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 [23], the integer Smith form arising in the computation of homology groups [26] can be determined using rank computations [12], image compression, robotics use computation of Gröbner basis where huge sparse linear systems must be solved [15].

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 [15]) and in the determination of homology groups of simplicial complexes [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, Yannakakis showed that finding the minimal fill-in (or equivalently the best pivots) is an NP-complete task [34]. 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 [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 [36], Duff and Reid [8] or Amestoy and Davis [2] have conducted many experiments on this subject. There is a third class of heuristics, namely the "nested dissection"[1, 17, 22]. 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 ($a_{ij} = a_{ij} + \delta_i * a_{kj}$) 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 Ω the number of non zero elements of a $m \times n$ matrix, and by ϖ ($= \frac{\Omega}{m}$) the average number of non zero elements per row.

In symbolic computations, LaMacchia and Odlyzko [23] 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[29] 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 column density (in the following algorithm, column degrees are stored in D_j and computed lines 6, 15 and 20).

Algorithm 1 Symbolic-Reordering

```
Input  : – a matrix $A \in \mathbb{F}^{m\times n}$.
Output : – the rank of $A$ over $\mathbb{F}$.
1  : $r = 0$
2  : **While** there exists a row with only one non-zero element, $a_{kj} \neq 0$ **Do**
3  :     $++r$
4  :     Remove row $k$ and column $j$ from $A$.
5  : **For** $j = 1$ **to** $n$ **Do**
6  :     Compute $D_j$, number of non-zero elements in column $j$.
7  : **While** there exists a column with only one non-zero element, $a_{kj} \neq 0$ **Do**
8  :     $++r$
9  :     Remove row $k$ and column $j$ from $A$.
10 : $\Lambda = \{k, \exists j, a_{kj} \neq 0\}$
11 : **While** $\Lambda \neq \emptyset$ **Do**
12 :     $++r$
13 :     Choose and remove $k \in \Lambda$ such that $|A[k]|$ is minimal.
14 :     **For all** $j$ **such that** $a_{kj} \neq 0$ **Do**
15 :         Decrement $D_j$
16 :     Choose $j$ such that $a_{kj} \neq 0$ and $D_j$ is minimal.
17 :     **For all** $i \in \Lambda$ **such that** $a_{ij} \neq 0$ **Do**
18 :         $A[i] = A[i] - \frac{a_{ij}}{a_{kj}} A[k]$.
19 :         **For** $h == 1$ **to** $n$ **Do**
20 :             Decrement or increment $D_h$ whenever $a_{ih}$ has changed.
21 :     Remove from $\Lambda$ the indices corresponding to new empty rows.
22 : Return r.
```

Theorem 1. *Let A be a sparse matrix in $\mathbb{F}^{m\times n}$ of rank r with at most ϖ non-zero elements per row. Algorithm 1 requires $2\sum_{k=1}^{r}(m-k)\min\{\varpi 2^k, n-k\}$ field operations and the same order of memory space in the worst case.*

Proof. We consider that each step doubles the number of elements per row. This gives at most $\min\{\varpi 2^k, n-k\}$ elements per row at step k.

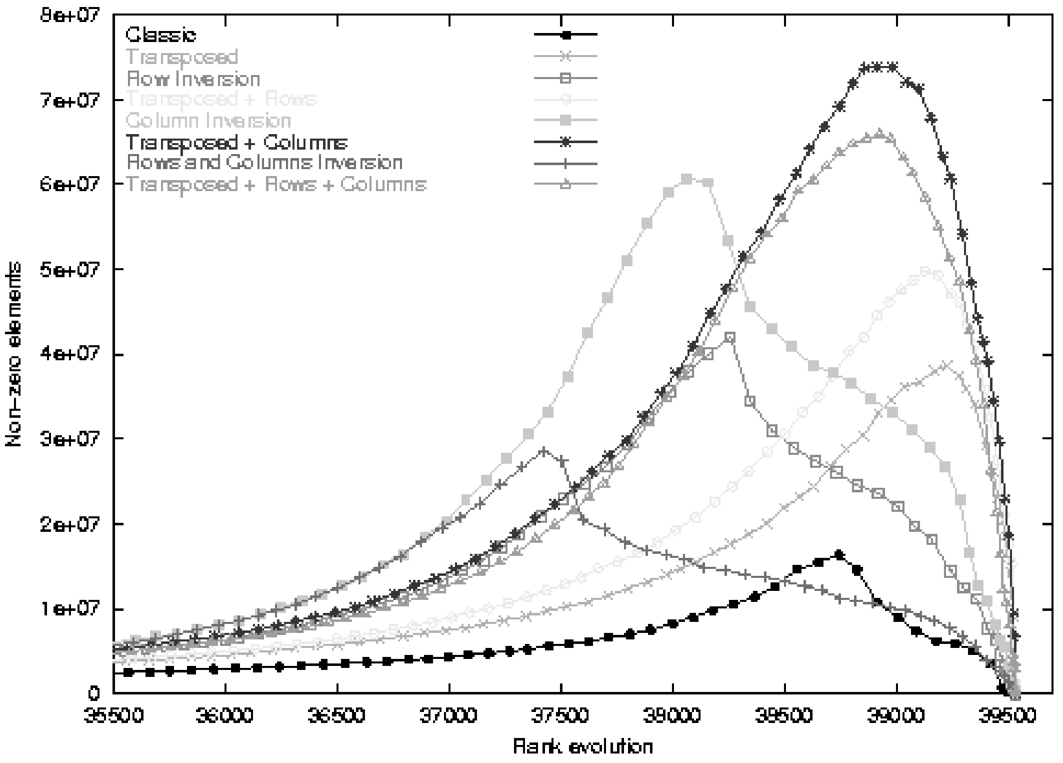
We can quantify fill-in using this theorem. Indeed, as we can see e.g. on figure 1, there are two phases: the first one lasts as long as the remaining $(m-k)\times(n-k)$ matrix is sparse. The number of sparse steps is s, usually in the order of $log_2(\frac{n}{\varpi})$ since $\varpi 2^{log_2(\frac{n}{\varpi})} = n > n-k$. This phase is fast since its steps are linear but does not last long. Its overall cost is only quadratic:

