Nearly Sparse Linear Algebra and application to Discrete Logarithms Computations

Antoine Joux, Cécile Pierrot

Chapter 1

Nearly Sparse Linear Algebra and application to Discrete Logarithms Computations

Antoine Joux

Chaire de Cryptologie de la Fondation de l'UPMC
Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606
Paris, France
Antoine.Joux@m4x.org

Cécile Pierrot

Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606
Paris, France
Cecile.Pierrot@lip6.fr

Abstract. In this article[1], we propose a method to perform linear algebra on a matrix with nearly sparse properties. More precisely, although we require the main part of the matrix to be sparse, we allow some dense columns with possibly large coefficients. This is achieved by modifying the Block Wiedemann algorithm. Under some precisely stated conditions on the choices of initial vectors in the algorithm, we show that our variation not only produces a random solution of a linear system but gives a full basis of the set of solutions. Moreover, when the number of heavy columns is small, the cost of dealing with them becomes negligible. In particular, this eases the computation of discrete logarithms in medium and high characteristic finite fields, where *nearly sparse matrices* naturally occur.

Keywords. Sparse Linear Algebra. Block Wiedemann. Discrete Log. Finite Fields.

[1]This work has been supported in part by the European Union's H2020 Programme under grant agreement number ERC-669891.
Cécile Pierrot has been funded by Direction Générale de l'Armement and CNRS.

1.1 Introduction

Linear algebra is a widely used tool in both mathematics and computer science. At the boundary of these two disciplines, cryptography is no exception to this rule. Yet, one notable difference is that cryptographers mostly consider linear algebra over finite fields, bringing both drawbacks – the notion of convergence is no longer available – and advantages – no stability problems can occur. As in combinatory analysis or in the course of solving partial differential equations, cryptography also presents the specificity of frequently dealing with sparse matrices. For instance, sparse linear systems over finite fields appeared in cryptography in the late 70s when the first sub-exponential algorithm to solve the discrete logarithm problem in finite fields with prime order was designed [1]. Nowadays, every algorithm belonging to the Index Calculus family deals with a sparse matrix [11, Section 3.4]. Hence, since both Frobenius Representation Algorithms (for small characteristic finite fields) and discrete logarithm variants of the Number Field Sieve (for medium and high characteristics) belong to this Index Calculus family, all recent discrete logarithm records on finite fields need to find a solution of a sparse system of linear equations modulo a large integer. Similarly, all recent record-breaking factorizations of composite numbers, which are based on the Number Field Sieve, need to perform a sparse linear algebra step modulo 2.

A sparse matrix is a matrix containing a relatively small number of coefficients that are not equal to zero. It often takes the form of a matrix in which each row (or column) only has a small number of non-zero entries, compared to the dimension of the matrix. With sparse matrices, it is possible to represent in computer memory much larger matrices, by giving for each row (or column) the list of positions containing a non-zero coefficient, together with its value. When dealing with a sparse linear system of equations, using plain Gaussian Elimination is often a bad idea, since it does not consider nor preserve the sparsity of the input matrix. Indeed, each pivoting step during Gaussian Elimination may increase the number of entries in the matrix and, after a relatively small number of steps, it overflows the available memory.

Thus, in order to deal with sparse systems, a different approach is required. Three main families of algorithms have been devised: the first one adapts the ordinary Gaussian Elimination in order to choose pivots that minimize the loss of sparsity and is generally used to reduce the initial problem to a smaller and slightly less sparse problem. The two other al-

gorithm families work in a totally different way. Namely, they do not try to modify the input matrix but directly aim at finding a solution of the sparse linear system by computing only matrix-by-vector multiplications. One of these families consists of Krylov Subspace methods, adapted from numerical analysis, and constructs sequences of mutually orthogonal vectors. For instance, this family contains the Lanczos and Conjugate Gradient algorithms, adapted for the first time to finite fields in 1986 [7].

Throughout this article, we focus on the second family that contains Wiedemann algorithm and its generalizations. Instead of computing an orthogonal family of vectors, D. Wiedemann proposed in 1986 [20] to reconstruct the minimal polynomial of the considered matrix. This algorithm computes a sequence of scalars of the form ${}^twA^iv$ where v and w are two vectors and A is the sparse matrix of the linear algebra problem. It then tries to extract a linear recurrence relationship that holds for this sequence. In 1994, to achieve computations in realistic time, D. Coppersmith [6] adapted the Wiedemann algorithm over the finite field $\mathbb{F}_2$ for parallel and even distributed computations. One year later E. Kaltofen [12] not only generalized this algorithm to arbitrary finite fields but also gave a provable variant of Coppersmith's heuristic method. The main idea of Coppersmith's Block Wiedemann algorithm is to compute a sequence of matrices of the form ${}^tWA^iV$ where V and W are not vectors as previously but *blocks* of vectors. This step is parallelized by distributing the vectors of the block V to several processors or CPUs – let us say c of them. The asymptotic complexity of extracting the recursive relationships within the sequence of small matrices is in $\tilde{O}(cN^2)$ where N is the largest dimension of the input matrix. Another algorithm was presented by B. Beckerman and G. Labahn in 1994 [5] for performing the same task in subquadratic time and a further improvement was proposed by E. Thomé [19] in 2002: he reduced the complexity of finding the recursive relationships to $\tilde{O}(c^2N)$. The current fastest method is an application of the algorithm proposed by P. Giorgi, C-P. Jeannerod and G. Villard in 2003 [9] which runs in time $\tilde{O}(c^{\omega-1}N)$, where ω is the exponent of matrix multiplication. At the time of writing the best known[2] asymptotic value of this exponent is $\omega \approx 2.37286$. It comes from a slight improvement of Coppersmith-Winograd algorithm [8] due to F. Le Gall [14] and published in 2014.

Note that both Krylov Subspace methods and Wiedemann algorithms

[2] Yet, for practical purposes, asymptotically fast matrix multiplication is unusable and working implementations of the algorithm of Giorgi, Jeannerod and Villard have complexity $\tilde{O}(c^2N)$.

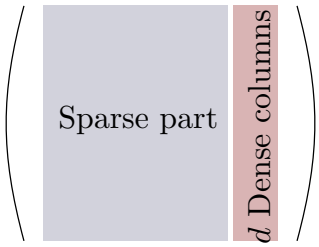


Fig. 1.1 A nearly sparse matrix

cost a number of matrix-by-vector multiplications equal to a small multiple of the matrix dimension: for a matrix containing λ entries per row in average, the cost of these matrix-by-vector multiplications is $O(\lambda N^2)$. With Block Wiedemann, it is possible to distribute the cost of these products up to c machines. In this case, the search for recursive relationships adds an extra cost of the form $\tilde{O}(c^{\omega-1}N)$. For a *nearly sparse matrix*, which includes d dense columns in addition to its sparse part, the cost of matrix-by-vector multiplications increases. As a consequence, the total complexity becomes $O((\lambda + d)N^2)$ with an extra cost of $\tilde{O}(c^{\omega-1}N)$ for Block Wiedemann. Figure 1.1 provides a quick overview of the structure of such nearly sparse matrices.

In this article, we aim at adapting the Coppersmith's Block Wiedemann algorithm to improve the cost of linear algebra on matrices that have nearly sparse properties and reduce it to $O(\lambda N^2) + \tilde{O}(\max(c, d)^{\omega-1}N)$. In particular, when the number of dense columns is lower than the number of processors used for the matrix-by-vector steps, we show that the presence of these unwelcome columns does not affect the complexity of solving linear systems associated to these matrices. In practice, this result precisely applies to the discrete logarithm problem. Indeed, nearly sparse matrices appear in both medium and high characteristic finite fields discrete logarithm computations. To illustrate this claim, we recall the latest record [4] announced in June 2014 for the computation of discrete logarithms in a prime field $\mathbb{F}_p$, where p is a 180 digit prime number. It uses a matrix containing 7.28M rows and columns with an average weight of 150 non-zero coefficients per row and also presents 4 dense Schirokauer maps columns. These columns precisely give to the matrix the nearly sparse structure we study in the sequel.

Outline. Section 1.2 makes a short recap on Coppersmith's Block Wiedemann algorithm, which is the currently best known algorithm to perform algebra on sparse linear systems while tolerating some amount of distributed computation. We propose in Section 1.4 the definition of a *nearly sparse matrix* and present then a rigorous algorithm to solve linear algebra problems associated to these matrices. In Section 1.4.7 we give a comparison of our method with preexisting linear algebra techniques and show that it is potentially competitive even with a tremendous number of dense columns. Section 1.5 ends by a practical application of this result: it explains how nearly sparse linear algebra eases discrete logarithm computations in medium and high characteristic finite fields.

