Computing the Rank of Large Sparse Matrices over Finite Fields
Jean-Guillaume Dumas, Gilles Villard

To cite this version:

HAL Id: hal-02068056
https://hal.archives-ouvertes.fr/hal-02068056
Submitted on 14 Mar 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Computing the Rank of Large Sparse Matrices over Finite Fields

Jean-Guillaume Dumas\(^1\) and Gilles Villard\(^2\)

\(^1\) Laboratoire de Modélisation et Calcul. B.P. 53, 38041 Grenoble, France.
\(^2\) Laboratoire de l’Informatique du Parallélisme, Allée d’Italie, F69364 Lyon, France.
Jean-Guillaume.Dumas@imag.fr Gilles.Villard@ens-lyon.fr

Abstract. We want to achieve efficient exact computations, such as the rank, of sparse matrices over finite fields. We therefore compare the practical behaviors, on a wide range of sparse matrices of the deterministic Gaussian elimination technique, using reordering heuristics, with the probabilistic, blackbox, Wiedemann algorithm. Indeed, we prove here that the latter is the fastest iterative variant of the Krylov methods to compute the minimal polynomial or the rank of a sparse matrix.

1 Introduction

Many applications of computer algebra require the most effective algorithms for the computation of normal form of matrices. An essential linear algebra part of these is often the computation of the rank of large sparse matrices; for example, many methods for factoring integers require the solution of large sparse linear systems \cite{24}, the integer Smith form arising in the computation of homology groups \cite{27} can be determined using rank computations \cite{13}, image compression, robotics use computation of Gröbner basis where huge sparse linear systems must be solved \cite{16}.

We first recall some Gaussian elimination strategies (§2) and propose a new reordering heuristic. Some experimental results with these are presented in (§2.2). Then we produce a fast preconditioning for the Wiedemann iterative method in order to compute the rank of a matrix via minimum polynomial computation (§3). We will then compare both methods using a discrete logarithm based arithmetic (§4). In §2.2 and §4 we will also present some results concerning matrices arising in the computation of Gröbner basis for image compression or robotics (GB project \cite{16}) and in the determination of homology groups of simplicial complexes \cite{5}. All the matrices used are available on the Sparse Integer Matrices Collection (www-lmc.imag.fr/lmc-mosaic/Jean-Guillaume.Dumas/simc.html).

2 Gaussian Elimination

2.1 Reordering Techniques

There exists many numerical methods to reduce fill-in during Gaussian elimination. Unfortunately, Yan- nakakis showed that finding the minimal fill-in (or equivalently the best pivots) is an NP-complete task \cite{35}. Therefore many heuristics has been developed. In numerical algorithms, there are two mainly used methods. One considers the matrix as an adjacency of a graph and uses minimal degree ordering to choose the pivot node \cite{4}. Another one uses a cost function: at step \(k\), \(r_i(k)\) is the number of non-zero elements within row \(i\) and \(c_j(k)\) is the number of non-zeroes within column \(j\). Markowitz’ version chooses to eliminate with the row and columns that minimize the function \((r_i(k) − 1)(c_j(k) − 1)\). Zlatev \cite{37}, Duff and Reid \cite{8} or Amestoy and Davis \cite{2} have conducted many experiments on this subject. There is a third class of heuristics, namely the “nested dissection”\cite{1, 18, 23}. This method reduces fill-in from 20 to 30% in some cases when compared to minimum degree ordering. But it needs a costly symbolic pre-factorization which often induces some overhead. Moreover it is not easily usable with non-symmetric, non-invertible matrices. Anyway, all these methods are adapted to numerical methods (the matrices are then very often supposed invertible !) and therefore do not take into account the values. For instance an elimination operation \((a_{ij} = a_{ij} + d_i * a_{kj})\) cannot produce new zeroes when done numerically. That is why pre-factorization is efficient. On the other hand, when dealing with small finite fields, new zeroes appears very often. We then propose a new heuristic to take those new zeroes into account.

In the following, we will denote by \(\Omega\) the number of non zero elements of a \(m \times n\) matrix, and by \(\bar{w}\) (= \(\frac{\Omega}{m}\)) the average number of non zero elements per row.
In symbolic computations, LaMacchia and Odlyzko [24] considered fill-in modulo 2 and proposed a heuristic adapted to integer factorization, further developed by S. Cavallar [6]. We here propose a heuristic, using some of the ideas of LaMacchia and Odlyzko together with a linear Markowitz’ cost function.

(1) We first try to reduce the size of the matrix by suppressing rows or columns containing at most one non-zero element.

(2) Then, because apparition of new zeroes cannot be predicted, we minimize fill-in only locally.

Moreover, in order to reduce the overhead of a total cost function, we instead choose the pivot with the following heuristic: at step k, we choose a row i with a minimal number of non-zero elements. Then we choose in this row the column with minimal number of elements. Therefore the pivot choice only requires \( m + r_i(k) \leq m + n \) tests per step, whereas classical Markowitz’ function requires \( \Omega = m \varpi \) tests and multiplications in the first steps and soon become quadratic as the matrix fills in. The idea is close to the local minima proposed by Rothberg[30] for numerical Cholesky.