$$2\varpi 2^s(m-s+1) < 2n(m-s+1).$$

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

$$\frac{1}{3}(r+1-s)\big(3n(m-r-s)+3m(n-r-s)+2r^2+\mathcal{O}(r)\big) \leq \frac{2}{3}rmn.$$

Fig. 1. Fill-in curve for different reordering strategies on matrix mk12.b4, 62370x51975 with rank 39535



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: $gain(\text{time}) = 100 * \frac{tps_{no}-tps_{re}}{tps_{no}}$, $gain(\text{op}) = 100 * \frac{op_{no}-op_{re}}{op_{no}}$. 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 $\texttt{GF}(p^k)$, one of its extension fields. We used a tabulated finite field implementation [10] 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 $\texttt{GF}(4)$ than over $\mathbb{Z}_{65521}$ or $\texttt{GF}(65536)$, 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.

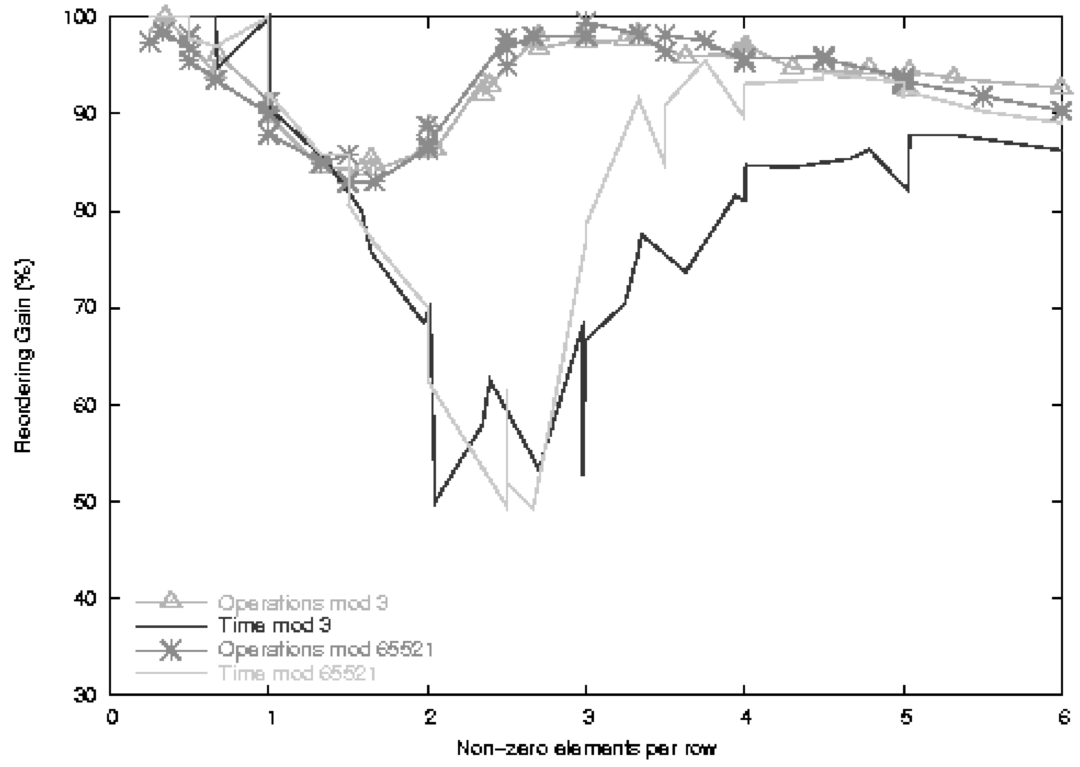
(2) Then, between 2 and 4 elements per row, we win a little less: for very sparse and quite small matrices, fill-in is not important even without reordering.

