3: it finds a sequence of k rows such that the leftmost coefficient of each rows does not appear in any subsequent row. It can thus be chosen as a pivot. Those specific rows are copied as-is into U without any arithmetic operation. The cost of this step is that of iterating over the entries of A.

Schur Complement. We then compute the Schur complement S with respect to the chosen pivots. This amounts to eliminating every entries below the chosen pivots. The left-looking side of the combination is that we use the GPLU algorithm to compute the Schur complement.

If we denote by P the permutation of the rows of A that "pushes" this well-chosen set of linearly independant rows at the top of A and if we ignore the permutation over the columns of A, the PLU factorization of A can be represented by the following 2-by-2 block matrix expression :

$$PA = \begin{pmatrix} U_{00} & U_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} Id & \\ L_{10} & L_{11} \end{pmatrix} \cdot \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix},$$

where $(U_{00}\ U_{01})$ is the part of U provided by the Faugère-Lachartre heuristic. What we need is $S = A_{11} - A_{10}U_{00}^{-1}U_{01}$, the Schur Complement of A with respect to U_{00}. We compute S row-by-row. To obtain the ith row $\mathbf{s}_i$ of S, denote by $(\mathbf{a}_{i0}\ \mathbf{a}_{i1})$ the ith row of $(A_{10}\ A_{11})$ and consider the following system:

$$(\mathbf{x}_0\ \mathbf{x}_1) \cdot \begin{pmatrix} U_{00} & U_{01} \\ & Id \end{pmatrix} = (\mathbf{a}_{i0}\ \mathbf{a}_{i1})$$

We get $\mathbf{x}_1 = \mathbf{a}_{i1} - \mathbf{x}_0 U_{01} = \mathbf{a}_{i1} - \mathbf{a}_{i0}U_{00}^{-1}U_{01}$. From the definition of S, it follows that $\mathbf{x}_1 = \mathbf{s}_i$. Thus S can be computed by a sequence of sparse triangular solve. Because we chose U to be as sparse as possible, this process is fast. It can also be parallelized efficiently: all the rows of S can be computed independently.

Computing the Rank of S. Once the Schur complement S has been computed, it remains to find its rank. Several options are possible. If S is sparse, we can either use the same technique recursively, or switch to GPLU, or switch to the Wiedemann algorithm. If S is small and dense, we switch to dense gaussian elimination. If S is big and dense, we abort and report failure.

In the first case, where several options are possible, some guesswork is required to find the best course of action. By default, we found that allowing only a limited number of recursive calls (usually less than 10, often 3) and then switching to GPLU yields good results.

Tall and Narrow Schur Complement. It is beneficial to consider a special case in the above procedure, when S has much more rows than columns (we transpose S if it has much more columns than rows). This happens in particular in the favorable situation where the Faugère-Lachartre heuristic finds almost all possible pivots, and only very few remain to be found.

This situation can be detected as soon as the pivots have been selected, because we known that S has size $(n-k) \times (m-k)$. In this case where S is very tall and very narrow, it is wasteful to compute S entirely. Many well-known techniques can be applied to obtain its rank by looking only at a fraction of its entries (see [24]). For instance, a naïve solution consists in choosing a small constant ϵ, building a dense matrix of size $(m-k+\epsilon) \times (m-k)$ with random (dense) linear combinations of the rows of S, and computing its rank using dense linear algebra. A linear combination of the rows of S can be formed by taking a random linear combinations of the rows of A, and then solving a triangular system, just like before.

4 Implementation and Results

All the experiments we carried on an Intel core i7-3770 with 8GB of RAM. Only one core was ever used. We used J.-G. Dumas's Sparse Integer Matrix Collection [16] as benchmark matrices. We restricted our attention to the 660 matrices with integer coefficients. Most of these matrices are small and their rank is easy to compute. Some others are pretty large. In all cases, their rank is known, as it could always be computed using the Wiedemann algorithm. We fixed $p = 42013$ in all tests.

Our implementation is quite straightforward. Matrices are stored in Compressed Sparse Row format. Coefficients are stored in `int` variables, and are reduced modulo p after each multiplication.

Sparse Elimination: Right-looking vs GPLU We first compare the efficiencies of the right-looking gaussian elimination algorithm implemented in LinBox and our implementation of the GPLU algorithm. LinBox uses its Markowitz-like pivot selection, while SpaSM uses its default setting : transposing the matrix if it has more columns than rows, and using the Faugère-Lachartre heuristic to select pivots before actually starting the factorization.