To be able to choose a row at a low cost, we use a compressed row format and maintain a vector of number of non-zero elements in the columns.

The second phase is slow since each step is now quadratic, thus yielding a cubic overall cost:

\[
\frac{1}{3} (r + 1 - s) (3n(m - r - s) + 3m(n - r - s) + 2r^2 + O(r)) \leq \frac{2}{3} rmn.
\]
2.2 Experimental Results

We used several class of matrices from different kind of applications. In order to make some comparisons we present results using gains: if \( tps_{no} \) (respectively \( op_{no} \)) is the execution time (resp. the number of arithmetic operations) without reordering and \( tps_{re} \) (resp. \( op_{re} \)) are the time and operations with reordering, we normalize the gains as follows: 
\[
\text{gain(time)} = 100 \times \frac{tps_{no} - tps_{re}}{tps_{no}}, \quad \text{gain(op)} = 100 \times \frac{op_{no} - op_{re}}{op_{no}}.
\]
A gain close to 0 is still an improvement when a negative value is a loss. The maximal gain is 100.

Experiments have been conducted over \( \mathbb{Z}_p \), the prime field with \( p \) elements or over \( \mathbb{GF}(p^k) \), one of its extension fields. We used a tabulated finite field implementation [11] so that the arithmetic cost of the basic operations remains nearly the same for prime fields and their extensions as long as the cardinality remains within machine word size. However, since zero appears more frequently e.g. over \( \mathbb{Z}_3 \) or \( \mathbb{GF}(4) \) than over \( \mathbb{Z}_{65521} \) or \( \mathbb{GF}(65536) \), we have conducted experiments with those two extreme kinds of word size finite fields.

Random matrices. We first ran our algorithms on random matrices on identical machines (133 MHz, ultra-SPARC II), in order to give an idea of the overall performances. Figure 2 presents the result of our heuristic for several matrices of dimension 1000 to 5000 with 1 to 70 non zero elements.

1. We see on figure 2 that for extremely sparse matrices (1 or 2 elements per row), our heuristic is nearly optimal. Clearly this is due to LaMacchia and Odlyzko’s trick.

2. Then, between 2 and 4 elements per row, we win a little less: for very sparse and quite small matrices, fill-in is not important even without reordering.

3. Now, as soon as the average number of non-zeroes per row is bigger than 3, we drastically reduce fill-in.

4. Moreover, we remark a slight difference between \( \mathbb{Z}_3 \) and \( \mathbb{Z}_{65521} \): this because our heuristic is even more efficient on small moduli.

Gröbner basis matrices. We now see that our heuristic can also reduce fill-in on denser matrices (used in Gröbner basis computations [16]). Unfortunately, for one of those (8555.mat9), the gain is not sufficient to compensate the strategy overhead. In the other cases we have speed-ups bigger than 20%.
Fig. 2. Reordering gains on random matrices

Graph 1: Reordering Gain (%) vs. Non-zero elements per row

Graph 2: Reordering Gain (%) vs. Non-zero elements per row

Legend:
- Operations mod 3
- Time mod 3
- Operations mod 65521
- Time mod 65521
Table 1. Reordering gains for Gröbner matrices

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Elements Non zero</th>
<th>Dimensions</th>
<th>Rank</th>
<th>Gain modulo 65521 (%)</th>
<th>Operations</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>robot24c1_mat5</td>
<td>12.39%</td>
<td>404 x 302</td>
<td>262</td>
<td>51.28</td>
<td>30.59</td>
<td></td>
</tr>
<tr>
<td>rkat7_mat5</td>
<td>7.44%</td>
<td>694 x 738</td>
<td>611</td>
<td>69.50</td>
<td>41.78</td>
<td></td>
</tr>
<tr>
<td>f855_mat9</td>
<td>2.77%</td>
<td>2456 x 2511</td>
<td>2331</td>
<td>19.88</td>
<td>-10.22</td>
<td></td>
</tr>
<tr>
<td>cyclic8_mat11</td>
<td>9.37%</td>
<td>4562 x 5761</td>
<td>3903</td>
<td>37.57</td>
<td>20.92</td>
<td></td>
</tr>
</tbody>
</table>

Homology matrices. We then used matrices of homology groups of simplicial complexes[5, 3, 29, 14]. These matrices are almost diagonal, therefore, reordering induces a slight overhead in many cases. The figure 3 presents performances for 37 of those matrices. For instance there are 5 matrices having more than 90% operation gain. These very sparse matrices have a quite long linear phase and then in a few steps fill-in is spectacular. On figure 1 we test slight changes of pivot and we can see that the required memory can vary from 130 to 590 Mb with the same matrix. Reordering those very peculiar matrices is then very unstable and no good general heuristic has been found yet.

BIBD Matrices. This test matrices come from combinatorics. They are very rectangular matrices of Balanced Incomplete Block Design [33]: once again when there is a very small number of rows, the steps remains linear and reordering is not useful as one can see on figure 4. Those tests are useful anyway to evaluate the overhead of our heuristic. Indeed, for the BIBD, there is no operation gain. The time overhead is therefore only due to our reordering. Because this overhead is close to 70%, we estimate that our reordering induces a time factor of 1.7. We see in next paragraph that this remains acceptable when compared to Markowitz' heuristic for instance.