(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 [15]). Unfortunately, for one of those (f855_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



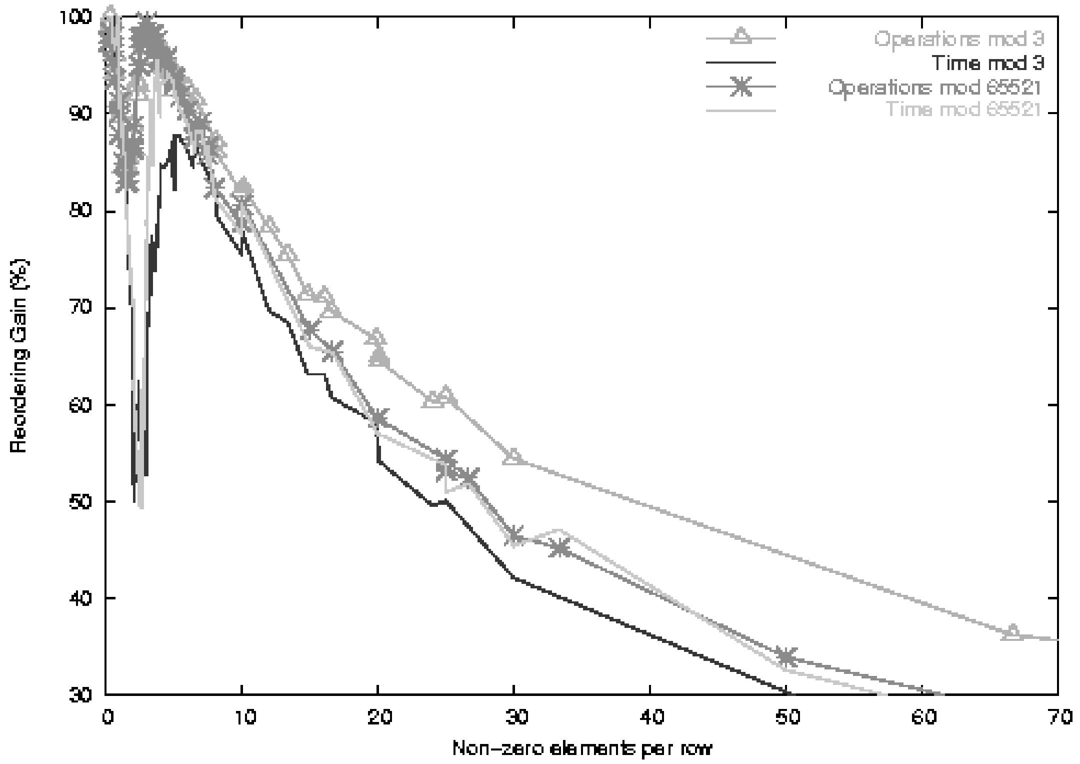
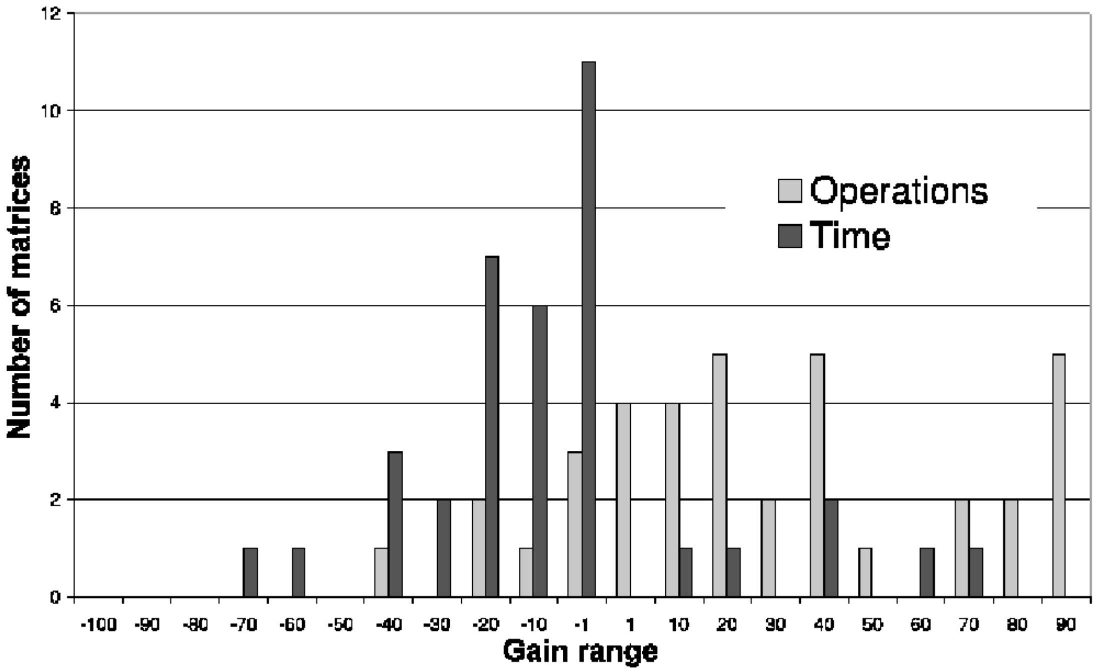