We quickly observed that no algorithm is always consistently faster than the other ; one may terminate instantly while the other may run for a long time and vice-versa. In order to perform a systematic comparison, we decided to set an arbitrary threshold of 60 seconds, and count the number of matrices that could be processed in this much time. LinBox could dispatch 579 matrices, while SpaSM processed 606. Amongst these, 568 matrices could be dealt with by both algorithms in less than 60s. This took 1100s to LinBox and 463s to SpaSM. LinBox was faster 112 times, and SpaSM 456 times. These matrices are "easy" for both algorithms, and thus we will not consider them anymore.

Table 2 shows the cases where one algorithm took less than 60s while the other took more. There are cases where each of the two algorithm is catastrophically slower than the other. In some cases, a bug made the LinBox test program crash with a segmentation fault. We conclude there is no clear winner (even if GPLU is usually a bit faster. The hybrid algorithm described in section 3 outperforms both.

Matrix	Right-looking	GPLU	Hybrid
`Franz/47104x30144bis`	39	488	1.7
`G5/IG5-14`	0.5	70	0.4
`G5/IG5-15`	1.6	288	1.1
`G5/IG5-18`	29	109	8
`GL7d/GL7d13`	11	806	0.3
`GL7d/GL7d24`	34	276	11.6
`Margulies/cat_ears_4_4`	3	184	0.1
`Margulies/flower_7_4`	7.5	667	2.5
`Margulies/flower_8_4`	37	9355	3.7
`Mgn/M0,6.data/M0,6-D6`	45	8755	0.1
`Homology/ch7-8.b4`	173	0.2	0.2
`Homology/ch7-8.b5`	611	45	10.7
`Homology/ch7-9.b4`	762	0.4	0.4
`Homology/ch7-9.b5`	3084	8.2	3.4
`Homology/ch8-8.b4`	1022	0.4	0.5
`Homology/ch8-8.b5`	5160	6	2.9
`Homology/n4c6.b7`	223	0.1	0.1
`Homology/n4c6.b8`	441	0.2	0.2
`Homology/n4c6.b9`	490	0.3	0.2
`Homology/n4c6.b10`	252	0.3	0.2
`Homology/mk12.b4`	72	9.2	1.5
`Homology/shar_te2.b2`	94	1	0.2
`Kocay/Trec14`	80	31	4
`Margulies/wheel_601`	7040	4	0.3
`Mgn/M0,6.data/M0,6-D11`	722	0.4	0.6
`Smooshed/olivermatrix.2`	75	0.6	0.1

Table 2: Comparison of sparse elimination techniques. Times are in seconds.

A reviewer asked a comparison with GBLA [3]. This is difficult, because GBLA is tailored for matrices arising in Gröbner basis computations, and exploit their specific shape. For instance, they have (hopefully small) *dense* areas, which GBLA rightly store in dense data structures. This phenomenon does not occur in our benchmark collection, which is ill-suited to GBLA. GBLA is nevertheless undoubtedly more efficient on Gröbner basis matrices.

Direct Methods vs Iterative Methods. We now turn our attention to the remaining matrices of the collection, the "hard" ones. Some of these are yet unamenable to any form of elimination, because they cause too much fill-in. However, some matrices can be processed much faster by our hybrid algorithm than by any other existing method.

For instance, `relat9` is the third largest matrix of the collection; computing its rank takes a little more than two days using the Wiedemann algorithm. Using the "Tall and Narrow Schur Complement" technique described above, the hybrid algorithm computes its rank in 34 minutes. Most of this time is spent in forming a size-8937 dense matrix by computing random dense linear combination of the 9 million remaining sparse rows. The rank of the dense matrix is quickly computed using the `Rank` function of FFLAS-FFPACK [27]. A straightforward parallelization using OpenMP brings this down to 10 minutes using the 4 cores of our workstation (the dense rank computation is not parallel). The same goes for the `rel9` matrix, which is similar.

The rank of the `M0,6-D9` matrix, which is the 9-th largest of the collection, could not be computed by the right-looking algorithm within the limit of 8GB of memory. It takes 42 hours to the Wiedemann algorithm to find its rank. The standard version of the hybrid algorithm finds it in less than 30 seconds.