Comparison with Markowitz' function. To compare our heuristic with other strategies, we have implemented Markowitz' total pivot, still using LaMacchia and Odlyzko’s trick. In practice, we see on table 2 that Markowitz is always slower. This is true even though for several matrices Markowitz method enables a higher reduction of fill-in.
Fig. 4. Reordering gain on BIBD matrices

Table 2. Our linear cost function versus Markowitz’

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Gain modulo 65521 (%)</th>
<th>Matrix</th>
<th>Gain modulo 65521 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>robot24_m5</td>
<td>3.99</td>
<td>n2c6.b4</td>
<td>27.93</td>
</tr>
<tr>
<td>rkat7_m5</td>
<td>-19.06</td>
<td>n2c6.b5</td>
<td>74.21</td>
</tr>
<tr>
<td>f855_m9</td>
<td>-56.24</td>
<td>n2c6.b6</td>
<td>45.30</td>
</tr>
<tr>
<td>cyclic8_m11</td>
<td>-76.72</td>
<td>n2c6.b7</td>
<td>23.63</td>
</tr>
<tr>
<td>bibd.14.7</td>
<td>-3.80</td>
<td>n3c6.b5</td>
<td>41.66</td>
</tr>
<tr>
<td>bibd.15.7</td>
<td>6.01</td>
<td>n3c6.b7</td>
<td>31.73</td>
</tr>
<tr>
<td>bibd.16.8</td>
<td>-1.55</td>
<td>n3c6.b8</td>
<td>84.02</td>
</tr>
<tr>
<td>bibd.17.4</td>
<td>10.30</td>
<td>n3c6.b9</td>
<td>1.41</td>
</tr>
<tr>
<td>bibd.17.8</td>
<td>-5.69</td>
<td>n4c5.b3</td>
<td>44.14</td>
</tr>
<tr>
<td>bibd.18.9</td>
<td>-0.30</td>
<td>n4c5.b4</td>
<td>64.18</td>
</tr>
<tr>
<td>bibd.19.9</td>
<td>-12.38</td>
<td>n4c5.b5</td>
<td>61.70</td>
</tr>
<tr>
<td>bibd.20.10</td>
<td>5.49</td>
<td>n4c5.b6</td>
<td>48.34</td>
</tr>
<tr>
<td>bibd.22.8</td>
<td>5.26</td>
<td>n4c5.b7</td>
<td>41.30</td>
</tr>
<tr>
<td>bibd.81.3</td>
<td>31.60</td>
<td>n4c5.b8</td>
<td>21.35</td>
</tr>
<tr>
<td>ch6-6.b3</td>
<td>48.46</td>
<td>n4c5.b9</td>
<td>21.45</td>
</tr>
<tr>
<td>ch6-6.b4</td>
<td>47.72</td>
<td>n4c6.b3</td>
<td>19.78</td>
</tr>
<tr>
<td>mk9.b12</td>
<td>45.92</td>
<td>n4c6.b4</td>
<td>67.58</td>
</tr>
<tr>
<td>mk9.b3</td>
<td>8.95</td>
<td>n2c6.b8</td>
<td>16.82</td>
</tr>
<tr>
<td>mk10.b2</td>
<td>46.58</td>
<td>n4c6.b13</td>
<td>19.96</td>
</tr>
<tr>
<td>mk10.b3</td>
<td>69.75</td>
<td>n4c6.b14</td>
<td>3.85</td>
</tr>
<tr>
<td>mk11.b2</td>
<td>34.46</td>
<td>61.67</td>
<td></td>
</tr>
<tr>
<td>mk11.b3</td>
<td>66.19</td>
<td>62.81</td>
<td></td>
</tr>
</tbody>
</table>
Comparison with SuperLU. Finally, on table 3 we compare our method to a generic version\(^1\) of the "SuperLU" numerical code\(^2\).

### Table 3. Our linear cost function versus SuperLU, on a PIII, 1Gb, 1GHz, (timings in seconds)

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Ω, n × m, r</th>
<th>Linear</th>
<th>SuperLU</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclic8.m11</td>
<td>2462970, 4562x5761, 3903</td>
<td>257.33</td>
<td>448.38</td>
</tr>
<tr>
<td>bbd32.8</td>
<td>8935560, 231x319770, 231</td>
<td>100.24</td>
<td>594.29</td>
</tr>
<tr>
<td>minc.b12</td>
<td>1721226, 25605x69235, 20165</td>
<td>188.34</td>
<td>1312.27</td>
</tr>
<tr>
<td>ch7-7.b5</td>
<td>211680, 35280x52920, 29448</td>
<td>2179.62</td>
<td>Memory Thrashing</td>
</tr>
<tr>
<td>ch7-8.b5</td>
<td>5375.76</td>
<td>Memory Thrashing</td>
<td></td>
</tr>
<tr>
<td>TF13</td>
<td>11185, 1121x1302, 1121</td>
<td>3.18</td>
<td>3.54</td>
</tr>
<tr>
<td>TF14</td>
<td>29862, 2644x3160, 2644</td>
<td>50.58</td>
<td>50.34</td>
</tr>
<tr>
<td>TF15</td>
<td>80057, 6334x7742, 6334</td>
<td>734.39</td>
<td>776.68</td>
</tr>
<tr>
<td>TF16</td>
<td>216173, 15437x19321, 15437</td>
<td>18559.40</td>
<td>15625.79</td>
</tr>
</tbody>
</table>

We can see that our method is really competitive and supports a higher memory demand.