1.2 A Reminder of Block Wiedemann Algorithm

This section first presents the classical problems of linear algebra that are encountered when dealing with sparse matrices. We then explain how the considered matrix is preconditioned into a square matrix. Section 1.2.2 draws the outline of the algorithm proposed by Wiedemann to solve linear systems given by a square matrix whereas Section 1.2.3 presents the parallelized variant due to Coppersmith. More precisely, the goal is to solve:

Problem 1.1. *Let $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$ be a prime finite field and $S \in \mathcal{M}_{n\times N}(\mathbb{K})$ be a (non necessarily square) sparse matrix with at most λ non-zero coefficients per row. Let $\vec{v}$ be a vector with n coefficients. The problem is to find a vector $\vec{x}$ with N coefficients such that $S\cdot\vec{x} = \vec{v}$ or, alternatively, a non-zero vector $\vec{x}$ such that $S\cdot\vec{x} = 0$.*

In practice, this problem is often generalized to rings $\mathbb{Z}/\mathcal{N}\mathbb{Z}$ for a modulus $\mathcal{N}$ of unknown factorization. However, for simplicity of exposition, and due to the fact that the algorithm of [9] is only proved over fields, we prefer to restrict ourselves to the prime field case.

1.2.1 *Preconditioning: making a sparse matrix square*

In order to be able to compute sequences of matrix-by-vector products of the form $(A^i\vec{y})_{i>0}$, both Wiedemann and Block Wiedemann algorithms need to work with a square matrix. Indeed, powers are only defined for square matrices. Consequently, if $N \neq n$, there is a necessary preliminary step to transform the given matrix into a square one. For example, it is

possible to pad the matrix with zeroes and then apply the analysis of [13] for solving linear systems which do not have full rank using Wiedemann's method. This is done by multiplying on the left and right by random matrices and then by truncating the matrix to a smaller invertible square matrix with the same rank as the original one.

In practice, heuristic methods are used instead. Typically, one creates a random sparse matrix $R \in \mathcal{M}_{N\times n}(\mathbb{K})$ with at most λ non-zero coefficients per row, and transform afterwards the two problems into finding a vector $\vec{x}$ such that $(RS)\vec{x} = R\vec{v}$ or, alternatively, such that $(RS)\vec{x} = 0$. Setting $A = RS$ and $\vec{y} = R\vec{v}$, we can rewrite Problem 1.1 as finding a vector $\vec{x}$ such that:

$$A \cdot \vec{x} = \vec{y}$$

or such that:

$$A \cdot \vec{x} = 0$$

depending on the initial problem. In addition, in order to avoid the trivial solution when solving $A\vec{x} = 0$, one frequently computes $\vec{y} = A\vec{r}$ for a random vector $\vec{r}$, solves $A\vec{x} = \vec{y}$ and outputs $\vec{x} - \vec{r}$ as a kernel element.

We do not go further into the details of how preconditioning is usually performed. Indeed, we propose in Section 1.4.2 a simple alternative technique, that provably works with our algorithm for nearly sparse matrices, under some explicit technical conditions. Since sparse matrices are a special case of nearly sparse matrices, this alternative would also work for usual sparse matrices.

1.2.2 *Wiedemann algorithm*

Let us now consider a square matrix A of size $N \times N$ and denote m_A the number of operations required to compute the product of a vector of $\mathbb{K}^N$ by A. Wiedemann algorithm works by finding a non-trivial sequence of coefficients $(a_i)_{0\leq i\leq N}$ such that:

$$\sum_{i=0}^{N} a_i A^i = 0. \tag{1.1}$$

Solving $A\vec{x} = \vec{y}$. If A is invertible, then we can assume $a_0 \neq 0$. Indeed, if $a_0 = 0$ we can rewrite $0 = \sum_{i=1}^{N} a_i A^i = A^\delta(\sum_{i=1}^{N} a_i A^{i-\delta})$ where a_δ is the first non zero coefficient. Multiplying by $(A^{-1})^\delta$ it yields the equality

Algorithm 1.1 Wiedemann algorithm for $A\vec{x} = \vec{y}$

Input: A matrix A of size $N \times N$, $\vec{y} \neq 0$ a vector with N coefficients
Output: $\vec{x}$ such that $A \cdot \vec{x} = \vec{y}$.

Computing a sequence of scalars
$\vec{v}_0 \leftarrow\in \mathbb{K}^N$, $\vec{w} \leftarrow\in \mathbb{K}^N$ two random vectors
for $i = 0, \cdots, 2N$ **do**
  $\lambda_i \leftarrow \vec{w} \cdot \vec{v}_i$
  $\vec{v}_{i+1} \leftarrow A\vec{v}_i$
end for
Berlekamp-Massey algorithm
From $\lambda_0, \cdots, \lambda_{2N}$ recover coefficients $(a_i)_{0 \leq i \leq N}$ *s.t.* $\sum_{i=0}^{N} a_i A^i = 0$ and $a_0 \neq 0$.
Resolution
return $-(1/a_0) \sum_{i=1}^{N-1} a_{i+1} A^i \vec{y}$.

$\sum_{i=0}^{N-\delta} a_{i+\delta} A^i = 0$. So, shifting the coefficients until we find the first non-zero one allows to write $a_0 \neq 0$. Let us apply Equation (1.1) to the vector $\vec{x}$ we are seeking. It yields $-a_0\vec{x} = \sum_{i=1}^{N} a_i A^i \vec{x} = \sum_{i=1}^{N} a_i A^{i-1}(A\vec{x})$. Finally we recover $\vec{x} = -(1/a_0) \sum_{i=1}^{N} a_i A^{i-1}\vec{y}$. This last sum can be computed using N sequential multiplications of the initial vector $\vec{y}$ by the matrix A. The total cost to compute $\vec{x}$ as this sum is $O(N \cdot m_A)$ operations.

Solving $A\vec{x} = 0$. Assuming that there exists a non-trivial element of the kernel of A, we deduce that $a_0 = 0$. Let again δ be the first index such that $a_\delta \neq 0$. Thus, for any vector $\vec{r}$ we have $0 = \sum_{i=\delta}^{N} a_i A^i \vec{r} = A^\delta(\sum_{i=\delta}^{N} a_i A^{i-\delta}\vec{r})$. We know that $\sum_{i=\delta}^{N} a_i A^{i-\delta} \neq 0$. Indeed, otherwise, $a_\delta \mathrm{Id} + \sum_{i=\delta+1}^{N} a_i A^{i-\delta} = a_\delta \mathrm{Id} + A(\sum_{i=\delta+1}^{N} a_i A^{i-\delta-1}) = 0$ would lead to $A(-(1/a_\delta) \sum_{i=\delta+1}^{N} a_i A^{i-\delta-1}) = \mathrm{Id}$, yet A is assumed non invertible. Thus, for a random vector $\vec{r}$, the sum $\sum_{i=\delta}^{N} a_i A^{i-\delta}\vec{r}$ is non zero with high probability: this vector is the zero vector if and only if $\vec{r}$ belongs to the kernel of the non null matrix $\sum_{i=\delta}^{N} a_i A^{i-\delta}$. Since the kernel of a non null matrix has at most dimension $N-1$, the probability for a random vector to be in its kernel is upper bounded by $|\mathbb{K}|^{N-1}/|\mathbb{K}|^N = 1/|\mathbb{K}|$.

Now, computing iteratively $A(\sum_{i=\delta}^{N} a_i A^{i-\delta}\vec{r})$, $A^2(\sum_{i=\delta}^{N} a_i A^{i-\delta}\vec{r})$, $\cdots, A^\delta(\sum_{i=\delta}^{N} a_i A^{i-\delta}\vec{r})$ yields an element of the kernel of A in $O(N \cdot m_A)$ operations as well. Indeed, the first index j in $[\![1, \delta]\!]$ such that $A^j(\sum_{i=\delta}^{N} a_i A^{i-\delta}\vec{r}) = 0$ shows that $A^{j-1}(\sum_{i=\delta}^{N} a_i A^{i-\delta}\vec{r}) \neq 0$ belongs to the kernel of A. Thus, this method finds a non trivial element of $\mathrm{Ker}(A)$ with probability higher than $(|\mathbb{K}| - 1)/|\mathbb{K}|$, which quickly tends to 1 as the cardinality of the field grows.

How to find coefficients a_i verifying Equation (1.1)**.** Cayley-Hamilton theorem testifies that the polynomial defined as $P = \det(A - X \cdot \mathrm{Id})$ annihilates the matrix A, *i.e.* $P(A) = 0$. So we know that there exists a polynomial of degree at most N whose coefficients satisfy Equation (1.1). Yet, directly computing such a polynomial would be too costly. The idea of Wiedemann algorithm is, in fact, to process by necessary conditions.

Let $(a_i)_{i \in [\![0,N]\!]}$ be such that $\sum_{i=0}^{N} a_i A^i = 0$. Then, for any arbitrary vector $\vec{v}$ we obtain $\sum_{i=0}^{N} a_i A^i \vec{v} = 0$. Again, for any arbitrary vector $\vec{w}$ and for any integer j we can write $\sum_{i=0}^{N} a_i {}^t\vec{w} A^{i+j} \vec{v} = 0$. Conversely, if $\sum_{i=0}^{N} a_i {}^t\vec{w} A^{i+j} \vec{v} = 0$ for any random vectors $\vec{v}$ and $\vec{w}$ and for any j in $[\![0, N]\!]$ then the probability to obtain coefficients verifying Equation (1.1) is high, assuming the cardinality of the field is sufficiently large [12]. Thus, Wiedemann algorithm seeks coefficients a_i that annihilate the sequence of scalars ${}^t\vec{w} A^i \vec{v}$. To do so, it can use the classical Berlekamp-Massey algorithm [2, 16] that finds the minimal polynomial of a recursive linear sequence in an arbitrary field. In a nutshell, the idea is to consider the generating function f of the sequence ${}^t\vec{w}\vec{v}, {}^t\vec{w}A\vec{v}, {}^t\vec{w}A^2\vec{v}, \cdots, {}^t\vec{w}A^{2N}\vec{v}$ and to find afterwards two polynomials g and h such that $f = g/h \mod X^{2N}$. Alternatively, the Berlekamp-Massey algorithm can be replaced by a half extended Euclidean algorithm, yielding a quasi-linear algorithm in the size of the matrix A.