The `shar_te.b3` matrix is an interesting case. It is a very sparse matrix of size 200200 with only 4 non-zero entries per row. Its rank is 168310. The right-looking algorithm fails, and the Wiedemann algorithm takes 3650s. Both GPLU and the hybrid algorithm terminate in more than 5 hours. However, performing *one* iteration of the hybrid algorithm computes a Schur complement of size 134645 with 7.4 non-zero entries per row on average. We see that the quantity $n|A|$, a rough indicator of the complexity of iterative methods, decreases a little. Indeed, computing the rank of the first Schur complement takes 2422s using the Wiedemann algorithm. This results in a 1.5× speed-up.

All-in-all, the hybrid algorithm is capable of quickly computing the rank of the 3rd, 6th, 9th, 11th and 13th largest matrices of the collection, whereas previous elimination techniques could not. Previously, the only possible option was the Wiedemann algorithm. The hybrid algorithm allows for large speedup of 100×, 1000× and 10000× in these cases.

Acknowledgement Claire Delaplace was supported by the french ANR under the BRUTUS project. We thank the anonymous reviewers for their comments.

Matrix	n	m	$\|A\|$	Right-looking	Wiedemann	Hybrid
kneser_10_4	349651	330751	992252	923	9449	0.1
mk13.b5	135135	270270	810810	M.T.	3304	41
M0,6-D6	49800	291960	1066320	42	979	0.1
M0,6-D7	294480	861930	4325040	2257	20397	0.8
M0,6-D8	862290	1395840	8789040	20274	133681	7.7
M0,6-D9	1395480	1274688	9568080	M.T.	154314	27.6
M0,6-D10	1270368	616320	5342400	22138	67336	42.7
M0,6-D11	587520	122880	1203840	722	4864	0.5
relat8	345688	12347	1334038		244	2
rel9	5921786	274667	23667185		127675	1204
relat9	9746232	274667	38955420		176694	2024

Table 3: Some harder matrices. Times are in seconds. M.T. stands for "Memory Thrashing"

References

1. Amestoy, P.R., Duff, I.S., Koster, J., L'Excellent, J.-Y..: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications 23(1), 15–41 (2001)
2. Bosma, W., Cannon, J., Playoust, C.: The Magma algebra system. I. The user language. J. Symbolic Comput. 24(3-4), 235–265 (1997), http://dx.doi.org/10.1006/jsco.1996.0125, computational algebra and number theory (London, 1993)
3. Boyer, B., Eder, C., Faugère, J., Lachartre, S., Martani, F.: GBLA - gröbner basis linear algebra package. CoRR abs/1602.06097 (2016), http://arxiv.org/abs/1602.06097
4. Cavallar, S.: Strategies in filtering in the number field sieve. In: Bosma, W. (ed.) Algorithmic Number Theory, 4th International Symposium, ANTS-IV, Leiden, The Netherlands, July 2-7, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1838, pp. 209–232. Springer (2000), http://dx.doi.org/10.1007/10722028_11
5. Chen, Y., Davis, T.A., Hager, W.W., Rajamanickam, S.: Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. ACM Trans. Math. Softw. 35(3), 22:1–22:14 (Oct 2008), http://doi.acm.org/10.1145/1391989.1391995
6. Cleveland Ashcraft, C., Grimes, R.G., Lewis, J.G., Peyton, B.W., Simon, H.D., Bjørstad, P.E.: Progress in sparse matrix methods for large linear systems on vector supercomputers. Int. J. High Perform. Comput. Appl. 1(4), 10–30 (Dec 1987), http://dx.doi.org/10.1177/109434208700100403
7. Coppersmith, D.: Solving homogeneous linear equations over $\mathbb{F}_2$ via block wiedemann algorithm. Mathematics of Computation 62(205), 333–350 (1994)
8. Davis, T.A.: Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. ACM Transactions On Mathematical Software 30(2), 196–199 (Jun 2004), http://dx.doi.org/10.1145/992200.992206
9. Davis, T.A.: Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2006)