3  Blackboxes and Diagonal Scaling

We present here an iterative method for computing the rank, i.e. we do not modify the matrix, only some matrix-vector products are needed. When a matrix is only used this way we call it a "Blackbox". Therefore an algorithm viewing matrices as blackboxes will not suffer from any fill-in.

We will first recall that we have access to the rank of a square matrix via its minimum polynomial when this matrix is well preconditioned \[22\]. We actually use a faster and more generic randomization:

**Theorem 2.** \[15, Theorem 6.2\] Let \(S\) be a finite subset of a field \(F\) that does not include 0. Let \(A \in F^{m \times n}\) having rank \(r\). Let \(D_1 \in S^{m \times m}\) and \(D_2 \in S^{n \times n}\) be two random diagonal matrices then \(\deg(\text{minpoly}(D_1 \times A^t \times D_2 \times A \times D_1)) = r\), with probability \(1 - \frac{11 \times n^2 - n}{2^{|S|}}\).

Informally this follows since first for a square matrix \(A\), \(DA\) will generically have its characteristic polynomial be a power of \(X\) times its minimum polynomial \[7\]. Then \(A^t.A\) is symmetric and will therefore have no zero blocks on its Jordan form. No power of \(X\) can then occur in its minimum polynomial. And then \(A^t.D.A\) will generically suppress self-orthogonality within rows and keep the rank unchanged. To conclude, the probability follows from Schwartz-Zippel lemma \[36\].

3.1  Wiedemann’s Algorithm

Then we use Wiedemann’s algorithm \[34\] to compute the minimum polynomial and therefore the rank when used with a good preconditioning. Wiedemann’s algorithm is a shift-register synthesis \[26\] using some projection of the powers of the matrices as a sequence:

**Algorithm 2** Diagonally scaled Wiedemann

- **Input**: – a sparse matrix \(A\) in \(F^{m \times n}\) of rank \(r\).
- **Output**: – the rank of \(A\) over \(F\) with probability \(1 - \frac{11 \times n^2 - n}{2^{|S|}}\).

**Blackbox realization**

1. Select random \(D_1 \in F^{m \times m}\) and \(D_2 \in F^{n \times n}\) as stated above.

   [Blackbox Initialization with explicit multiplication by the diagonals]

2. a : Compute \(B = D_1 \ast A \ast D_2\), the matrix whose entries are \(D_1_{ij}A_{ij}D_2_{ij}\).

   [Blackbox Initialization otherwise]

\(^1\) www-sop.inria.fr/galaad/logiciels/synaps
\(^2\) www.nersc.gov/~xiaoye/SuperLU
Form the true Blackbox composition \( C = D_2 A' D_1 A D_2 \)

Wiedemann Sequence Initialization
3 : Set \( u \in \mathbb{F}^m \) a random vector.
4 : set \( S_0 = u^t u \)

Berlekamp/Massey Initialization
5 : set \( b = 1; e = 1; L = 0; \varphi = 1 \in \mathbb{F}[X]; \psi = 1 \in \mathbb{F}[X]; \)
6 : For \( k = 0 \) to \( 2 \min(m, n) \) Do

Berlekamp/Massey shift register synthesis
7 : \( \delta = S_k + \sum_{i=1}^{L} \varphi_i S_{k-i} \)
8 : If \( \delta = 0 \) Then
9 : ++
10 : Else, if \( 2L > k \) Then
11 : \( \varphi = \varphi - \frac{\delta}{\varphi} X^e \psi \)
12 : ++
13 : Else
14 : \( \varphi = \varphi - \frac{\delta}{\varphi} X^e \psi // \psi = \varphi \)
15 : \( L = k + 1 - L; b = \delta; e = 1 \)

Early termination
16 : If \( e > \text{TerminationThreshold} \) Then Break.

[Next coefficient, non-symmetric case]
17 a : If \( k \) even Then
18 a : \( v = Au \)
19 a : \( S_{k+1} = v^t u \)
20 a : Else \{ or \}
21 a : \( u = A^t v \)
22 a : \( S_{k+1} = u^t u \)

[Next coefficient, symmetric case]
17 b : If \( k \) even Then
18 b : \( v = Au \)
19 b : \( S_{k+1} = u^t u \)
20 b : Else \{ or \}
21 b : \( u = v \)
22 b : \( S_{k+1} = u^t u \)

Las Vegas Check
23 : Set \( z \in \mathbb{F}^n \) a random vector.