Table 1. Reordering gains for Gröbner matrices

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

Homology matrices. We then used matrices of homology groups of simplicial complexes[5, 3, 28, 13]. These matrices are almost diagonal. therefore, reordering induces a slight overhead in many cases. The figure 3

Fig. 3. Reordering gains on some Homology matrices



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* [32]: 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

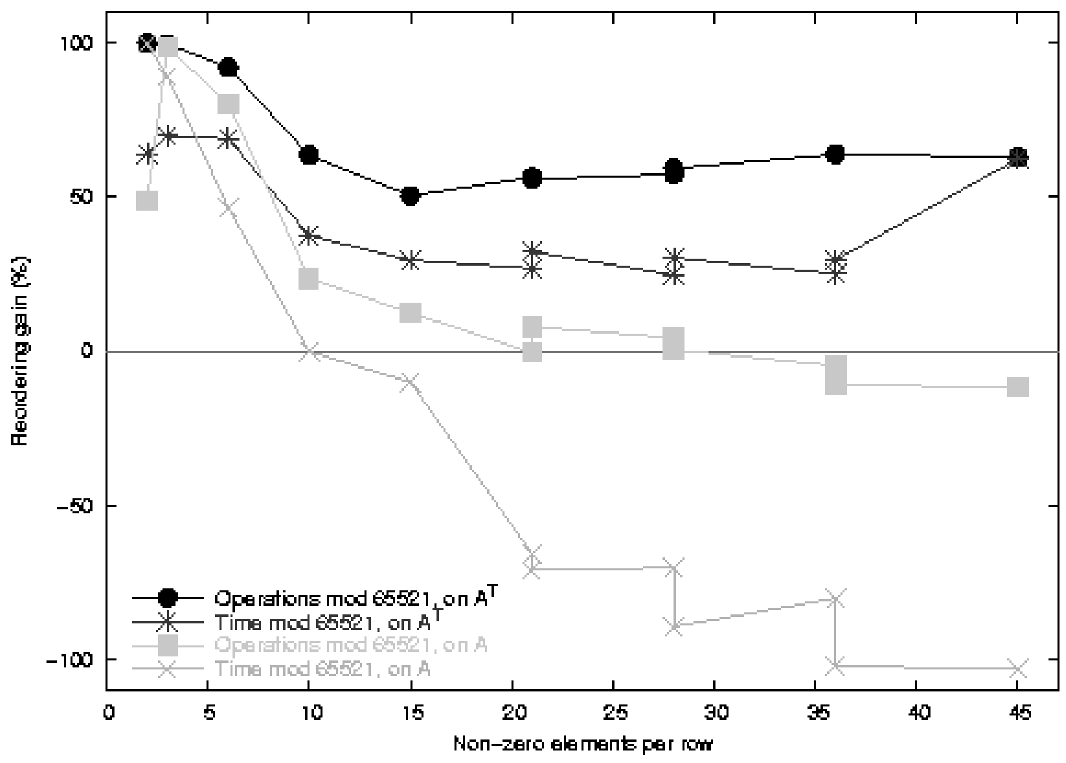


Table 2. Our linear cost function versus Markowitz'

Matrix	Gain modulo 65521 (%)		Matrix	Gain modulo 65521 (%)	
	Operations	Time		Operations	Time
robot24_m5	3.99	26.09	n2c6.b4	27.93	68.66
rkat7_m5	-19.06	44.12	n2c6.b5	74.21	85.77
f855_m9	-56.24	71.96	n2c6.b6	45.30	77.52
cyclic8_m11	-76.72	42.61	n2c6.b7	23.63	80.69
bibd.14.7	-3.80	55.77	n3c6.b5	41.66	78.78
bibd.15.7	6.01	51.61	n3c6.b7	31.73	78.07
bibd.16.8	-1.55	47.01	n3c6.b8	84.02	88.65
bibd.17.4	10.30	72.73	n3c6.b9	1.41	78.76
bibd.17.8	-5.69	45.86	n4c5.b3	44.14	62.50
bibd.18.9	-0.30	53.81	n4c5.b4	64.18	78.35
bibd.19.9	-12.38	53.95	n4c5.b5	61.70	83.58
bibd.20.10	5.49	60.20	n4c5.b6	48.34	82.52
bibd.22.8	5.26	60.82	n4c5.b7	41.30	86.77
bibd.81.3	31.60	91.31	n4c5.b8	21.35	82.24
ch6-6.b3	48.46	76.32	n4c5.b9	21.45	71.43
ch6-6.b4	47.72	78.16	n4c6.b3	19.78	65.46
mk9.b2	45.92	57.14	n4c6.b4	67.58	63.72
mk9.b3	8.95	42.86	n2c6.b8	16.82	62.07
mk10.b2	46.58	44.44	n4c6.b13	19.96	85.41
mk10.b3	69.75	77.94	n4c6.b14	3.85	70.59
mk11.b2	34.46	61.67			
mk11.b3	66.19	62.81			

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)