1.2.3 *Coppersmith's Block Wiedemann algorithm*

The Block Wiedemann algorithm is a parallelization of the previous Wiedemann algorithm introduced by Don Coppersmith. It targets the context where sequences of matrix-vector products are computed on ℓ processors, instead of one. In this case, rather than solving Equation (1.1), it searches, given ℓ vectors $\vec{v}_1, \cdots, \vec{v}_\ell$, for coefficients a_{ij} such that:

$$\sum_{j=1}^{\ell} \sum_{i=0}^{\lceil N/\ell \rceil} a_{ij} A^i \vec{v}_j = 0 \tag{1.2}$$

Note that the number of coefficients remains approximately the same as in the previous algorithm.

Solving $A\vec{x} = \vec{0}$. There, we choose ℓ random vectors $\vec{r}_1, \cdots, \vec{r}_\ell$ and set $\vec{v}_i = A\vec{r}_i$. Let δ denote the first index in $[\![1, \lceil N/\ell \rceil]\!]$ such that there exists j in $[\![1, \ell]\!]$ satisfying $a_{\delta j} \neq 0$. Equation (1.2) gives $\sum_{j=1}^{\ell} \sum_{i=\delta}^{\lceil N/\ell \rceil} a_{ij} A^{i+1} \vec{r}_j = \vec{0}$, *i.e.* $A^{\delta+1}(\sum_{j=1}^{\ell} \sum_{i=\delta}^{\lceil N/\ell \rceil} a_{ij} A^{i-\delta} \vec{r}_j) = \vec{0}$. Let $\vec{b}$ denote the vector

Algorithm 1.2 Block Wiedemann algorithm for $A\vec{x} = \vec{0}$

Input: A matrix A of size $N \times N$
Output: $\vec{x}$ such that $A \cdot \vec{x} = \vec{0}$.

Computing a sequence of matrices

$\vec{r}_1 \leftarrow\in \mathbb{K}^N, \cdots, \vec{r}_\ell \leftarrow\in \mathbb{K}^N$ and $\vec{w}_1 \leftarrow\in \mathbb{K}^N, \cdots, \vec{w}_\ell \leftarrow\in \mathbb{K}^N$
$\vec{v}_1 \leftarrow A\vec{r}_1, \cdots, \vec{v}_\ell \leftarrow A\vec{r}_\ell$
for any of the ℓ processors indexed by j **do**
  $u_0 \leftarrow v_j$
  for $i = 0, \cdots, 2\lceil N/\ell \rceil$ **do**
    for $k = 1, \cdots, \ell$ **do**
      $\lambda_{i,j,k} \leftarrow \vec{w}_k \cdot \vec{u_i}$
      $\vec{u_{i+1}} \leftarrow A\vec{u_i}$
    end for
  end for
end for
for $i = 0, \cdots, 2\lceil N/\ell \rceil$ **do**
  $M_i \leftarrow (\lambda_{i,j,k})$ the $\ell \times \ell$ matrix containing all the products of the form ${}^t\vec{w}A^i\vec{v}$
end for

Thomé or Giorgi, Jeannerod, Villard's algorithm

From $M_0, \cdots, M_{2\lceil N/\ell \rceil}$ recover coefficients a_{ij} *s.t.* $\sum_{j=1}^{\ell} \sum_{i=0}^{\lceil N/\ell \rceil} a_{ij} A^i \vec{v}_j = \vec{0}$.

Resolution

$\delta \leftarrow$ the first index in $[\![1, \lceil N/\ell \rceil]\!]$ such that there exists j in $[\![1, \ell]\!]$ satisfying $a_{\delta j} \neq 0$.
$\vec{b} \leftarrow \sum_{j=1}^{\ell} \sum_{i=\delta}^{\lceil N/\ell \rceil} a_{ij} A^i \vec{r}_j$.
$\vec{k} \leftarrow$ Error: trivial kernel element
while $\vec{b} \neq 0$ **do**
  $\vec{k} \leftarrow \vec{b}$
  $\vec{b} \leftarrow A\vec{k}$
end while
return $\vec{k}$

$\sum_{j=1}^{\ell} \sum_{i=\delta}^{\lceil N/\ell \rceil} a_{ij} A^i \vec{r}_j$. According to [12], $\vec{b}$ is non zero with high probability. Hence, computing iteratively $A\vec{b}$, $A^2\vec{b}$, $\cdots$, $A^\delta\vec{b}$ yields an element of the kernel of A in $O(N \cdot m_A)$ operations again. Indeed, the first index k in $[\![1, \delta]\!]$ such that $A^k\vec{b} = 0$ shows that $A^{k-1}\vec{b}$ is a non trivial element of the kernel of A.

Solving $A\vec{x} = \vec{y}$. In order to solve $A\vec{x} = \vec{y}$, several different approaches are possible. For example, in [12] the size of A is increased by 1, adding $\vec{y}$ as an new column and adding a new zero row. It is then explained that a random kernel element, as produced by the above method, involves $\vec{y}$ and thus produces a solution of $A\vec{x} = \vec{y}$.

Another option is to set $\vec{v}_1 = \vec{y}$ and choose for $i \in [\![2, \ell]\!]$ the vectors $\vec{v}_i = A\vec{r}_i$, where each $\vec{r}_i$ is a random vector of the right size and to assume

that $a_{01} \neq 0$. From Equation (1.2) we derive:

$$\sum_{i=0}^{\lceil N/\ell \rceil} a_{i1} A^i \vec{y} + \sum_{j=2}^{\ell} \sum_{i=0}^{\lceil N/\ell \rceil} a_{ij} A^{i+1} \vec{r}_j = 0.$$

Multiplying by the inverse of A, we obtain:

$$a_{01} \vec{x} + \sum_{i=1}^{\lceil N/\ell \rceil} a_{i1} A^i \vec{y} + \sum_{j=2}^{\ell} \sum_{i=0}^{\lceil N/\ell \rceil} a_{ij} A^i \vec{r}_j = 0.$$

Thus, we can recover $\vec{x}$ by computing:

$$(-1/a_{01}) \cdot \left(\sum_{i=1}^{\lceil N/\ell \rceil} a_{i1} A^{i-1} \vec{y} + \sum_{j=2}^{\ell} \sum_{i=0}^{\lceil N/\ell \rceil} a_{ij} A^i \vec{r}_j \right).$$

This can be done with a total cost of $O(N \cdot m_A)$ operations parallelized over the ℓ processors: each one is given one starting vector $\vec{v}$ and computes a sequence of matrix-by-vector products of the form $A^i \vec{v}$. The cost for each sequence is $O(N \cdot m_A/\ell)$ arithmetic operations. We do not deal here with the case where $a_{01} = 0$ since Section 1.4 covers all cases for nearly sparse matrices, thus for sparse matrices.

1.2.4 *How to find coefficients a_i verifying Equation* (1.2).

Let $\vec{v}_1, \cdots, \vec{v}_\ell$ be ℓ vectors and let consider the $\ell(\lceil N/\ell \rceil)$ elements obtained by the matrix-by-vector products of the form $A^i \vec{v}_j$ that appear in the sum of Equation (1.2). Since $\ell(\lceil N/\ell \rceil) > N$, all these vectors cannot be independent, so there exist coefficients satisfying (1.2). As for Wiedemann algorithm, we process by necessary conditions. More precisely, let $\vec{w}_1, \ldots, \vec{w}_\ell$ be ℓ vectors. Assume that for any κ in $[\![0, \lceil N/\ell \rceil]\!]$ and k in $[\![1, \ell]\!]$ we have $\sum_{j=1}^{\ell} \sum_{i=0}^{\lceil N/\ell \rceil} a_{ij} {}^t\vec{w}_k A^{i+\kappa} \vec{v}_j = 0$, then the probability that the coefficients a_{ij} verify Equation (1.2) is close to 1 when $\mathbb{K}$ is large[3] (again see [12]). So Block Wiedemann algorithm looks for coefficients that annihilate the sequence of $2\lceil N/\ell \rceil$ small matrices of dimension $\ell \times \ell$ computed as $({}^t\vec{w}_k A^\nu \vec{v}_j)$. Here, $\nu \in [\![0, 2\lceil N/\ell \rceil]\!]$ numbers the matrices, while k and j respectively denote the column and row numbers within each matrix. It is possible to compute the coefficients a_{ij} in subquadratic time (see Section 1.3 for details). For instance, Giorgi, Jeannerod, Villard give an efficient method with complexity $\tilde{O}(\ell^{\omega-1} N)$. This is the final component needed to write Block Wiedemann as Algorithm 1.2.