[Apply \( \varphi \) to \( A' A \) and \( z \), non-symmetric case]
24 a : \( w = \varphi(A'A).v \)

[Apply \( \varphi \) to \( A \) and \( z \), symmetric case]
24 b : \( w = \varphi(A).v \)
25 : If \( w == 0 \) Then
26 : Return degree(\( \varphi \)) - valuation(\( \varphi \)).
27 : Else
28 : ”Failure”

Remark 1. As first observed by Lobo [20], when the matrix has rank \( r < \min(n, m) \) this algorithm will produce the minimal polynomial after only \( 2r \) steps. Therefore one can heuristically stop it when \( \phi \) remains the same after a certain number of steps. This is done via the EarlyTerminationThreshold within algorithm 2. The number of matrix-vector computations can therefore be drastically reduced in some cases. This argument is heuristic in general but provable for the Lanczos algorithm on preconditioned matrices over suitably large fields [15, also Eberly (private Communication 2000)]

Remark 2. We present here both the generic algorithm when explicit multiplication by a diagonal is possible and one with true Blackbox. The cost of a matrix-vector product is reduced in the former case but the size of the set of random choices is divided by 2 since it is equivalent to selecting a random diagonal matrix in the set of squares of \( F \).
Theorem 3. Let $A$ be a matrix in $\mathbb{F}^{m \times n}$ of rank $r$. Algorithm 2 is correct and requires $2r$ matrix-vector products with $4r^2 + 2r(n + m)$ supplementary field operations. It also requires $5n$ memory space in addition to the matrix storage.

Proof. For correctness of the minimum polynomial see [26, 34]. For correctness of the rank use theorem 2. Now for arithmetic complexity: the loop ends when the degree of the polynomial reaches $r$ (i.e. when $k = 2r$) according to remark 1. Loop $k$ has a cost of $L = \frac{2r}{r}$ “gaxpy” operations for discrepancy computation, another $\frac{r}{2}$ for the polynomial update, a matrix-vector product, and a $n$ or $m$-dotproduct. The additional complexity is therefore $\sum_{k=3}^{2r-1} \frac{2r}{r} + 2 \frac{n \cdot m}{r} = 4r^2 - 2r + 2nr + 2m$. We conclude with the memory complexity: one vector is needed to store $S$, two vectors are needed to compute the matrix-vector products and only two polynomials are required to perform the shift-register step. Indeed, the fact that the shifting degree, $x$, is greater than one allows us to perform the polynomial updates in place.

In the case where $A$ is sparse with $\Omega$ non-zero elements (and $\varpi = \frac{2}{\pi}$, the average number of non-zero elements per row, is a constant), the total cost of this algorithm is $4r(\Omega + r + n) = O(n^2 \omega)$. So asymptotically this algorithm is better than Gaussian elimination which has a worst case arithmetic complexity in $O(n^3 \omega)$. However the constant factor of $4\varpi + 8$ is not negligible when one considers that Gaussian elimination has a low cost in its first steps. This cost is then growing up to $\frac{2}{7}mn^2$ according to the fill-in.

Remark 3. For the symmetric case there are only $n$-dotproducts. Therefore if $n > m$ it is better to form this other Blackbox: $B = D_1 A D_2 A^T D_1$. Indeed, in view of the following lemma 1, the minimal polynomial will have the same degree (or only a difference of 1) and therefore the number of iterations will not change.

Lemma 1. Let $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times m}$ be two rectangular matrices. Then the minimal polynomials of $AB$ and $BA$ are equal or differ by a factor of $X$.

Proof. Let $m_{AB}(X)$ and $m_{BA}(X)$ be respectively the minimal polynomials of $AB$ and $BA$. The Cayley-Hamilton theorem [17, Theorem IV, §4.2] states that $m_{AB}(AB) = 0$. Then, by multiplying on both sides by $B$ and $A$ we have $Bm_{AB}(AB)A = 0$ which means that $(X m_{AB})(BA) = 0$. Since $m_{BA}$ is the minimal polynomial of $BA$ it follows that $m_{BA}$ divides $X m_{AB}$. We can similarly prove that $m_{AB} X m_{BA}$. Then either $m_{AB} = X m_{BA}$ or $m_{AB} = m_{BA}$ or $X m_{AB} = m_{BA}$.