Matrix	Ω, $n \times m$, r	Linear	SuperLU
cyclic8_m11	2462970, 4562x5761, 3903	257.33	448.38
bibd_22_8	8953560, 231x319770, 231	100.24	594.29
n4c6.b12	1721226, 25605x69235, 20165	188.34	1312.27
ch7-7.b5	211680, 35280x52920, 29448	2179.62	Memory Thrashing
ch7-8.b5	846720, 141120x141120, 92959	5375.76	Memory Thrashing
TF13	11185, 1121x1302, 1121	3.18	3.54
TF14	29862, 2644x3160, 2644	50.58	50.34
TF15	80057, 6334x7742, 6334	734.39	776.68
TF16	216173, 15437x19321, 15437	18559.40	15625.79

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 [21]. We actually use a faster and more generic randomization:

Theorem 2. *[14, 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 $D_1 \in S^{n\times n}$ and $D_2 \in S^{m\times m}$ be two random diagonal matrices then degree(minpoly($D_1 \times A^t \times D_2 \times A \times D_1$)) $= r$, with probability $1 - \frac{11.n^2-n}{2|S|}$.*

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 [35].

3.1 Wiedemann's Algorithm

Then we use Wiedemann's algorithm [33] to compute the minimum polynomial and therefore the rank when used with a good preconditioning. Wiedemann's algorithm is a shift-register synthesis [25] using some projection of the powers of the matrix as a sequence:

Algorithm 2 *Diagonally scaled Wiedemann*

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

    Blackbox realization
1 : Select random $D_1 \in \mathbb{F}^{m\times m}$ and $D_2 \in \mathbb{F}^{n\times n}$ as stated above.
    [Blackbox Initialization with explicit multiplication by the diagonals]
2 a : Compute $B = D_1 * A * D_2$, the matrix whose entries are $D_{1_{ii}} A_{ij} D_{2_{jj}}$.
    [Blackbox Initialization otherwise]
```

[1] www-sop.inria.fr/galaad/logiciels/synaps
[2] www.nersc.gov/~xiaoye/SuperLU

```
2 b : Form the true Blackbox composition $C = D_2 A^t D_1 A D_2$
      Wiedemann Sequence Initialization
3   : Set $u \in \mathbb{F}^n$ 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   :     $\delta = S_k + \sum_{i=1}^{L} \varphi_i S_{k-i}$                                $2\frac{k}{2}$
8   :     If $(\delta == 0)$ Then
9   :         $++e$
10  :     Else, if $2L > k$ Then
11  :         $\varphi = \varphi - \frac{\delta}{b} X^e \psi$                                 $2\frac{k}{2}$
12  :         $++e$
13  :     Else
14  :         $\varphi = \varphi - \frac{\delta}{b} X^e \psi$ // $\psi = \varphi$                       $2\frac{k}{2}$
15  :         $L = k + 1 - L; b = \delta; e = 1$
          Early termination
16  :     If $e > TerminationThreshold$ Then Break.
          [Next coefficient, non-symmetric case]
17 a :    If $k$ even Then
18 a :        $v = Au$                                                $\Omega$
19 a :        $S_{k+1} = v^T v$                                       2m
20 a :    Else                                                    {or}
21 a :        $u = A^T v$                                             $\Omega$
22 a :        $S_{k+1} = u^T u$                                       2n
          [Next coefficient, symmetric case]
17 b :    If $k$ even Then
18 b :        $v = Au$                                                $\Omega$
19 b :        $S_{k+1} = u^T v$                                       2n
20 b :    Else                                                    {or}
21 b :        $u = v$
22 b :        $S_{k+1} = u^T u$                                       2n
      Las Vegas Check
23  : Set $z \in \mathbb{F}^n$ a random vector.
      [Apply $\varphi$ to $A^t A$ and $z$, non-symmetric case]
24 a : $w = \varphi(A^t A).v$
      [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 [19], 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 ϕ 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 [14, 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 a 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 [25, 33]. 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 $L = \frac{k}{2}$ "gaxpy" operations for discrepancy computation, another $\frac{k}{2}$ for the polynomial update, a matrix-vector products, and a n or m-dotproduct. The additional complexity is therefore $\sum_{k=0}^{2r-1} 4\frac{k}{2} + 2\frac{n+m}{2} = 4r^2 - 2r + 2nr + 2mr$. 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 Ω non-zero elements (and $\varpi = \frac{\Omega}{n}$, the average number of non-zero elements per row, is a constant), the total cost of this algorithm is $4r(\Omega + r + n) = O(rn)$. So asymptotically this algorithm is better than Gaussian elimination which has a worst case arithmetic complexity in $O(rn^2)$. 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 $\frac{2}{3}rn^2$ 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 : $B = D_1AD_2A^tD_1$. 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 $A \in R^{m\times n}$ and $B \in R^{n\times m}$ 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 [16, 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 $(Xm_{AB})(BA) = 0$. Since m_{BA} is the minimal polynomial of BA it follows that m_{BA} divides Xm_{AB}. We can similarly prove that $m_{AB}|Xm_{BA}$. Then either $m_{AB} = Xm_{BA}$ or $m_{AB} = m_{BA}$ or $Xm_{AB} = m_{BA}$.

