J. Symbolic Computation (2002) **33**, 757–775
doi:10.1006/jsco.2002.0533
Available online at http://www.idealibrary.com on IDEAL®

Subquadratic Computation of Vector Generating Polynomials and Improvement of the Block Wiedemann Algorithm

EMMANUEL THOMÉ

LIX (UMR CNRS 7650), École polytechnique, 91128 Palaiseau Cedex, France

This paper describes a new algorithm for computing linear generators (vector generating polynomials) for matrix sequences, running in subquadratic time. This algorithm applies in particular to the sequential stage of Coppersmith's block Wiedemann algorithm. Experiments showed that our method can be substituted in place of the quadratic one proposed by Coppersmith, yielding important speedups even for realistic matrix sizes. The base fields we were interested in were finite fields of large characteristic. As an example, we have been able to compute a linear generator for a sequence of 4×4 matrices of length 242 304 defined over $\mathbb{F}_{2^{607}-1}$ in less than 2 days on one 667 MHz alpha ev67 CPU.

1. Introduction

Although it can be stated in a rather general context, we will here envision the problem of finding a linear generator for a matrix sequence in the light of how it applies to the block Wiedemann algorithm, described in Coppersmith (1994). This algorithm addresses the problem of finding one or several solutions to a large sparse linear system defined over a finite field, or in other words, solutions w to the equation $Bw = 0$, where B is a singular $N \times N$ matrix defined over the field $K = \mathbb{F}_q$, q being a prime power, and B is sparse: it has only few non-zero coefficients per row. The block Wiedemann algorithm takes advantage of this last fact (the fewer non-zero coefficients B has, the faster the computations). Many other "sparse" linear algebra algorithms exist (Wiedemann, 1986; LaMacchia and Odlyzko, 1990; Coppersmith, 1993; Montgomery, 1995). This is in contrast to more general-purpose procedures, like the well-known Gaussian elimination, which does not consider nor preserve the sparsity of the input matrix.

Sparse linear systems over finite fields occur in a variety of contexts, more specifically in computational algebraic number theory. We originally encountered the problem in the course of solving discrete logarithm problems over $\mathbb{F}_{2^n}$ with the *index-calculus* algorithm of Coppersmith (1984). This computation is described in Thomé (2001b, 2002). Generally, any *index-calculus*-type algorithm for computing discrete logarithms in an appropriate group calls for the solution of a sub-problem of this kind: see Odlyzko (1985) and for instance Gaudry (2000a,b). Huge sparse linear systems defined over the binary field $\mathbb{F}_2$ also occurred in the course of the recent record-breaking factorizations of composite numbers with the number field sieve (Cavallar *et al.*, 2000; CABAL, 2000).

0747–7171/02/050757 + 19 $35.00/0

Coppersmith's block Wiedemann algorithm is a clever generalization of an older algorithm proposed in Wiedemann (1986). In the latter algorithm, one is interested at some point in finding a linear generator for a given scalar sequence. The Berlekamp–Massey or the extended Euclidean algorithms can do this in quadratic time. Subquadratic alternatives exist, which can take advantage of fast polynomial multiplication algorithms. These are the HGCD (*half-gcd*) algorithm from Aho *et al.* (1974) and the PRSDC (*polynomial remainder sequences by divide-and-conquer*) algorithm from Gustavson and Yun (1979). Coppersmith (1994) introduces a multi-dimensional variant of Wiedemann's algorithm, whose main advantage is that it allows partial distribution and/or parallelization of part of the computations. In this algorithm, the linear generator finding task is transformed into a multi-dimensional analogue (defined precisely in Section 2), which Coppersmith solves by a "matrix Berlekamp–Massey".

The work in this paper provides a subquadratic variant of Coppersmith's "matrix Berlekamp–Massey". The complexity reduction is obtained by the use of the fast Fourier transform (FFT) method. Our method is recursive, as for the HGCD or PRSDC algorithms from which it was actually adapted. Other subquadratic algorithms exist for this task (Beckerman and Labahn, 1994), also using FFT. We will discuss more deeply the respective complexities and the differences between our algorithm and Beckermann and Labahn's in Section 2.2, once the required concepts have been defined.

An earlier version of this work appeared in Thomé (2001a). This paper completes the results presented at ISSAC'2001 by providing a better theoretical setting and improving the presentation of the algorithm. We have also now implemented our algorithm with success, and provide running times that could be employed to draw a comparison with Beckermann and Labahn's method.