[3] When $\mathbb{K}$ is small, it is easy to make the probability close to 1 by increasing the number of vectors w beyond ℓ in the analysis as done in [6].

Moreover, putting together the matrix-by-vector products and the search for coefficients, the overall complexity can be expressed as $O(N \cdot m_A) + \tilde{O}(\ell^{\omega-1}N)$. Where the $O(N \cdot m_A)$ part can be distributed on up to ℓ processors and the $\tilde{O}(\ell^{\omega-1}N)$ part is computed sequentially.

Remark 1.1. In this section, we assumed that the number of sequences ℓ is equal to the number of processors c. This is the most natural choice in the classical application of Block Wiedemann, since increasing ℓ beyond the number of processors can only degrades the overall performance. More precisely, the change leaves the $O(N \cdot m_A)$ contribution unaffected but increases $\tilde{O}(\ell^{\omega-1}N)$. However, since values of ℓ larger than c are considered in Section 1.4, it is useful to know that this can be achieved by sequentially computing several independent sequences on each processor. In this case, it is a good idea in practice to make the number of sequences a multiple of the number of processors, in order to minimize the wall clock running time of the matrix-by-vector multiplications.

1.3 Minimal basis computations

In this section, we recall an important result of Giorgi, Jeannerod and Villard [9], used in Section 1.2 for presenting Block Wiedemann. This result is a key ingredient for the algorithm we describe in Section 1.4. Let $\mathbb{K}$ be a finite field and G a matrix of power series over $\mathbb{K}$ of dimension $m \times n$ with $n < m$, i.e. an element of $K[[X]]^{m\times n}$. For an approximation order b, we consider m-dimensional row vectors $\vec{u}(X)$ of polynomials that satisfy the equation:

$$\vec{u} \cdot G \equiv \vec{0} \mod X^b. \tag{1.3}$$

For a vector of polynomial, we define its degree $\deg(\vec{u})$ as the maximum of the degree of the coordinates of $\vec{u}$.

Definition 1.1. A σ-basis of the set of solutions of Equation (1.3) is a square $m \times m$ matrix M of polynomials of $\mathbb{K}[X]$ such that:

- Every row vector $\vec{M_i}$ of M satisfies (1.3).
- For every solution $\vec{u}$ of (1.3), there exists a unique family of m polynomials $c_1, \ldots, c_m$ such that for each i:
$$\deg(c_i\vec{M_i}) \leq \deg(\vec{u}),$$
that, in addition, satisfies the relation:
$$\vec{u} = \sum_{i=1}^{m} c_i\vec{M_i}.$$

Giorgi, Jeannerod and Villard give an algorithm that computes a σ-basis for Equation (1.3) using $\tilde{O}(m^\omega b)$ algebraic operations in $\mathbb{K}$. Note that for practical implementations, especially with small values of m, ω should be replaced by 3, thus matching the complexity of the related algorithm given by Thomé [19].

1.4 Nearly Sparse Linear Algebra

In this section, our aim is twofold. We first aim at adapting the Block Wiedemann algorithm to improve the resolution of some linear algebra problems that are not exactly sparse but close enough to be treated similarly. We also give more precise conditions on the choices of the vectors $\vec{v}_i$ and $\vec{w}_i$ that are made in Block Wiedemann algorithm. Rather than insisting on random choices as in [12] we give explicit conditions on these choices. When the conditions are satisfied, we show that our algorithm not only recovers a random solution of the linear system of equations given as input but, in fact, gives an explicit description of the full set of solutions.

The cornerstone of our method consists in working with the sparse part of the matrix while forcing part of the initial vectors of the sequences computed by Block Wiedemann algorithm to be derived from the dense columns of the matrix in addition to random initial vectors. In the rest of this section, we describe this idea in details.

1.4.1 *Nearly sparse matrices*

In the sequel we focus on linear algebra problems of the following form:

Problem 1.2. *Let M be a matrix of size $N \times (s+d)$ with coefficients in a field $\mathbb{K}$. We assume that there exist two smaller matrices $M_s \in \mathcal{M}_{N\times s}(\mathbb{K})$ and $M_d \in \mathcal{M}_{N\times d}(\mathbb{K})$ such that :*

(1) $M = M_s|M_d$, where $|$ is the concatenation of matrices.[4]
(2) M_d is arbitrary.
(3) M_s is sparse. Let us assume it has at most λ non-zero coefficients per row.

If $\vec{y}$ is a given vector with N coefficients, the problem is to find all vectors $\vec{x}$

[4] Our method would also work for matrices with d dense columns located at any position. It would suffice to reorder the columns of the linear algebra problem. However, for simplicity of exposition, we assume the dense columns are the d final columns of M.

with $s + d$ coefficients such that:

$$M \cdot \vec{x} = \vec{y}$$

or, alternatively, such that:

$$M \cdot \vec{x} = \vec{0}.$$

Such a matrix M is said to be d-nearly sparse, or as a shortcut, simply nearly sparse when d is implied by context. Note that, in our definition, there is no a priori *restriction on the number of dense columns that appear in the matrix.*

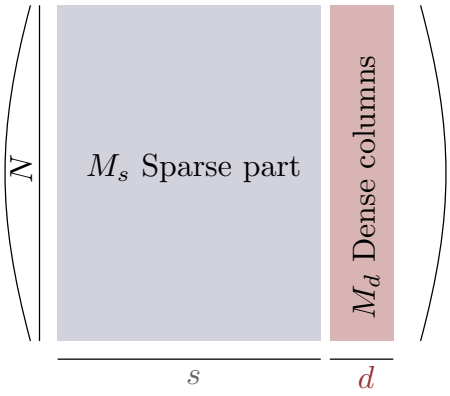


Fig. 1.2 Parameters of the nearly spare linear algebra problem

An interesting consequence of the fact that we want to construct all the solutions of these linear algebra problems is that we only need to deal with the second (homogeneous) sub-problem of Problem 1.2. Indeed, it is easy to transform the resolution of $M \cdot \vec{x} = \vec{y}$ into the resolution of $M' \cdot \vec{x'} = \vec{0}$ for a nearly sparse matrix M' closely related to M. It suffices to set $M' = M|\vec{y}$ the matrix obtained by concatenating one additional dense column equal to $\vec{y}$ to the right of M. Now we see that $\vec{x}$ is a solution of $M \cdot \vec{x} = \vec{y}$ if and only if $\vec{x'} = {}^t({}^t\vec{x}| - 1)$ is a solution of $M' \cdot \vec{x'} = \vec{0}$. Keeping this transformation in mind, in the sequel we only explain how to compute a basis of the kernel of a nearly sparse matrix. When solving the first (affine) sub-problem, we just need at the end to select in the kernel of M' the vectors with a -1 in the last position.

Thus, the two variants that appear in Problem 1.2 are more directly related in our context than their counterparts in Problem 1.1 are in the context of the traditional (Block) Wiedemann algorithm.

With such a nearly sparse matrix M, it would of course be possible to directly apply the usual Block Wiedemann algorithm and find a random element of its kernel. However, in this case, the cost of multiplying a vector by the matrix M becomes larger, of the order of $(\lambda + d)N$ operations. As a consequence, the complete cost of the usual Block Wiedemann algorithm becomes $O((\lambda + d) \cdot N^2) + \tilde{O}(\ell^{\omega-1}N)$ when using ℓ processors.

Figure 1.3 gives a roadmap of the various steps we go through in order to obtain an efficient bijection between the kernel of M and a subset of the solutions resulting from a minimal basis computation using the algorithm of Giorgi, Jeannerod and Villard.

1.4.2 *Preconditioning for a nearly sparse matrix*

If $N = s$, the matrix is already square and nothing needs to be done. Note the case $N < s$ does not usually appear in applications such as discrete logarithm computations where extra equations can easily be added. In the rare event where this case would appear, the simplest approach to deal with it is probably to artificially move $s - N$ columns from the sparse part to the dense part of the matrix. After this, the dense part becomes square $(N \times N)$ while the number of columns in M_d increases to $d' = d + s - N$.

So in the sequel we focus on the case where the sparse part of M has more rows than columns, namely $N > s$. To turn the sparse part of M into a square matrix, a simple method consists in embedding the rectangular matrix M_s into a square one A by adding $N - s$ zero columns[5] at the right side of M_s.

Finding an element ${}^t(x_s, x_d)$ in the kernel of M is equivalent to finding a longer vector ${}^t(x_s, x_{tra}, x_d)$ in the kernel of $(A|M_d)$, where x_s, x_{tra} and x_d are respectively row vectors of $\mathbb{K}^s$, $\mathbb{K}^{N-s}$ and $\mathbb{K}^d$ (x_{tra} denotes the extraneous coordinates).

In the sequel, we focus on the matrix A regardless of how it has been constructed.