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 $v^{\mathrm{T}}u, v^{\mathrm{T}}Au, v^{\mathrm{T}}A^2u, \ldots$ for a random vector v. We let d be the degree of the minimum linear generating polynomial $f_v^{A,u}(x) = x^d - f_{d-1}x^{d-1} - \ldots - f_1x - f_0$ of the latter sequence, meaning that $v^{\mathrm{T}}A^{d+i}u = f_{d-1}(v^{\mathrm{T}}A^{d+i-1}u) + \ldots + f_{i+1}(v^{\mathrm{T}}Au) + f_i(v^{\mathrm{T}}u), i \geq 0$. Then $f_v^{A,u}(x)$ always divides $f^{A,u}$ and, furthermore, with high probability, $f_v^{A,u}(x) = f^{A,u}$ [33]. The Berlekamp-Massey algorithm will compute $f_v^{A,u}(x)$ 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 intermixed 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 [24]. With high probability, the minimum polynomial $f^{A,u}$ of a random vector u is the minimum polynomial f^A of A [33, 21]. The basic implementation computes the minimum polynomial of A using Wiedemann's algorithm and two random vectors u and v to generate the sequence $\{v^{\mathrm{T}}A^iu\}_{0\leq i\leq 2n-1}$. 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 [14]. A Monte Carlo check of the early termination is implemented by incorporating the application of the computed polynomial to the vector u. From [24] 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 matrix-vector 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 Ω operations.

	Early termination f^A	Monte Carlo f^A
Wiedemann [$5n$]	$2d\Omega + 4d(n+d)$	$2n\Omega + 4n^2 + 2d(n+d)$
Lanczos [$3n$]	$2d\Omega + 8dn$	$2n\Omega + 4n^2 + 4dn$

4 David versus Goliath

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

	machine ints	Tabulated Field	Gauß	Wiedemann
Minimum	23.08	13.04	0	7.25
Average	23.08	13.04	1.82	8.67
Maximum	23.08	13.04	4.57	12.12

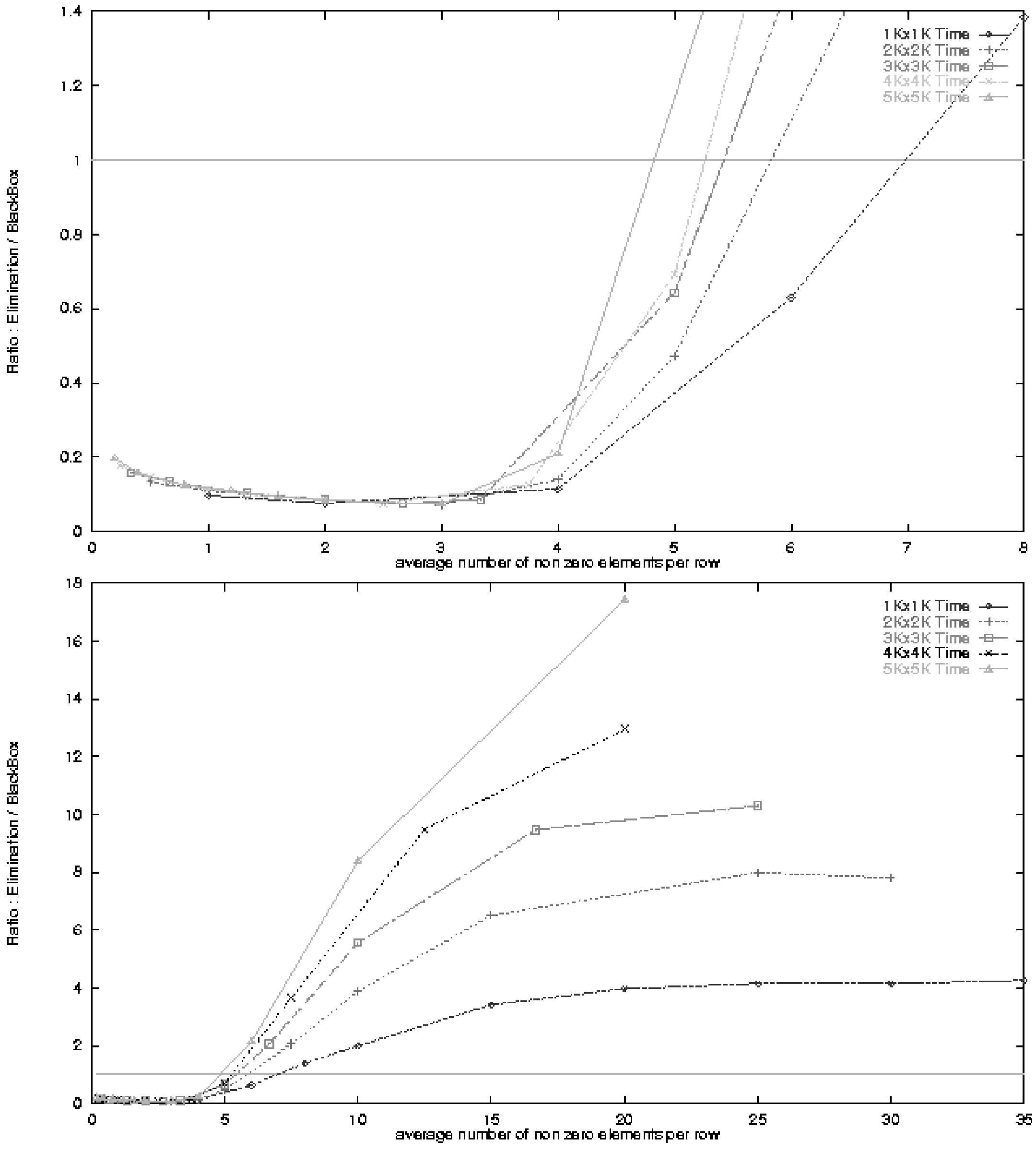
We therefore produce the performance obtained for all the matrices of section 2.2 and compare them in table 5 together with a dense dot product with machine `int` or using a tabulated implementation of a finite field (see [10]). Then, comparing the number of operations reached by our two algorithms, 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 Gauß 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.