3.2 Other Krylov Methods

The minimum polynomial $f^{A,u}$ of a vector $u$ is computed as a linear dependency between the iterates $u, Au, A^2u, \ldots$. In the Krylov/Wiedemann approach, the dependency is found by applying the Berlekamp-Massey algorithm to the sequence $v^T u, v^T Au, v^T A^2u, \ldots$ for a random vector $v$. We let $d$ be the degree of the minimum linear generating polynomial $f^{A,u}_x(x) = x^d - f_{d-1}x^{d-1} - \ldots - f_1 x - f_0$ of the latter sequence, meaning that $v^T A^i u = f_{d-i} (v^T A^{i+1} u) + \ldots + f_{1+i} (v^T Au) + f_0 (v^T u), i \geq 0$. Then $f^{A,u}_x(x)$ always divides $f^{A,u}_x$ and, furthermore, with high probability, $f^{A,u}_x(x) = f^{A,u}[34]$. The Berlekamp-Massey algorithm will compute $f^{A,u}_u(x)$ after processing the first $2d$ elements of the sequence. The generation of the sequence is distinguished from the computation of the linear recurrence. In the Lanczos approach, these tasks are interleaved into a unique iteration. While the vectors in the Krylov subspace are generated and orthogonalized, the coefficients of the recurrence are computed on the fly as dot products. For a unifying study of both approaches over finite fields we refer to [25]. With high probability, the minimum polynomial $f^{A,u}_x$ of a random vector $u$ is the minimum polynomial $f_A^x$ of $A$ [34, 22]. The basic implementation computes the minimum polynomial of $A$ using Wiedemann’s algorithm and two random vectors $u$ and $v$ to generate the sequence $\{v^T A^i u\}_{0 \leq i \leq 2n-1}$. The algorithm is randomized of the Monte Carlo type. As first observed by Lobo, the cost can be reduced by early termination. As soon as the linear generator computed by Berlekamp-Massey process remains the same for a few steps (as indicated by discrepancies of zero) then the minimum polynomial is known. The argument is heuristic in the general case but probabilistic when applied over large fields to preconditioned matrices with Lanczos’ algorithm [15]. A Monte Carlo check of the early termination is implemented by incorporating the application of the computational polynomial to the vector $u$. From [25] and [9, Chap. 6], we give the dominant terms of the arithmetic costs in Table 4. Terms between brackets give the number of memory locations required for field elements. Early termination and randomized Monte Carlo algorithms correspond to bi-orthogonal Lanczos algorithms with or without look-ahead. In both approaches, the number of vector-matrix products may be cut in half if the matrix is symmetric. Since the update of the linear generator is computed
by dot products instead of elementary polynomial operations, a Lanczos strategy may always have a slightly higher cost for computing the minimum polynomial.

Table 4. Costs of Wiedemann and Lanczos algorithms for $f^A$ of degree $d$. $A$ or $A^T$ can be applied to a vector using at most $\Omega$ operations.

<table>
<thead>
<tr>
<th></th>
<th>Early termination $f^A$</th>
<th>Monte Carlo $f^A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wiedemann [5n]</td>
<td>$2d\Omega + 4d(n + d)$</td>
<td>$2n\Omega + 4n^2 + 2d(n + d)$</td>
</tr>
<tr>
<td>Lanczos [3n]</td>
<td>$2d\Omega + 8dn$</td>
<td>$2n\Omega + 4n^2 + 4dn$</td>
</tr>
</tbody>
</table>

4 David versus Goliath

In this section we report on some experiments comparing the behavior of Wiedemann’s algorithm and Gaussian’s algorithm.

4.1 Arithmetic

We first compare the respective implementations of our algorithms on a Sun UltraII 250 MHz. The idea is to measure the number of millions of field operations (Mops) that each algorithm is able to perform in a second.

Table 5. Million of field (65521 elements) OPerations per Second, dense dot product compared with Gauss and Wiedemann.

<table>
<thead>
<tr>
<th></th>
<th>machine ints</th>
<th>Tabulated Field</th>
<th>Gauss</th>
<th>Wiedemann</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>23.08</td>
<td>13.04</td>
<td>0</td>
<td>7.25</td>
</tr>
<tr>
<td>Average</td>
<td>23.08</td>
<td>13.04</td>
<td>1.82</td>
<td>8.67</td>
</tr>
<tr>
<td>Maximum</td>
<td>23.08</td>
<td>13.04</td>
<td>4.57</td>
<td>12.12</td>
</tr>
</tbody>
</table>

We therefore produce the performance obtained for all the matrices of section 2.2 and compare them in table 5 together with a dense dot product with machine int or using a tabulated implementation of a finite field (see [11]). Then, comparing the number of operations reached by our two algorithms, we recall that sparse elimination requires many data structures manipulations and that reordering requires a few more structures manipulations. Next thing to remark is that we achieve better performances when measured in term of “Mops” with Blackbox methods than with elimination; we were able to run at 12.12 Mops in the best case. Indeed in Wiedemann algorithm there are sparse matrix-vector operations which need structural manipulations but also a fairly important proportion of dense dot products and polynomial multiplications.

Remark 4. In the following experiments we will present results for matrices of size greater than half the size of the field. The conditions of theorem 3 do not hold anymore for the probabilities of success of Wiedemann algorithm. However it seems clear that the probability estimates of 2 are too pessimistic for most of the matrices. Indeed, we check our results with the deterministic Gaussian algorithm whenever it is possible. It appears that with a field of size 65521, Wiedemann’s algorithm simply never failed on our matrices when the rows or columns containing only one entry were removed. We therefore present timings associated with correct Wiedemann answers when the Gaussian elimination result is present. For some other matrices, Gaussian elimination ran out of memory and we don’t know the correct answer. Some experiments with arbitrary precision integers and larger fields will have to be conducted to check correctness. Anyway we present the timings associated to these matrices with the restriction that the rank might not be correct.
4.2 Matrices

**Random matrices** We first produce in figure 5 the results obtained for random matrices. We see that for very sparse matrices Gaussian elimination is far better since it has nearly nothing to do! Anyway, and even for small matrices, as soon as the fill-in is no longer negligible Wiedemann algorithm takes the advantage.