1.4.3 *Preliminary transformations with conditions*

We set B an integer to be determined later (see Section 1.4.5), let $\vec{\delta}_1$, $\ldots$, $\vec{\delta}_d$ denote the column vectors of M_d and choose $\ell - d$ random vectors

[5] Although this zero-padding method fits the theoretical analysis well, other randomized preconditionning methods are also used in practice.

$\vec{r}_{d+1}, \cdots, \vec{r}_\ell$ in $\mathbb{K}^N$. From these vectors, we construct the family:

$$\mathcal{F} := \left\{ \begin{array}{c} \vec{\delta}_1, A\vec{\delta}_1, \cdots, A^{B-1}\vec{\delta}_1, \cdots, \vec{\delta}_d, A\vec{\delta}_d, \cdots, A^{B-1}\vec{\delta}_d, \\ \vec{r}_{d+1}, A\vec{r}_{d+1}, \cdots, A^{B-1}\vec{r}_{d+1}, \cdots, \vec{r}_\ell, A\vec{r}_\ell, \cdots, A^{B-1}\vec{r}_\ell \end{array} \right\}.$$

Our first condition is the assumption that $\mathcal{F}$ generates the full vector space $\mathbb{K}^N$. We discuss the validity of this assumption in Section 1.4.4.1.

Condition on V.

To initialize the ℓ sequences the main idea is to force the first d ones to start from the d dense columns of M_d. In other words, by setting $\vec{v}_i = \vec{\delta}_i$ for i in $[\![1, d]\!]$ and $\vec{v}_i = \vec{r}_i$ for in $[\![d+1, \ell]\!]$. Then, we see that the assumption on $\mathcal{F}$ can be rewritten as:

$$\text{Vect}\left(\{A^i\vec{v}_j|^{\,i\,=\,0,\,\cdots\,,\,B\,-\,1}_{\,j\,=\,1,\,\cdots\,,\,\ell}\}\right) = \mathbb{K}^N. \tag{1.4}$$

Let ${}^t({}^t\vec{x}|x'_1|\cdots|x'_d)$ be a vector in the kernel of $(A|M_d)$. If Equation (1.4) is satisfied, there exist, in particular, coefficients $\lambda_{ij} \in \mathbb{K}$ such that:

$$\vec{x} = \sum_{j=1}^{\ell}\sum_{i=0}^{B-1} \lambda_{ij} A^i \vec{v}_j$$

Thus we obtain:

$$\begin{aligned} (A|M_d)\,{}^t({}^t\vec{x}|x'_1|\cdots|x'_d) &\Leftrightarrow \\ A\sum_{j=1}^{\ell}\sum_{i=0}^{B-1} \lambda_{ij} A^i\vec{v}_j + M_d\,{}^t(x'_1|\cdots|x'_d) = \vec{0} &\Leftrightarrow \\ \sum_{j=1}^{d}\sum_{i=1}^{B} \lambda_{(i-1)j} A^i\vec{\delta}_j + \sum_{j=d+1}^{\ell}\sum_{i=1}^{B} \lambda_{(i-1)j} A^i\vec{r}_j + \sum_{j=1}^{d} x'_j\vec{\delta}_j = \vec{0} &\Leftrightarrow \\ \sum_{j=1}^{\ell}\sum_{i=0}^{B} a_{ij} A^i\vec{v}_j = \vec{0} & \end{aligned}$$

where the coefficients a_{ij} are defined through:

$$a_{ij} = \left\{ \begin{array}{ll} \lambda_{(i-1)j} & \text{if } i > 0. \\ x'_j & \text{if } i = 0 \text{ and } j \leq d. \\ 0 & \text{if } i = 0 \text{ and } j > d. \end{array} \right.$$

To put it in a nutshell, as soon as the condition given by Equation (1.4) on the matrix $V = (\vec{v}_1|\cdots|\vec{v}_\ell)$ is verified, every element of the kernel of $(A|M_d)$ gives a solution of:

$$\sum_{j=1}^{\ell}\sum_{i=0}^{B} a_{ij} A^i\vec{v}_j = \vec{0}, \tag{1.5}$$

where the coefficients a_{0j} are zeroes for $j > d$. Conversely (whether or not condition (1.4) is satisfied) any solution of Equation (1.5) with zeroes on these positions yields an element of the kernel of $(A|M_d)$. Thus, under condition (1.4), determining the kernel of $(A|M_d)$ is equivalent to finding a basis of the solutions of Equation (1.5) with the $\ell - d$ aforementioned zeroes.

Condition on W.

Of course, Equation (1.5) can be seen as a system of N linear equations over $\mathbb{K}$. However, solving it directly would not be more efficient than directly computing the kernel of M. Instead, we remark that for any matrix $W = (\vec{w}_1|\cdots|\vec{w}_\ell)$ consisting in ℓ columns of vectors in $\mathbb{K}^N$, a solution (a_{ij}) of Equation (1.5) leads to a solution of:

$$\sum_{j=1}^{\ell}\sum_{i=0}^{B} a_{ij}\,{}^t\vec{w}_k A^{i+\kappa}\vec{v}_j = 0 \tag{1.6}$$

for any $k \in [\![1, \ell]\!]$ and any $\kappa \in \mathbb{N}$.

In the reverse direction, assume that we are given a solution $(a_{ij})_{i\in[\![1,\ell]\!]}^{j\in[\![0,B]\!]}$ that satisfies Equation (1.6) for all $k \in [\![1, \ell]\!]$ and for all $\kappa \in [\![0, B-1]\!]$. Now assume that:

$$\mathrm{Vect}\left(\{{}^t\vec{w}_j A^i |\, {}^{i\,=\,0,\,\cdots\,,\,B\,-\,1}_{j\,=\,1,\,\cdots\,,\,\ell}\}\right) = \mathbb{K}^N. \tag{1.7}$$

Under this condition, a_{ij} is also a solution of Equation (1.5). Indeed, by assumption, the vector $\sum_{j=1}^{\ell}\sum_{i=0}^{B} a_{ij}A^i\vec{v}_j$ is orthogonal to every vector in the basis of $\mathbb{K}^N$ listed in condition (1.7). Thus, it must be the zero vector.

Rewriting Equation (1.6) *with matrix power series.*

For a fixed value of κ, we can paste together the ℓ copies of Equation (1.6) for $k \in [\![1, \ell]\!]$. In order to do this, let $\vec{a_i}$ denote the vector ${}^t(a_{i1}, a_{i2}, \cdots, a_{il})$. With this notation, the ℓ equations can be grouped as:

$$\sum_{i=0}^{B}({}^tWA^{i+\kappa}V)\cdot\vec{a_i} = \vec{0}. \tag{1.8}$$

Let us now define the matrix power series $S(X)$ and the vector polynomial $P(X)$ as follows:

$$S(X) = \sum_{i\in\mathbb{N}}({}^tWA^iV)X^i \quad \text{and} \quad P(X) = \sum_{i=0}^{B}\vec{a_i}X^{B-i}.$$

Consider the product of $S(X)$ by $P(X)$. By definition of the multiplication for power series, we see that the coefficient corresponding to the monomial $X^{B+\kappa}$ in the product $S(X)P(X)$ is $\sum_{i=0}^{B}({}^tWA^{i+\kappa}V)\cdot\vec{a_i}$. According to Equation (1.8), this is $\vec{0}$ for all $\kappa\in\mathbb{N}$.

As a consequence, the vector power series $S(X)P(X)$ is in fact a vector polynomial $Q(X)$ of degree at most $B-1$. Thus, given $S(X)$ we search for vector polynomials $P(X)$ and $Q(X)$ of respective degrees at most B and at most $B-1$ such that $S(X)P(X)-Q(X)=\vec{0}$. To fit into the notations of Section 1.3, define $G(X)=(S(X)|-\mathrm{Id}(X))$ to be the $\ell\times 2\ell$ matrix power series formed by concatenating the opposite of the $\ell\times\ell$ identity matrix to $S(X)$. Denote $\vec{u}(X)$ the dimension 2ℓ row vector obtained by concatenating ${}^tP(X)$ and ${}^tQ(X)$. We now have $G(X){}^t\vec{u}=\vec{0}$, transpose and obtain:

$$\vec{u}(X)\cdot{}^tG(X)=\vec{0}. \tag{1.9}$$

Note that $\vec{u}(X)$ has degree at most B on its first ℓ coordinates and degree at most $B-1$ on the other coordinates. Furthermore, knowing that the coefficients a_{0j} are zeroes for $j>d$ leads to a zero constant coefficient for all polynomial coordinates from $d+1$ to ℓ.

In order to use the algorithm of Giorgi, Jeannerod and Villard, we prefer to work modulo a large monomial instead of dealing with power series. Indeed, we clearly have $\vec{u}(X)\cdot{}^tG(X)=\vec{0}$ modulo X^b for any integer b, with the same three constraints on $\vec{u}(X)$. We analyze the value of b permitting to claim that a solution of Equation (1.9) modulo X^b, and with the same constraints on the vector $\vec{u}$, can be transformed back into a solution of Equation (1.6) for any integer κ in $[\![0,B-1]\!]$.