The organization of this paper is as follows. Sections 2–4 concentrate on the task of computing a linear generator for a matrix sequence. Section 2 defines this central concept of the generator in 2.1, explains which quantities are computed by our algorithm and by Beckermann and Labahn's in 2.2. Section 3 presents the framework and requirements that are shared by Coppersmith's algorithm for finding linear generators and ours. Our new algorithm is presented in Section 4. Sections 5–7 concentrate on the influence of our new algorithm on the block Wiedemann algorithm. Section 5 introduces the block Wiedemann algorithm, and its connections to the presentation that we make of the linear generator finding problem. In Section 6, we discuss the overall cost of the block Wiedemann algorithm, along with the optimal value of its parameters. Section 7 discusses practical concerns about the implementation of our approach inside a more extended computation like the discrete logarithm computation in Thomé (2001b, 2002). Section 8 shows the results of our experiments with the new algorithm.

2. Linear Generators for Matrix Sequences

2.1. DEFINITIONS

Throughout this paper, K denotes a finite field, and m and n are two chosen integers. We make no hypotheses on the characteristic of K, nor on m and n being greater than or equal to the other. We focus on sequences of $m \times n$ matrices, which are represented as matrices of formal power series, that is, elements of the structure $K[[X]]^{m\times n}$. Similarly, we also introduce several connected structures, like the $n \times r$ matrices with polynomial entries, denoted $K[X]^{n\times r}$, or the n-dimensional column vectors with polynomial entries,

denoted $K[X]^{n\times 1}$. When dealing with these structures, one must keep in mind the equivalence with the point of view of polynomials with matrix coefficients. Therefore, when we make use of the "degree" of a matrix of polynomials, it is actually the maximal degree of its entries. We define the concept of linear generator for series in $K[[X]]^{m\times n}$.

Definition. Let $r \in \mathbb{N}^*$. A sequence $A(X) \in K[[X]]^{m\times n}$ admits $F(X) \in K[X]^{n\times r}$ (non-zero) as a right-hand linear generator if $A(X)F(X) \in K[X]^{m\times r}$.

The polynomial $F(X)$ is also often referred to as a matrix generating polynomial for $A(X)$. For $r = 1$, $F(X)$ is called a vector generating polynomial. This is actually the central object we will concentrate on. Also, if $r = n$, and if the square matrix $F(X)$ is invertible in $K[[X]]^{n\times n}$ (we also say *unimodular*), then we can write down $A(X)$ in the right rational form $G(X)F^{-1}(X)$. It is understood that all the concepts above can be translated to the left-hand situation.

For a pair $(F(X), G(X)) \in K[X]^{n\times r} \times K[X]^{m\times r}$ we also introduce the notation: $\delta(F, G) := \max(\deg F, 1 + \deg G)$. Thus one can say that $F(X)$ is a generator if and only if $\delta(F, AF)$ is finite.

2.2. existing algorithms for computing generators

Generally stated, our problem is the computation of a right-hand $n \times r$ matrix generating polynomial. In some cases, only $r = 1$ will be required. Several algorithms have been introduced to deal with the computation of matrix generating polynomials. Kaltofen (1995) provides a short survey on these, the earliest work cited being Rissanen (1972). Coppersmith suggested for this purpose an algorithm which addresses the case $r = 1$ with complexity $O((m+n)n^2k^2)$, where k is the degree of the computed generator. Coppersmith's algorithm relies on some non-degeneracy assumptions concerning the input series $A(X)$ that are summarized in 3.1. One can reasonably expect these requirements to be satisfied for non-particular input. For the same input series $A(X)$, the "power Hermite–Padé solver" of Beckerman and Labahn (1994) assumes that FFT is available over the base field, and computes $n \times r$ matrix generating polynomials for any r in time $O((m+n)^2mk\log^2 k)$, again with k the degree of the generator. The algorithm of Beckermann and Labahn makes no regularity assumptions on the input sequence $A(X)$. Other algorithms are cited in Kaltofen (1995) and Villard (1997a), notably a method due to Bitmead and Anderson (1980) and Morf (1980). It appears that even if Coppersmith's algorithm is not supposed to be the asymptotically fastest of all these algorithms, it has been the preferred one for experiments with the block Wiedemann algorithm like Lobo (1995), Kaltofen and Lobo (1999) and Penninga (1998). Several reasons might explain this. Comparatively, Coppersmith's algorithm is pretty simple. Also, the absence of need for randomization might be considered as an advantage, compared to the Bitmead/Anderson/Morf method. Another good point for Coppersmith's algorithm is that there is no big hidden constant in the complexity: to be exact, $\frac{m+n}{2}n^2k^2$ scalar multiplications are required. Last but not least, asymptotically fast algorithms using recursion and the FFT, like the algorithms we are concerned about, are usually not worthwhile below a certain threshold. This threshold might still be above the current computation sizes, thus making quadratic approaches preferable.