![Fig. 5. Compared timings of elimination and iterative techniques on random matrices.](image)

This happens when $\omega$ is greater than 6. Moreover, this advantage remains until matrices are nearly dense, say as long as $\omega < \frac{\min\{m,n\}}{3}$ (i.e. up to 30% sparse, when comparing arithmetic complexities in theorems 1 and 3).

**Gröbner and Homology matrices** Also, for much more triangular matrices, Gaussian elimination has not much to compute and is therefore better than Wiedemann's algorithm. However, we note that for very large matrices Wiedemann's algorithm is able to execute whereas Gaussian elimination fails because of
memory thrashing. For these cases, where memory is the limiting factor, even the slightest fill-in can kill elimination.

Table 6. Compared timings of elimination and iterative techniques on homology and Gröbner matrices.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>$\Omega, n \times m, r$</th>
<th>Gauß</th>
<th>Wiedemann</th>
</tr>
</thead>
<tbody>
<tr>
<td>robot24c1_mat5</td>
<td>15118, 404x302, 262</td>
<td>0.52</td>
<td>1.84</td>
</tr>
<tr>
<td>rkat7_mat5</td>
<td>38114, 694x738, 611</td>
<td>1.85</td>
<td>10.51</td>
</tr>
<tr>
<td>f855_mat9</td>
<td>171214, 2456x2511 , 2331</td>
<td>10.54</td>
<td>202.17</td>
</tr>
<tr>
<td>c8_mat11</td>
<td>2462970,4562x5761, 3903</td>
<td>671.33</td>
<td>4972.99</td>
</tr>
<tr>
<td>robot24c1_mat5</td>
<td>3780,945x1260,875</td>
<td>0.26</td>
<td>2.11</td>
</tr>
<tr>
<td>rkat7_mat5</td>
<td>35280,5040x35280,5040</td>
<td>4.67</td>
<td>119.53</td>
</tr>
<tr>
<td>f855_mat9</td>
<td>75600,15120x12600, 8989</td>
<td>49.32</td>
<td>412.42</td>
</tr>
<tr>
<td>c8_mat11</td>
<td>211680,35280x52920,2944</td>
<td>2179.62</td>
<td>4141.32</td>
</tr>
<tr>
<td>ch7-7.b4</td>
<td>1881600,376320x117600</td>
<td>33 hours</td>
<td></td>
</tr>
</tbody>
</table>

Table 7. Compared timings of elimination and Wiedemann on incidence matrices (in cpu seconds on an Intel PIII 993 MHz and 1Gb).

<table>
<thead>
<tr>
<th>Matrix</th>
<th>$\Omega, n \times m, r$</th>
<th>Gauß</th>
<th>Wiedemann</th>
</tr>
</thead>
<tbody>
<tr>
<td>n=10</td>
<td>622,99x107,99</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>n=11</td>
<td>1607,216x236,216</td>
<td>0.02</td>
<td>0.09</td>
</tr>
<tr>
<td>n=12</td>
<td>4231,488x552,488</td>
<td>0.22</td>
<td>0.62</td>
</tr>
<tr>
<td>n=13</td>
<td>11185,1121x1302,1121</td>
<td>4.37</td>
<td>4.45</td>
</tr>
<tr>
<td>n=14</td>
<td>29862,2644x3160,2644</td>
<td>61.57</td>
<td>27.21</td>
</tr>
<tr>
<td>n=15</td>
<td>80057,6334x7742,6334</td>
<td>1002.14</td>
<td>165.67</td>
</tr>
<tr>
<td>n=16</td>
<td>216173,15437x19321,15437</td>
<td>18559.4</td>
<td>1248.46</td>
</tr>
<tr>
<td>n=17</td>
<td>586218,38132x48630,38132</td>
<td>Memory Thrashing</td>
<td>7094.97</td>
</tr>
<tr>
<td>n=18</td>
<td>1597545,95368x123867,95368</td>
<td>Memory Thrashing</td>
<td>58893.6</td>
</tr>
<tr>
<td>n=19</td>
<td>4370721,241029x317955,241029</td>
<td>Memory Thrashing</td>
<td>359069</td>
</tr>
</tbody>
</table>

5 Block Algorithms and Parallelism

We have compared the sequential methods to compute the rank of large sparse matrices. Next question is whether this comparison will remain when parallelizing the algorithms. In this section we sketch the parallel strategies and infer their respective behaviors from a communication and time cost analysis.

5.1 Turbo

A way to compute the rank in parallel is to use an exact $LU$ factorization (the effective computation of only the $U$ matrix is of course sufficient). A first idea is to use a parallel direct method on matrices stored by rows (respectively columns). There, at stage $k$ of the classical Gaussian elimination algorithm, eliminations are executed in parallel on the $n - k - 1$ remaining rows; thus giving only a small grain. The next idea is therefore to mimic numerical methods and use sub-matrices. Now, the problem is that usually, for symbolic computation, these blocks are singular. To solve this problem one has mainly two alternatives. One is to perform a dynamic cutting of the matrix and to adjust it so that the blocks are reduced and become