This happens when ϖ is greater than 6. Moreover, this advantage remains until matrices are nearly dense, say as long as $\varpi < \frac{min\{m;n\}}{3}$ (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.

Matrix	Ω, $n \times m$, r	Gauß	Wiedemann
robot24c1_mat5	15118,404x302,262	0.52	1.84
rkat7_mat5	38114, 694x738, 611	1.85	10.51
f855_mat9	171214, 2456x2511 , 2331	10.54	202.17
c8_mat11	2462970,4562x5761, 3903	671.33	4972.99
mk9.b3	3780,945x1260,875	0.26	2.11
ch7-7.b6	35280,5040x35280,5040	4.67	119.53
ch7-6.b4	75600, 15120x12600, 8989	49.32	412.42
ch7-7.b5	211680,35280x52920,29448	2179.62	4141.32
ch8-8.b4	1881600, 376320x117600	Memory Thrashing	33 hours

Incidence matrices These matrices are incidence matrices of unlabelled trees on n nodes versus unlabelled forests on n nodes with $n-2$ edges [27]. 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.

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

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

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 invertible. Such a method is shown by Ibarra et al [18]. They build an algorithm computing the rank of an $m \times n$ matrix with arithmetic complexity $\mathcal{O}(m^{\omega-1}n)$, 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 [11]. 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. Table 8 shows the acceleration one can obtain this

Table 8. Parallel Wiedemann on a Sun Ultra-II 4×250 MHz, modulo 32749

Matrix	ϖ, $n \times m$	Sequential	4 processors	Speed-up
bibd_22_8	38760, 231x319770	995.41	343.80	2.90
ch7-6.b4	5, 15120x12600	412.42	240.24	1.72
ch7-7.b5	6, 35280x52920	4141.32	1865.12	2.22
mk12.b4	5, 62370x51975	7539.21	2959.42	2.55
ch8-8.b4	5, 376320x117600	33 hours	10 hours	3.37
ch7-9.b5	6, 423360x317520	105 hours	34 hours	3.10
ch8-8.b5	6, 564480x376320		55 hours	-

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 [31]. 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 [20] and [9, Chap. 6.7].

Table 9. Costs of Coppersmith's block Wiedemann and block Lanczos algorithms for f^A of degree d, using p initial vectors at the same time. A or A^{T} can be applied to a vector using at most Ω operations.

	Early termination f^A
Coppersmith $[4pn + 2p^2 + 2dp]$	$2d\Omega + 4pd(n + d)$
Block Lanczos $[3pn + 3p^2]$	$2d\Omega + (8p + 2)dn + O(dp^2)$

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 $\Omega = \mathcal{O}(n)$ 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 Ω in the $\mathcal{O}(n\ log(n))$ 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