The algorithm that we present in this paper is a subquadratic version of Coppersmith's algorithm. It relies on the same assumptions, and also computes a vector generating

polynomial. Using a generic fast polynomial multiplication algorithm, which requires $\mathsf{M}(d)$ operations to multiply two polynomials of degree d with coefficients in K, the complexity of our algorithm is $O((m+n)^3\mathsf{M}(k)\log k)$. If FFT is available over the base field, this complexity reduces to $O((m+n)^2k(m+n+\log k)\log k)$. If m and n are in $O(\log k)$, this expresses as $c(\phi)\frac{(m+n)^3}{m}k\log^2 k + (3+\phi)(m+n)^3k\log k$, where $c(\phi)$ is in the range $[0.5, 5]$ and depends only on $\phi = \frac{m}{m+n}$ ($c(\phi)$ is very close to 5ϕ). As in the rest of the paper, log denotes here the logarithm in base 2.

We add two remarks concerning these algorithms. First, while Coppersmith's algorithm and ours are focused on the situation $r = 1$, it is not hard to see that for generic input, a solution for $r = n$ is produced simultaneously, yielding a rational form for the series (but for this to be ensured, we would need additional requirements). Second, Beckermann and Labahn's as well as our algorithm have the complexities announced when FFT is available over the base field. If this is not the case, but the base field is a prime field, then we can work around the absence of native FFT by padding the data into integers, and using integer FFT, at the mere cost of an additional $\log\log k$ factor in the complexity.

3. A Matrix Version of the Berlekamp–Massey Algorithm

3.1. INPUT REQUIREMENTS

The framework that we will present for computing a vector generating polynomial makes assumptions that are summarized hereafter. First, we will make use of the following theorem.

THEOREM 3.1. *Let $A(X) \in K[[X]]^{m\times n}$. Suppose that $A(X)$ has a left rational form $D(X)^{-1}N(X)$. If we have matrices $F(X) \in K[X]^{n\times r}$, $G(X) \in K[X]^{m\times r}$, $E(X) \in K[[X]]^{m\times r}$, and an integer t such that:*

$$A(X)F(X) = G(X) + X^tE(X),$$
$$\textit{Then:} \qquad t - \delta(F,G) \geq \delta(D,N) \Rightarrow E(X) = 0.$$

The verification of this assertion is an easy matter (checking degrees suffices).

In order to be able to use Theorem 3.1, we assume that $A(X)$ has a left rational form $D(X)^{-1}N(X)$. We denote by d the integer $\delta(D,N)$. We will not need to compute $N(X)$ and $D(X)$, but we assume that they exist. Furthermore, we introduce an integer s which is the least integer such that the subspace of $K^{m\times 1}$ spanned by the columns of the first s coefficients of $A(X)$ (as a matrix polynomial) has maximal dimension m. If this is impossible (because even with $s \to \infty$ the dimension is less than m), then by a change of basis we can drop the superfluous lines in $A(X)$, then have m smaller, and therefore assume that s exists without loss of generality.

Depending on the inputs above (the two integers s and d), Coppersmith's matrix generalization of the Berlekamp–Massey algorithm, as well as our accelerated alternative, provide a constructive proof to the following assertion:

THEOREM 3.2. *A right-hand vector generating polynomial $u(X)$ for $A(X)$ can be deterministically computed using only the first $L = s + \left\lceil \frac{m+n}{n}d \right\rceil$ coefficients of $A(X)$. The computed generator $u(X)$ satisfies $\delta(u, Au) \leq s + \left\lceil \frac{m}{n}d \right\rceil$.*

The integer $L = s + \left\lceil \frac{m+n}{n} d \right\rceil$ introduced in the last proposition will remain fixed throughout the paper.