Incidences matrices These matrices are incidence matrices of unlabelled trees on $n$ nodes versus unlabelled forests on $n$ nodes with $n-2$ edges [28]. The major point here is that those matrices have an average number of non zero elements growing from 6 to 18. Our analysis on random matrices shows that Wiedemann is better on those. This is indeed the case as shown in table 7.
invertible. Such a method is shown by Ibarra et al. They build an algorithm computing the rank of an \( m \times n \) matrix with arithmetic complexity \( O(m^{\omega-1}n) \), where the complexity of matrix multiplication is \( O(m^\omega) \). Unfortunately, their method is not so efficient in parallel: it induces synchronizations and significant communications at each stage in order to compute the block redistribution. Another method, called Turbo, using *singular static blocks* in order to avoid these synchronizations and redistributions has been proposed in [12]. This algorithm also has an optimal sequential arithmetic complexity and is able to *avoid 30%* of the communications for dense matrices.

Still, the amount of communications remains fairly large even when compared to the arithmetic cost on dense matrices. Those methods are therefore not suited at all to sparse matrices. Moreover, they requires even more memory than the sequential versions. Therefore, there remains to design a direct parallel method suited to sparse matrices and to study effective methods to reorder sparse matrices in parallel.

### 5.2 Parallelization of Wiedemann’s Algorithm

An easy parallelization of Wiedemann’s algorithm is to cut \( A \) in cyclic bi-dimensional blocks and to parallelize the matrix-vector and dot products [10, §4.1] and [9, §6.8.1]. Table 8 shows the acceleration one can obtain this way. As for the direct parallel methods, these speed-ups are on symmetric multi-processor machines. Unfortunately, this does not work well on distributed architectures. Indeed on \( p \) processors, one has to communicate a total of \( 2n\sqrt{p} \) values for each bi-dimensional matrix-vector product. This volume of communication thus induces an insufficient overlapping. On distributed architectures the solution is instead to use the iterative algorithms with several starting vectors as shown in next section.

### 5.3 Block Krylov Methods

In this section we focus on the block versions of the Krylov methods. These are variants of Lanczos or Wiedemann’s approaches but using several initial vectors at the same time. Therefore computations are grouped and the number of iterations is reduced. There are two main advantages. The probabilities of success are improved in small finite fields [32]. Also, there is more parallelism (with exactly the same number of matrix-vector products). However, the overall number of field operations is unfortunately higher than for the classical versions. The most commonly used block variants are block Lanczos and Coppersmith’s version of block Wiedemann. Just like in table 4, we give in table 9 the dominant terms of the arithmetic costs from [21] and [9, Chap. 6.7].

<table>
<thead>
<tr>
<th>Matrix</th>
<th>( \omega, n \times m )</th>
<th>Sequential</th>
<th>4 processors</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>bibd22_8</td>
<td>38760, 231x319770</td>
<td>995.41</td>
<td>343.80</td>
<td>2.90</td>
</tr>
<tr>
<td>ch7-6.b4</td>
<td>5, 15120x12600</td>
<td>412.42</td>
<td>240.24</td>
<td>1.72</td>
</tr>
<tr>
<td>ch7-7.b5</td>
<td>6, 35280x52920</td>
<td>4141.32</td>
<td>1865.12</td>
<td>2.22</td>
</tr>
<tr>
<td>mk12.b4</td>
<td>5, 62370x51975</td>
<td>7539.21</td>
<td>2959.42</td>
<td>2.55</td>
</tr>
<tr>
<td>ch8-8.b4</td>
<td>5, 376320x117600</td>
<td>105 hours</td>
<td>34 hours</td>
<td>3.10</td>
</tr>
<tr>
<td>ch9-9.b5</td>
<td>6, 423360x317520</td>
<td>105 hours</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>ch8-8.b5</td>
<td>6, 564480x376320</td>
<td>105 hours</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Coppersmith’s version of block Wiedemann’s algorithm uses blocks of vectors to perform several matrix-vector products at the same time. Still the computation of the block generating polynomial (a matrix polynomial) remains sequential. Together with its higher cost, this prevents us to use this parallelization on our matrices. Indeed, the block algorithm is more interesting when the matrix-vector product
cost is dominating. With $\Omega = O(n)$ as is the case for our matrices, the matrix-vector products are faster than the expensive matrix polynomial generation. More experiments have to be conducted with slightly denser matrices, say with $\Omega$ in the $O(n \log(n))$ range.

6 Conclusion

We have experimented with the currently known best methods for computing the rank of large sparse matrices. We first validated an efficient heuristic for fill-in reduction in Gaussian elimination. We have then seen that for certain highly structured matrices Gaussian elimination is still faster since it has relatively little to do! On the other hand we can say that Wiedemann’s iterative algorithm is very practical. We show also that this is the fastest iterative method to compute the rank. It has good behavior in general, even for small matrices, and is the only solution for extremely large matrices. Moreover, an effective parallelization on Symmetric Multi Processor machines is possible even though not fully expendable to distributed architectures.

Therefore, there remains to extend the comparisons and developments of both the direct and iterative parallel block versions.

Acknowledgements. We are grateful to Volkmar Welker, Jean-Charles Faugère, Mark Giesbrecht, Nicolas M. Thiéry for their providing us with test matrices.

References