Let us assume we have a vector $\vec{u}={}^t(u^{(1)},\cdots,u^{(2\ell)})$ solution of Equation (1.9) modulo X^b with:

- $\forall i\in[\![1,\ell]\!]$, $\deg u^{(i)}(X)\leq B$,
- $\forall i\in[\![\ell+1,2\ell]\!]$, $\deg u^{(i)}(X)\leq B-1$,
- $\forall i\in[\![d+1,\ell]\!]$, $u^{(i)}(0)=0$.

Since $\vec{u}$ consists of 2ℓ polynomials we can cut it into two separate parts and consider only its first ℓ polynomial terms, that are of degree at most B. There exists a canonical correspondence between this vector of polynomials and a polynomial $P(X)$ of degree at most B where the coefficients are vectors in $\mathbb{K}^\ell$. Writing $P(X)=\sum_{i=0}^{B}\vec{z}_iX^i$ with $\vec{z}_i\in\mathbb{K}^\ell$ we can define for all i in $[\![0,B]\!]$ and all j in $[\![1,\ell]\!]$:

$$a_{ij}=\text{the } j\text{-th coordinate of the vector } \vec{z}_{B-i}.$$

Since modulo X^b the product $P(X)S(X)$ is a vector polynomial of degree at most $B-1$ we deduce that for any integer κ such that $0 \leq \kappa \leq b-B-1$ the coefficient in $P(X)S(X)$ related to the monomial $X^{B+\kappa}$ is the zero vector. Combining with $P(X) = \sum_{i=0}^{B} \vec{a_i} X^{B-i}$ and $S(X) = \sum_{i\in\mathbb{N}} ({}^tWA^iV)X^i$ it leads to $\sum_{i=0}^{B} ({}^tWA^{i+\kappa}V) \cdot \vec{a_i} = \vec{0}$. Hence multiplying V by the vectors $\vec{a_i}$ we get $\sum_{j=1}^{\ell} \sum_{i=0}^{B} {}^tWA^{i+\kappa}(a_{ij}\vec{v}_j) = \vec{0}$. Finally, if we consider each row ${}^t\vec{w}_k$ with k in $[\![1, \ell]\!]$ and any κ in $[\![0, b-B-1]\!]$ we obtain coefficients a_{ij} that are solutions of Equation (1.6). Thus, to get Equation (1.6) for any κ in $[\![0, B-1]\!]$ it suffices to set $b = 2B$.

Summary of the transformations.

To sum it up, we have transformed the problem of finding the kernel of M into the problem of finding all solutions of Equation (1.9) modulo X^{2B}, with degree at most B on the first ℓ coordinates, degree at most $B-1$ on the other coordinates and a zero constant coefficient for coordinates $d+1$ to ℓ. Under the two conditions (1.4) and (1.7), the above analysis directly gives a bijection between the set of solutions of the two problems, as illustrated in Figure 1.3.

1.4.4 *Applying Giorgi, Jeannerod and Villard algorithm.*

Thanks to Giorgi, Jeannerod and Villard [9] we can compute a minimal σ-basis of the solution vectors of Equation (1.9) modulo X^{2B} in time $\tilde{O}(\ell^\omega B)$. However, we need to post-process this σ-basis to recover a basis of the kernel of M. More precisely, we need to derive an explicit description of all solution vectors of Equation (1.9) that have degree at most B on the first ℓ coordinates, degree at most $B-1$ on the last ℓ coordinates and a zero constant coefficient for coordinates $d+1$ to ℓ. We first show how to obtain all solution vectors that have degree at most B on the 2ℓ coordinates. A final filtering is then used to ensure that the stronger degree bound on the last ℓ coordinates holds and the $\ell - d$ constant coefficients are zeroes.

We first let $\vec{b_1}, \cdots, \vec{b_t}$ denote the t vectors[6] in the σ-base with degree at most B. Let $\vec{u}$ denote any solution vector of Equation (1.9) with degree at most B. From the minimality of the σ-base, we know that $\vec{u}$ can be written as linear combinations $\sum_{i=1}^{t} c_i \vec{b}_i$ where the c_i are polynomials in $\mathbb{K}[X]$ such that $\deg c_i + \deg b_i \leq \deg \vec{u}$ for any $i = 1, \cdots, t$. Thus, the set

[6] At most, there are 2ℓ such vectors. Note in practice, it is convenient to run the σ-basis algorithm on power series with precision slightly higher than $2B$ in order to have fewer vectors at this point (usually ℓ).

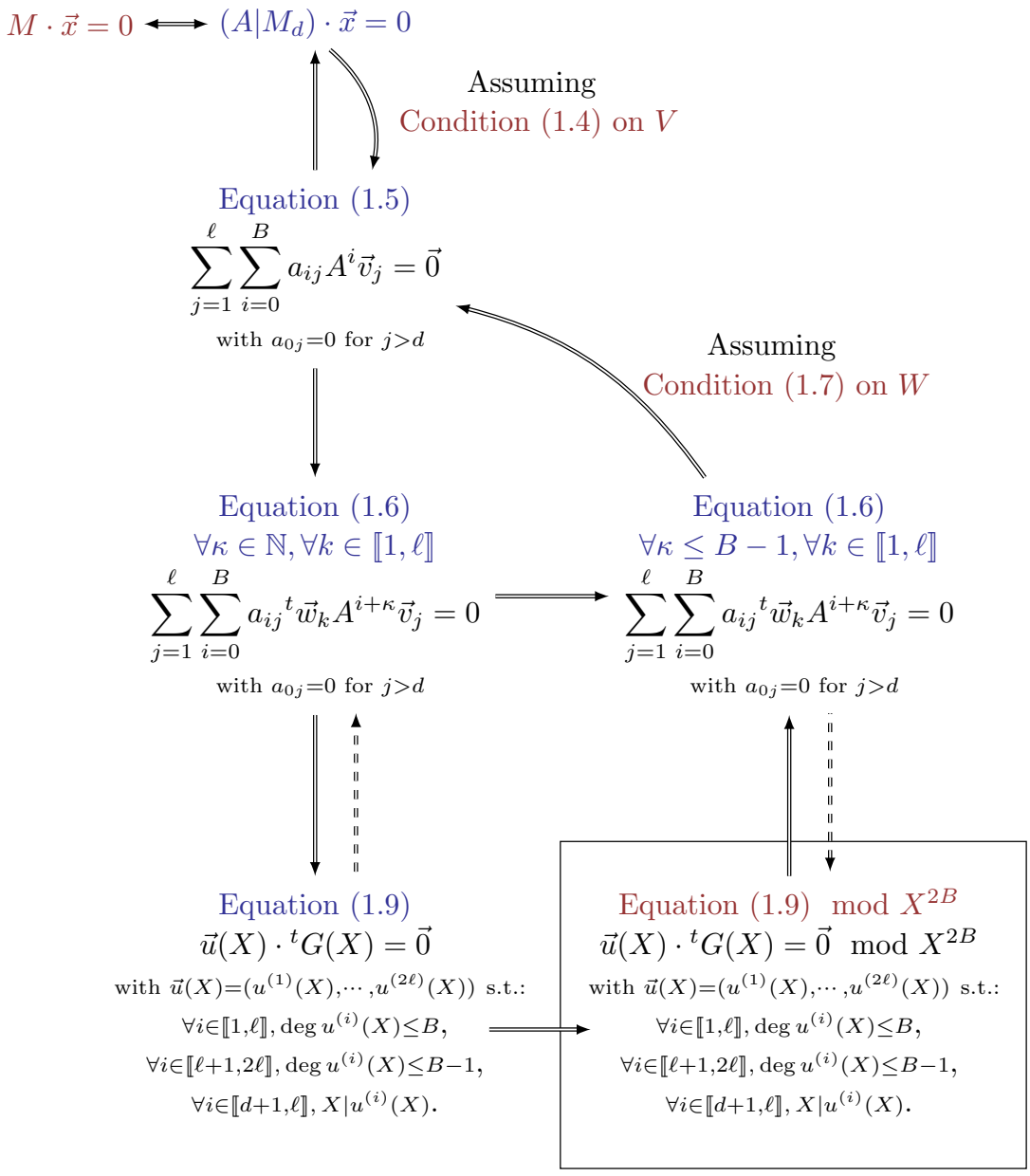


Fig. 1.3 How the computation of the kernel of a nearly sparse matrix M reduces to the computation of the kernel of a power series matrix G. Equivalences and implications between various problems have to be read as follows: $A \Rightarrow B$ means that a solution of Equation A can be transformed into a solution of Equation B. The unknowns are denoted by x, a_{ij} or $\vec{u}$ whereas all the other variables $M, d, M_d, A, \ell, B, V = \{\vec{v}_1, \cdots, \vec{v}_\ell\}, W = \{\vec{w}_1, \cdots, \vec{w}_\ell\}$ and G are assumed to be known. Note that, even if it is true, we don't prove Equation (1.9) $\Rightarrow$ Equation (1.6) here since the others implications are already sufficient to conclude. Same remark for Equation (1.6) with all $\kappa \leq B-1 \Rightarrow$ Equation (1.9) mod X^{2B}.

of all solution vectors of Equation (1.9) with degree at most B is generated by the family:

$$\mathcal{E} := \bigcup_{i=1}^{t} \{\vec{b_i}, X\vec{b_i}, X^2\vec{b_i}, \cdots, X^{B-\deg \vec{b}_i}\vec{b_i}\}.$$

Note that this family is free and thus a basis of the subspace of solutions of Equation (1.9). Indeed, the t vectors $\vec{b_i}$ belong to a σ-basis and, thus, are linearly independent. Moreover, multiplication by X induces a block diagonal structure on the matrix representing $\mathcal{E}$. Due to this structure, all vectors in $\mathcal{E}$ are also linearly independent.