3.2. FRAMEWORK

The strategy that we use in order to produce a vector generating polynomial involves writing equations like in the hypothesis of Theorem 3.1 for several (as many as $m+n$) candidate vectors f_j, and vectors g_j which are approximations of Af_j. With the quantity δ playing a crucial role in Theorem 3.1, we will also maintain a bound δ_j on $\delta(f_j, g_j)$,

$$\forall j \in [\![1, m+n]\!], \qquad \begin{aligned} A(X)f_j(X) &= g_j(X) + X^t e_j(X), \\ \delta(f_j, g_j) &\leq \delta_j. \end{aligned} \tag{C1}$$

The f_j's, g_j's, and e_j's are gathered to form the $m+n$ columns of the three matrices $f(X)$, $g(X)$, and $e(X)$ (with, respectively, n, m, and m lines). Another condition will be enforced ($[X^k]P$ denotes the coefficient of degree k in P):

$$\operatorname{rank}([X^0]e) = m. \tag{C2}$$

Since Theorem 3.1 states that f_j is a generator if the gap between t and $\delta(f_j, g_j)$ is big enough, we will try to infer, from the equations (C1) and (C2) above, the same equations with t increased by one, and the δ_j's increased by less than one on average, so that eventually the gap will be big enough for some j. We will explain how the original setting is obtained in 3.3. As for how we go from t to $t+1$, this is exactly addressed by Coppersmith (1984), and we have also detailed this step in Thomé (2001a). We will not repeat this verbosely here, but rather refer to this procedure as a black-box algorithm named ALGO_1, which achieves the following:

THEOREM 3.3. *Assuming conditions* (C1) *and* (C2) *hold at step t, there is an algorithm* ALGO_1 *that, knowing $[X^0]e^{(t)}$ and $(\delta_1^{(t)}, \ldots, \delta_{m+n}^{(t)})$, computes a $(m+n)\times(m+n)$ matrix $P^{(t)}$ along with integers $(\delta_1^{(t+1)}, \ldots, \delta_{m+n}^{(t+1)})$ such that:*

$$f^{(t+1)} = f^{(t)}P^{(t)}, \qquad g^{(t+1)} = g^{(t)}P^{(t)}, \qquad e^{(t+1)} = e^{(t)}P^{(t)}\frac{1}{X},$$

and the $\delta_j^{(t+1)}$'s satisfy conditions (C1) *and* (C2) *at step $t+1$. Furthermore, we have $\sum_j \delta_j^{(t+1)} - \sum_j \delta_j^{(t)} = m$.*

3.3. INITIALIZATION

The initialization of the iterative process is somewhat involved, but can be done deterministically. Let us recall that we have assumed that the columns of the matrices $a_0, \ldots, a_{s-1}$ span the full vector space $K^{m\times 1}$. Hence we can find m vectors $r_1, \ldots, r_m$, all belonging to the canonical basis of $K^{n\times 1}$, along with integers $i_1, \ldots, i_m$ in the range $[\![0, s-1]\!]$, satisfying the property that the vectors $a_{i_k}r_k$, for $k \in [\![1, m]\!]$, form a basis of $K^{m\times 1}$. Given this data, we can provide initialization data for the algorithm, beginning at $t_0 = s$. We set the first n columns of $f^{(t_0)}(X)$ to be the identity matrix I_n. The remaining m columns will be the $X^{s-i_k}r_k$'s. All the δ_j's are initially set to $t_0 = s$. We can see trivially that condition (C1) is satisfied. Condition (C2) is a consequence of the

choice of the i_k's and r_k's. Let us denote by $\beta(X)$ the last m columns of the matrix $e^{(t_0)}(X)$ ($\beta(X)$ is an $m \times m$ matrix). By the choice of the i_k's and r_k's, the columns of $\beta(0)$ form a basis of $K^{m\times 1}$, hence $\det \beta(0) \neq 0$, and $\operatorname{rank}(e^{(t_0)}(0)) = m$. Furthermore, let us define an $(m+n) \times (m+n)$ matrix $h(X)$, which is the vertical concatenation of $f(X)$ and $e(X)$. The initial matrix $h^{(t_0)}$ has the following shape:

$$h^{(t_0)} = \left(\begin{array}{c|ccc} I_n & X^{s-i_1}r_1 & \cdots & X^{s-i_m}r_m \\ \hline \vdots & & \beta(X) & \end{array}\right).$$

Since $i_k < s$ for all k, the upper right part of $h^{(t_0)}(0)$ is zero. Therefore, we have $\det h^{(t_0)}(0) = \det \beta(0) \neq 0$. Thus, $h^{(t_0)}(X)$ is unimodular. This fact will be important to prove that the algorithm produces non-trivial output.