1. P. Amestoy, F. Pellegrini, and J. Roman. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. In *Proceeding of IRREGULAR'99*, volume 1225, pages 986–995, Puerto Rico, Apr. 1999. Lecture Notes in Computer Science.
2. P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, Oct. 1996.
3. E. Babson, A. Björner, S. Linusson, J. Shareshian, and V. Welker. Complexes of not i-connected graphs. *Topology*, 38(2):271–299, 1999.
4. P. Berman and G. Schnitger. On the performance of the minimum degree ordering for Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 11(1):83–88, Jan. 1990.
5. A. Björner, L. Lovász, S. T. Vrećica, and R. T. Živaljević. Chessboard complexes and matching complexes. *Journal of the London Mathematical Society*, 49(1):25–39, 1994.
6. S. Cavallar. Strategies in filtering in the number field sieve. Technical report, Centrum voor Wiskunde en Informatica, May 2000. http://www.cwi.nl/ftp/CWIreports/MAS/MAS-R0012.ps.Z.
7. L. Chen, W. Eberly, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. Efficient matrix preconditioners for black box linear algebra. *Linear Algebra and its Applications*, 343-344:119–146, 2002.
8. I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, UK, 1986.
9. J.-G. Dumas. *Algorithmes parallèles efficaces pour le calcul formel : algèbre linéaire creuse et extensions algébriques*. PhD thesis, Institut National Polytechnique de Grenoble, France, Dec. 2000. `ftp://-ftp.imag.fr/pub/Mediatheque.IMAG/theses/2000/Dumas.Jean-Guillaume`.
10. J.-G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In T. Mora, editor, *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, Lille, France*. ACM Press, New York, July 2002.
11. J.-G. Dumas and J.-L. Roch. A fast parallel block algorithm for exact triangularization of rectangular matrices. In *SPAA'01. Proceedings of the Thirteenth ACM Symposium on Parallel Algorithms and Architectures, Kreta, Greece.*, pages 324–325, July 2001.
12. J.-G. Dumas, B. D. Saunders, and G. Villard. Integer Smith form via the Valence: experience with large sparse matrices from Homology. In Traverso [30], pages 95–105.
13. J.-G. Dumas, B. D. Saunders, and G. Villard. On efficient sparse integer matrix Smith normal form computations. *Journal of Symbolic Computations*, 32(1/2):71–99, July–Aug. 2001.
14. W. Eberly and E. Kaltofen. On randomized Lanczos algorithms. In W. W. Küchlin, editor, *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii*, pages 176–183. ACM Press, New York, July 1997.
15. J.-C. Faugère. Parallelization of Gröbner basis. In H. Hong, editor, *First International Symposium on Parallel Symbolic Computation, PASCO '94, Hagenberg/Linz, Austria*, volume 5 of *Lecture notes series in computing*, pages 124–132, Sept. 1994.
16. F. R. Gantmacher. *The Theory of Matrices*. Chelsea, New York, 1959.
17. B. Hendrickson and E. Rothberg. Improving the run time and quality of nested dissection ordering. *SIAM Journal on Scientific Computing*, 20(2):468–489, Mar. 1999.
18. O. H. Ibarra, S. Moran, and R. Hui. A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, 3(1):45–56, Mar. 1982.
19. E. Kaltofen, W.-S. Lee, and A. A. Lobo. Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. In Traverso [30], pages 192–201.
20. E. Kaltofen and A. Lobo. Distributed matrix-free solution of large sparse linear systems over finite fields. In A. Tentner, editor, *Proceedings of High Performance Computing 1996, San Diego, California*. Society for Computer Simulation, Simulation Councils, Inc., Apr. 1996.
21. E. Kaltofen and B. D. Saunders. On Wiedemann's method of solving sparse linear systems. In *Applied Algebra, Algebraic Algorithms and Error–Correcting Codes (AAECC '91)*, volume 539 of *Lecture Notes in Computer Science*, pages 29–38, Oct. 1991.

22. G. Karypis and V. Kumar. A fast and high quality mutilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, Jan. 1999.
23. B. A. LaMacchia and A. M. Odlyzko. Solving large sparse linear systems over finite fields. *Lecture Notes in Computer Science*, 537:109–133, 1991. http://www.research.att.com/~amo/doc/arch/sparse.linear.eqs.ps.
24. R. Lambert. *Computational aspects of discrete logarithms*. PhD thesis, University of Waterloo, Ontario, Canada, 1996. http://www.cacr.math.uwaterloo.ca/techreports/2000/lambert-thesis.ps.
25. J. L. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, IT-15:122–127, 1969.
26. J. R. Munkres. *Elements of algebraic topology*, chapter The computability of homology groups, pages 53–61. Advanced Book Program. The Benjamin/Cummings Publishing Company, Inc., 1994.
27. M. Pouzet and N. M. Thiéry. Invariants algébriques de graphes et reconstruction. *Comptes Rendus de l'Académie des Sciences*, 333(9):821–826, 2001.
28. V. Reiner and J. Roberts. Minimal resolutions and the homology of matching and chessboard complexes. *Journal of Algebraic Combinatorics*, 11(2):135–154, Mar. 2000.
29. E. Rothberg and S. C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications*, 19(3):682–695, July 1998.
30. C. Traverso, editor. *ISSAC'2000. Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation, Saint Andrews, Scotland*. ACM Press, New York, Aug. 2000.
31. G. Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Technical Report 975–IM, LMC/IMAG, Apr. 1997.
32. W. D. Wallis, A. P. Street, and J. S. Wallis. *Combinatorics: Room Squares, Sum-Free Sets, Hadamard Matrices*, volume 292 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1972.
33. D. H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, Jan. 1986.
34. M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79, Mar. 1981.
35. R. Zippel. *Effective Polynomial Computation*, chapter Zero Equivalence Testing, pages 189–206. Kluwer Academic Publishers, 1993.
36. Z. Zlatev. *Computational Methods for General Sparse Matrices*, chapter Pivotal Strategies for Gaussian Elimination, pages 67–86. Kluwer Academic Publishers, Norwell, MA, 1992.