To obtain a basis of the kernel of M, we now need a filtering step to ensure that we only keep the vectors of $\mathcal{E}$ with degree at most $B-1$ on the last ℓ coordinates and constant coefficients that are zeroes in positions $d+1$ to ℓ. Interestingly, the first property already holds for all vectors of $\mathcal{E}$ but the final multiple of each $\vec{b_i}$, i.e. $X^{B-\deg \vec{b}_i}\vec{b_i}$. Similarly, the second property already holds for all multiples of $X^j\vec{b_i}$ with $j \neq 0$. Thus, for each vector $\vec{b_i}$ all multiples except (possibly) the first and last already satisfy all extra condition. In addition, some linear combinations of these first and last multiples may satisfy the extra conditions and some may not. However, it is easy to construct the combinations that work. Indeed, the extra conditions are linear and only involve coefficients in $2\ell - d$ positions. Thus, to find these combinations, it suffices to extract the relevant coefficients from the polynomial multiples that do not already satisfy the condition and assemble them in a matrix of dimension $2\ell - d$ by at most[7] $2t$. The kernel of this matrix describes the desired combinations and it can be computed it in $O(\ell^\omega)$ operations asymptotically; in $O(\ell^3)$ operations in practice, especially for small values of ℓ. We let $\vec{b'_i}$ denote the t' combinations which are thus obtained.

We conclude that the basis of all solutions of Equation (1.9) that satisfy the three conditions is given by:

$$\mathcal{U} := \{\vec{b'_1}, \cdots, \vec{b'_{t'}}\} \cup \bigcup_{i=1}^{t} \{X\vec{b_i}, X^2\vec{b_i}, \cdots, X^{B-1-\deg \vec{b}_i}\vec{b_i}\}.$$

Note that this can be represented in a compact form, just by giving $t + t'$ vectors, with $t' \leq 2t$. This precisely gives a basis of the solutions of the equation highlighted by a frame in Figure 1.3.

[7]Indeed, vectors $\vec{b_i}$ with exact degree B appear once in the matrix while others appear twice.

Algorithm 1.3 Nearly sparse algorithm for $(A|M_d)\vec{x} = \vec{0}$

Input: A matrix A of size $N \times N$ and a matrix $M_d = (\vec{\delta}_1|\cdots|\vec{\delta}_d)$ of size $N \times d$
Output: A basis of $\mathrm{Ker}(A|M_d)$.

Compute a sequence of matrices

$\vec{r}_{d+1} \leftarrow\in \mathbb{K}^N, \cdots, \vec{r}_\ell \leftarrow\in \mathbb{K}^N$ and $\vec{w}_1 \leftarrow\in \mathbb{K}^N, \cdots, \vec{w}_\ell \leftarrow\in \mathbb{K}^N$
$\vec{v}_1 \leftarrow \vec{\delta}_1, \cdots, \vec{v}_d \leftarrow \vec{\delta}_d$
$\vec{v}_{d+1} \leftarrow \vec{r}_{d+1}, \cdots, \vec{v}_\ell \leftarrow \vec{r}_\ell$
$B \leftarrow \lceil N/\ell \rceil$
for any of the ℓ processors indexed by j **do**
  $u_0 \leftarrow v_j$
  for $i = 0, \cdots, 2B$ **do**
    for $k = 1, \cdots, \ell$ **do**
      $\lambda_{i,j,k} \leftarrow \vec{w}_k \cdot \vec{u}_i$
      $\vec{u}_{i+1} \leftarrow A\vec{u}_i$
    end for
  end for
end for
for $i = 0, \cdots, 2B$ **do**
  $M_i \leftarrow (\lambda_{i,j,k})$ the $\ell \times \ell$ matrix containing all the products of the form ${}^t\vec{w}A^i\vec{v}$
end for

Apply Giorgi, Jeannerod, Villard's algorithm

$S \leftarrow \sum_{i=0}^{2B-1} M_i X^i$.
Recover a $\sigma-$basis of the matrix ${}^t(S| - \mathrm{Id})$ modulo X^{2B}.
$\vec{b_1}, \cdots, \vec{b_t} \leftarrow$ the vectors in this $\sigma-$basis of degree lower than B.
$\vec{b'_1}, \cdots, \vec{b'_{t'}} \leftarrow$ a basis of the linear combinations of $\vec{b_1}, \cdots, \vec{b_t}, X^{B-\deg b_1}\vec{b_1}, \cdots, X^{B-\deg b_1}\vec{b_t}$ s.t. the ℓ last coordinates have degree lower than $B-1$ and coordinates between $d+1$ and ℓ are divisible by X.
$U \leftarrow [\vec{b'_1}, \cdots, \vec{b'_{t'}}, X\vec{b_1}, \cdots, X^{B-1-\deg b_1}\vec{b_1}, \cdots, X\vec{b_t}, \cdots, X^{B-1-\deg b_t}\vec{b_t}]$

Resolution

Sol $\leftarrow [\,]$
for $\vec{u} \in U$ **do**
  for $i = 0, \cdots, B$ **do**
    for $j = 1, \cdots, \ell$ **do**
      $a_{ij} \leftarrow$ the coefficient associated to the monomial X^{B-i} in the polynomial that is the j-th coefficient of $\vec{u}$.
    end for
  end for
  $\vec{x} \leftarrow {}^t({}^t(\sum_{j=1}^{\ell} \sum_{i=0}^{B-1} a_{(i+1)j} A^i \vec{v}_j)|a_{01}|\cdots|a_{0d})$
  Add $\vec{x}$ to Sol
end for
return Sol

Algorithm 1.3 sums up in pseudo-code the main steps that occur to compute the kernel of a nearly sparse matrix M that has been preconditioned into a matrix composed of a square matrix A concatenated with the dense part M_d of M.

1.4.4.1 *Checking condition* (1.4).

A benefit of this process is that it also checks the validity of (1.4). Indeed, looking back at the family $\mathcal{F}$ we see that is consists of ℓB vectors in $\mathbb{K}^N$. The matrix corresponding to $\mathcal{F}$ has full rank if and only if the dimension of its kernel is $\ell B - N$. Yet, an element of this kernel is exactly a family of coefficients (a_{ij}) such that $\sum_{j=1}^{\ell} \sum_{i=0}^{B-1} a_{ij} A^i v_j = 0$. Note that it differs from Equation (1.5) from the fact that the sum ends at $B-1$ and not B and nothing it said about the coefficients a_{0j}. Following the paths given in Figure 1.3 we can derive a bijection between the kernel of this matrix and the set of solutions of Equation (1.9) with degree $B-1$ on the first ℓ coordinates and $B-2$ on the last ℓ coordinates. Since we already have computed a larger set of solutions of Equation (1.9), we can check if the dimension of the restricted set is $\ell B - N$. If not, the elements of the kernel of M that are obtained are still valid, but the basis of the kernel may be incomplete.

1.4.5 *Requirements on the parameters*

At this point, we need to choose the values of the parameters B and ℓ depending on the input parameters N, s and d. By construction, we already know that $\ell \geq d$. However, for conditions (1.4) and (1.7) to be satisfiable, there are additional restrictions. In particular, condition (1.7) requires a family of ℓB vectors to have rank N, thus we need:

$$B \geq \left\lceil \frac{N}{\ell} \right\rceil .$$

There are other hidden implied requirements. Indeed, looking again at condition (1.7), we see that all vectors of $\left\{ {}^t\vec{w}_j A^i \middle|\, {}^{i\,=\,1,\,\cdots\,,\,B\,-\,1}_{j\,=\,1,\,\cdots\,,\,\ell} \right\}$ belong to the image of A. Thus, the dimension of the vector space in condition (1.7) is upper bounded by $\mathrm{Rank}(A) + \ell$. Moreover, due to the preconditioning of Section 1.4.2, we know that the rank of A is at most s. This implies that the algorithm requires:

$$\ell \geq \max(N - s, d).$$

Note that, over a large field $\mathbb{K}$, the dimension of the vector space in condition (1.7) for randomly chosen vectors $\vec{w}_i$ is $\mathrm{Rank}(A) + \ell$ with probability close to one.

The requirements associated to condition (1.4) do not give stronger arithmetic conditions on ℓ and B. However, the family of vectors in condition (1.4) also contains fixed vectors (derived from the dense columns of

M), thus we cannot claim that the condition hold for random choices of V. However, since our algorithm also checks the validity of Condition (1.4), this is a minor drawback.