3.4. termination

Coppersmith's algorithm for finding linear generators consists of simply iterating ALGO_1 until a generator is produced. The data which needs to be kept along with this computation is the polynomial matrix $f(X)$, and the scalar matrix $[X^0]e(X)$. Since it is obvious from Theorem 3.3 that the *average* value $\overline{\delta}$ of the δ_j's increases by $\frac{m}{m+n}$ each time t increases by 1, we can express the average gap:

$$t - \overline{\delta} = t - \left(t_0 + (t-t_0)\frac{m}{m+n}\right) = (t-t_0)\frac{n}{m+n}.$$

For $t = t_0 + \left\lceil \frac{m+n}{n} d \right\rceil$, we have:

$$t - \overline{\delta} \geq d, \qquad \text{and} \qquad \overline{\delta} = s + \frac{m}{m+n}\left\lceil \frac{m+n}{n} d \right\rceil \leq s + \left\lceil \frac{m}{n} d \right\rceil.$$

Therefore there exists at least one j such that f_j is a vector generating polynomial, with the properties announced in 3.2. Note that f_j cannot be zero because otherwise we would have a zero column in $h(X)$, contradicting the fact that $h(X)$ is unimodular.

4. An Accelerated Version of Coppersmith's Algorithm

4.1. balancing polynomial multiplications

In Coppersmith's algorithm, the quadratic cost comes from the evaluation of $[X^t](Af)$ and the multiplication of $f(X)$ by a degree 1 matrix at each step t, for $t_0 \leq t \leq L$. Our divide-and-conquer approach aims at replacing these numerous very unbalanced computations by a few big polynomial multiplications, in order to take advantage of fast multiplication algorithms, like the FFT. In order to do this, we make extensive use of Theorem 3.3. Specifically, the fact that the only knowledge of $[X^t](Af)$—that is, $[X^0]e$—is necessary will prove to be crucial. In fact, knowing the first k coefficients of $e^{(t)}(X)$, is enough to compute $P^{(t)}$ up to $P^{(t+k-1)}$, without updating $f(X)$. Let us formalize these considerations.

Definition. A *k-context* is a pair of the form $E = (e(X), \Delta)$ corresponding to some iteration step of the iterative algorithm outlined in Section 3.2 where the $(m+n)$-tuple $\Delta = (\delta_j)_{j\in[\![1,m+n]\!]}$ and the first k coefficients of $e(X)$ are known.

DEFINITION. Generalizing the matrix $P^{(t)}$ introduced in Theorem 3.3, if E is a context corresponding to iteration step t of the algorithm in 3.2, and a, b are integers such that $0 \le a \le b$, we call $\pi_E^{(a,b)}$ the $(m+n) \times (m+n)$ matrix:

$$\pi_E^{(a,b)} = P^{(t+a)} \cdots P^{(t+b-1)}, \qquad \text{and} \qquad \pi_E^{(a,b)} = \mathrm{id} \qquad \text{if } a = b,$$

where the $P^{(t+i)}$ are the matrices computed as described by Theorem 3.3 at the corresponding iteration steps after t. This definition is justified by:

THEOREM 4.1. *A given k-context E determines completely any $\pi_E^{(a,b)}$ as long as $0 \le a \le b \le k$. If E corresponds to iteration step t of the algorithm, say $E = E^{(t)}$, then a $(k-b)$-context $E^{(t+b)}$ follows from the computation of $\pi_E^{(0,b)}$.*

PROOF. The proof is easy by induction. Theorem 3.3 states that $E^{(t)}$ determines $P^{(t)}$. $e^{(t+1)}$ follows since $e^{(t+1)} = e^{(t)} P^{(t)} \frac{1}{X}$. $\Delta^{(t+1)}$ follows as well since:

$$\delta_j^{(t+1)} = \max_i \{\delta_i^{(t)} + \deg P_{i,j}^{(t)}\}.$$

By an abuse of notation, we denote the latter $\Delta^{(t)} P^{(t)}$. Together, $e^{(t)} P^{(t)} \frac{1}{X}$ and $\Delta^{(t)} P^{(t)}$ form a $(k-1)$-context. Generalization of this step from t to $t+1$ to the result of the theorem is trivial.□