10. Davis, T.A., Natarajan, E.P.: Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. ACM Trans. Math. Softw. 37(3) (2010), http://doi.acm.org/10.1145/1824801.1824814
11. Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S., Liu, J.W.H.: A supernodal approach to sparse partial pivoting. SIAM J. Matrix Analysis and Applications 20(3), 720–755 (1999)
12. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Numerical Mathematics and Scientific Computation, Oxford University Press, USA, first paperback edition edn. (1989)
13. Duff, I.S., Reid, J.K.: Some design features of a sparse matrix code. ACM Trans. Math. Softw. 5(1), 18–35 (Mar 1979), http://doi.acm.org/10.1145/355815.355817
14. Duff, I.S., Reid, J.K.: The multifrontal solution of indefinite sparse symmetric linear. ACM Trans. Math. Softw. 9(3), 302–325 (Sep 1983), http://doi.acm.org/10.1145/356044.356047
15. Dumas, J.G., Villard, G.: Computing the rank of sparse matrices over finite fields. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC'2002, Proceedings of the fifth International Workshop on Computer Algebra in Scientific Computing, Yalta, Ukraine. pp. 47–62. Technische Universität München, Germany (September 2002), http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/Publications/sparseeliminationCASC2002.pdf
16. Dumas, J.-G..: Sparse integer matrices collection, http://hpac.imag.fr
17. Dumas, J.-G.., Elbaz-Vincent, P., Giorgi, P., Urbanska, A.: Parallel computation of the rank of large sparse matrices from algebraic k-theory. In: Maza, M.M., Watt, S.M. (eds.) Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007, University of Western Ontario, London, Ontario, Canada. pp. 43–52. ACM (2007), http://doi.acm.org/10.1145/1278177.1278186
18. Faugère, J.-C.., Lachartre, S.: Parallel gaussian elimination for gröbner bases computations in finite fields. In: Maza, M.M., Roch, J.-L.. (eds.) PASCO. pp. 89–97. ACM (2010)
19. Gilbert, J.R., Peierls, T.: Sparse partial pivoting in time proportional to arithmetic operations. SIAM Journal on Scientific and Statistical Computing 9(5), 862–874 (1988), http://dx.doi.org/10.1137/0909058
20. Kleinjung, T., Aoki, K., Franke, J., Lenstra, A.K., Thomé, E., Bos, J.W., Gaudry, P., Kruppa, A., Montgomery, P.L., Osvik, D.A., te Riele, H.J.J., Timofeev, A., Zimmermann, P.: Factorization of a 768-bit RSA modulus. In: Rabin, T. (ed.) Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6223, pp. 333–350. Springer (2010), http://dx.doi.org/10.1007/978-3-642-14623-7_18
21. LaMacchia, B.A., Odlyzko, A.M.: Solving large sparse linear systems over finite fields. In: Menezes, A., Vanstone, S.A. (eds.) Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings. Lecture Notes in Computer Science, vol. 537, pp. 109–133. Springer (1990), http://dx.doi.org/10.1007/3-540-38424-3_8
22. Markowitz, H.M.: The elimination form of the inverse and its application to linear programming. Manage. Sci. 3(3), 255–269 (Apr 1957), http://dx.doi.org/10.1287/mnsc.3.3.255

23. May, J.P., Saunders, B.D., Wan, Z.: Efficient matrix rank computation with application to the study of strongly regular graphs. In: Wang, D. (ed.) Symbolic and Algebraic Computation, International Symposium, ISSAC 2007, Waterloo, Ontario, Canada, July 28 - August 1, 2007, Proceedings. pp. 277–284. ACM (2007), http://doi.acm.org/10.1145/1277548.1277586
24. Saunders, B.D., Youse, B.S.: Large matrix, small rank. In: Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation. pp. 317–324. ISSAC '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1576702.1576746
25. Saunders, D.: Matrices with two nonzero entries per row. In: Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation. pp. 323–330. ISSAC '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2755996.2756679
26. The CADO-NFS Development Team: CADO-NFS, an implementation of the number field sieve algorithm (2015), http://cado-nfs.gforge.inria.fr/, release 2.2.0
27. The FFLAS-FFPACK group: FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package, v2.0.0 edn. (2014), http://linalg.org/projects/fflas-ffpack
28. The Sage Developers: Sage Mathematics Software (Version 5.7) (2013), http://www.sagemath.org
29. Wiedemann, D.H.: Solving sparse linear equations over finite fields. IEEE Trans. Information Theory 32(1), 54–62 (1986), http://dx.doi.org/10.1109/TIT.1986.1057137