1.4.6 *Complexity analysis*

The total cost of our method contains two parts. One part is the complexity of the matrix-by-vector products whose sequential cost is $O(\lambda N^2)$ including the preparation of the sequence of $\ell \times \ell$ matrices and the final computation of the kernel basis. It can easily be distributed on several processors, especially when the number of sequences ℓ is equal to the number of processors c or a multiple of it. This minimizes the wall clock time of the matrix-by-vector phases at $O(\lambda N^2/c)$. Moreover, since $B \approx N/\ell$, the phase that recovers the coefficients a_{ij} has complexity $\tilde{O}(\ell^{\omega-1}N)$ using Giorgi, Jeannerod and Villard algorithm. The filtering step after this algorithm costs $O(\ell^w)$ and can thus be neglected (since obviously $\ell \leq N$).

To minimize the cost of Giorgi, Jeannerod and Villard algorithm, we let ℓ be the smallest multiple of c larger than d. In that case, the total sequencial cost of the algorithm becomes:

$$O(\lambda N^2) + \tilde{O}(\max(c,d)^{\omega-1}N).$$

This has to be compared with the previous $O((\lambda + d)N^2) + \tilde{O}(c^{\omega-1}N)$ obtained when combining Block Wiedemann algorithm with Giorgi, Jeannerod and Villard variant to solve the same problem. Note that the wall clock time also decreases from $O((\lambda+d)N^2/c)+\tilde{O}(c^{\omega-1}N)$ to $O(\lambda N^2/c)+\tilde{O}(\max(c,d)^{\omega-1}N)$.

If $d \leq c$ then the complexity of the variant we propose is clearly in $O(\lambda N^2)+\tilde{O}(c^{\omega-1}N)$, which is exactly the complexity obtained when combining Block Wiedeman algorithm with Giorgi, Jeannerod and Villard variant to solve a linear algebra problem on a (truly) sparse matrix of the same size. In a nutshell, when parallelizing on c processors, **it is possible to tackle up to c dense columns for free.**

1.4.7 *How dense can nearly sparse matrices be ?*

We already know that our nearly sparse algorithm behaves better than the direct adaptation of sparse methods. However, when the number d of dense columns becomes much larger, it makes more sense to compare to the complexity of dense methods, *i.e.* to compare our complexity with $O(N^\omega)$. In this case, we expect the number of processors to be smaller

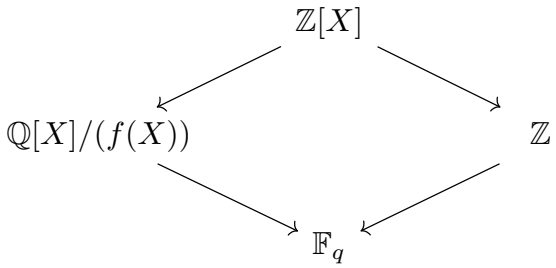
that the number of dense columns and thus replace $\max(c, d)$ by d in the complexity formulas.

Assume that $d \leq N^{1-\epsilon}$ with $\epsilon > 0$ then our complexity becomes $\tilde{O}(d^{\omega-1}N) = O(N^{\omega-\epsilon(\omega-1)}(\log N)^{\alpha})$ for some $\alpha > 0$, which is asymptotically lower than $O(N^{\omega})$. However, when the matrix is almost fully dense, i.e. for $d = \Omega(N)$, our technique becomes slower, by a logarithm factor, than the dense linear algebra methods.

1.5 Application to Discrete Logarithm Computations

In this section, we discuss the application of our adaptation of Block Wiedemann to discrete logarithm computations using the Number Field Sieve (NFS) algorithm [10, 15, 17], which applies to medium and high characteristic finite fields $\mathbb{F}_q$.

NFS contains several phases. First, a preparation phase constructs a commutative diagram of the form:



when using a rational side. Even if there also exists a generalization with a number field on each side of the diagram, for the sake of simplicity, we only sketch the description of the rational-side case.

The second phase builds multiplicative relations between the images in $\mathbb{F}q$ of products of ideals of small prime norms in the number field $\mathbb{Q}[X]/(f(X))$ and products of small primes. These relations are then transformed into linear relations between virtual logarithms of ideals and logarithms of primes modulo the multiplicative order of $\mathbb{F}_q^*$. Writing down these linear relations requires to get rid of a number of technical obstructions. In practice, this means that each relation is completed using a few extra unknowns in the linear system whose coefficients are computed from the so-called Schirokauer's maps [18]. Essentially, these maps represent the contribution of units from the number field in the equations. Due to the way they are computed, each of these maps introduces a dense column in

the linear system of equations. The total number of such columns is upper-bounded by the degree of f (or the sum of the degrees when there are two number fields in the diagram).

The third phase is simply the resolution of the above linear system. In a final phase which we do not describe, NFS computes individual logarithms of field elements. An optimal candidate to apply our adaptation of Coppersmith's Block Wiedemann algorithm precisely lies in this third sparse linear algebra phase. Indeed, the number of dense columns is small enough to be smaller than the number of processors that one would expect to use in such a computation. Typically, in [4], the degree of the number field was 5, whereas the number of maps (so the number of dense columns) was 4, and the number of processors 12. Asymptotically, we know that in the range of application of NFS, the degree of the polynomials defining the number fields are at most $O((\log q/\log\log q)^{2/3})$. This is negligible compared to the size of the linear system, which is about $L_q(1/3) = \exp(O((\log q)^{1/3}(\log\log q)^{2/3}))$.

Thus, our new adaptation of Coppersmith's Block Wiedemann algorithm completely removes the difficulty of taking care of the dense columns that appear in this context. It is worth noting that these dense columns were a real practical worry and that other, less efficient, approaches have been tried to lighten the associated cost. For instance, in [3], the construction of the commutative diagram was replaced by a sophisticated method based on automorphisms to reduce the number of maps required in the computation. In this extend, this approach is no longer useful.

Moreover, since we generally have some extra processors in practice, it is even possible to consider the columns corresponding to very small primes or to ideals of very small norms as part of the dense part of the matrix and further reduce the cost of the linear algebra.

Bibliography

[1] Leonard Adleman, *A subexponential algorithm for the discrete logarithm problem with applications to cryptography*. 20th Annual Symposium on Foundations of Computer Science, 1979.

[2] Elwyn R. Berlekamp, *Nonbinary BCH decoding (Abstr.)*. IEEE Transactions on Information Theory, 1968.

[3] Razvan Barbulescu, Pierrick Gaudry, Aurore Guillevic and François Morain, *Improvements to the number field sieve for non-prime finite fields*. INRIA Hal Archive, Report 01052449, 2014.

[4] Cyril Bouvier, Pierrick Gaudry, Laurent Imbert, Hamza Jeljeli and Emmanuel Thomé, *Discrete logarithms in GF(p) – 180 digits*. Announcement to the NMBRTHRY list, item 003161, June 2014.

[5] Bernhard Beckermann and George Labahn *A Uniform Approach for the Fast Computation of Matrix-Type Padé Approximants*. SIAM Journal on Matrix Analysis and Applications, 1994.

[6] Don Coppersmith, *Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm*. Mathematics of Computation, 1994.

[7] Don Coppersmith, Andrew Odlyzko and Richard Schroeppel, *Discrete Logarithms in GF(p)*. Algorithmica, 1986.

[8] Don Coppersmith, and Shmuel Winograd, *Matrix Multiplication via Arithmetic Progressions*. Journal of Symbolic Computation, 1990.

[9] Pascal Giorgi and Claude-Pierre Jeannerod and Gilles Villard *On the complexity of polynomial matrix computations*. Symbolic and Algebraic Computation, International Symposium ISSAC, 2003.

[10] Antoine Joux, Reynald Lercier, Nigel Smart and Frederik Vercauteren, *The number field sieve in the medium prime case*. Advances in Cryptology-CRYPTO 2006.

[11] Antoine Joux and Andrew Odlyzko and Cécile Pierrot, *The past, evolving present and future of discrete logarithm*. Open Problems in Mathematics and Computational Sciences, C. K. Koc, ed. Springer, 2014.

[12] Erich Kaltofen, *Analysis of Coppersmith's Block Wiedemann Algorithm for the Parallel Solution of Sparse Linear Systems*. Mathematics of Computation, 1995.

[13] Erich Kaltofen and David Saunders, *On Wiedemann's Method of Solving Sparse Linear Systems*. Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, New Orleans, USA, October 7-11, 1991.

[14] François Le Gall, *Powers of tensors and fast matrix multiplication*. International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014.

[15] Arjen K. Lenstra and Hendrik W. Lenstra, Jr., *The development of the number field sieve*. Springer-Verlag, Lecture Notes in Mathematics, 1993.

[16] James L. Massey, *Shift-register synthesis and BCH decoding*. IEEE Transactions on Information Theory. 1969.

[17] Cécile Pierrot, *The Multiple Number Field Sieve with Conjugation and Generalized Joux-Lercier Methods*. Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015.

[18] Oliver Schirokauer, *Discrete logarithm and local units*. Philosophical Transactions of the Royal Society of London, 1993.

[19] Emmanuel Thomé, *Subquadratic Computation of Vector Generating Polynomials and Improvement of the Block Wiedemann Algorithm*. J. Symb. Comput., 2002.

[20] Douglas H. Wiedemann *Solving sparse linear equations over finite fields*. IEEE Transactions on Information Theory, 1986.