With this formalism, it becomes clear that our main point of interest is the quantity $\pi_{E^{(t_0)}}^{(0,L-t_0)}$ where $E^{(t_0)} = (e^{(t_0)}, \Delta^{(t_0)})$ is the initial L—t_0-context. Once $\pi_{E^{(t_0)}}^{(0,L-t_0)}$ is known, then all the columns of $f^{(t_0)}(X)\pi_{E^{(t_0)}}^{(0,L-t_0)}$ satisfy the conditions (C1) and (C2) from 3.2 with $t = L$, and since we know the δ_j's, we can pick a column that suits the requirements of Theorem 3.1. In fact, Coppersmith's algorithm described in the previous section does nothing more than that. It computes $\pi_{E^{(t_0)}}^{(0,L-t_0)}$ from $E^{(t_0)}$, and can be trivially generalized to compute $\pi_E^{(0,b)}$ from a given b-context E, in quadratic time.

From Theorem 4.1, we design an algorithm whose task is the computation of $\pi_E^{(0,b)}$ from a given b-context E. It is described in Program 4.1. In that piece of pseudo-code, ALGO_1 is the algorithm introduced in Theorem 3.3. Cutting at $\frac{b}{2}$ is legitimate because of Theorem 4.1. The recursive algorithm is named MSLGDC, by lack of imagination, from "matrix sequences linear generator by divide-and-conquer". It will be applied to the $(L-t_0)$-context $E^{(t_0)} = (e^{(t_0)}, \Delta^{(t_0)})$. As often with recursive algorithms, there exists a certain threshold under which the quadratic counterpart is more efficient. For the case of algorithm MSLGDC, taking this into account is easy: we replace the invocation of ALGO_1, on the second line, by an invocation of ALGO_k as soon as $b \le \mathsf{k}$, where we denote by ALGO_k any algorithm capable of computing $\pi_E^{(0,\mathsf{k})}$ from a k-context. For instance, Coppersmith's original algorithm can play this role.

4.2. COMPLEXITY OF MSLGDC

Since we are interested in large base fields, we will only count scalar multiplications. Our algorithm requires two non-trivial (more than linear) operations at each recursion level. These are:

$$e_R \leftarrow (e\pi_L \bmod X^b) \operatorname{div} X^{\lfloor \frac{b}{2} \rfloor}, \text{ and } \pi \leftarrow \pi_L \pi_R.$$

```
Algorithm MSLGDC
INPUT: A b-context E = (e, Δ).
OUTPUT: π_E^(0,b).
{
    if {b==0} return I_{m+n};
    if {b==1} return ALGO_1(e, Δ);
    (e_L, Δ_L)=((e mod X^⌊b/2⌋), elta);                    /* A ⌊b/2⌋-context */
    π_L=MSLGDC(e_L, Δ_L);
    (e_R, Δ_R)=(((eπ_L mod X^b) div X^⌊b/2⌋), Δπ_L);       /* A ⌈b/2⌉-context */
    π_R=MSLGDC(e_R, Δ_R);

    π=π_L × π_R;
    return π;
}
```

Program 4.1. Recursive algorithm for computing $\pi_E^{(0,b)}$

Of these polynomials, e has degree b, and π_L and π_R have degree $\frac{m}{m+n}\frac{b}{2}$. Using a generic fast multiplication algorithm requiring $\mathsf{M}(k)$ operations to multiply two polynomials of degree k, these operations would cost $m(m+n)^2\mathsf{M}(b)$ and $(m+n)^3\mathsf{M}\big(\frac{m}{m+n}\frac{b}{2}\big)$, that is, at most $\frac{3}{2}m(m+n)^2\mathsf{M}(b)$. Now, if we use the FFT, we can do much better. The expression of the complexity involves the cost M_1 of a multiplication in K, and the ratio $\phi = \frac{m}{m+n}$. The function $c(\phi)$ will appear in the proof.

THEOREM 4.2. *If K supports FFT (see von zur Gathen and Gerhard* 1999. *Chapter* 8*), the two operations above can be achieved in time $c(\phi)\mathsf{M}_1(m+n)^2 b\log b + (3+\phi)\mathsf{M}_1 m(m+n)^2 b + O((m+n)^2 b)$. This yields a complexity bound for algorithm* MSLGDC *with a b-context of $c(\phi)\mathsf{M}_1(m+n)^2 b\log^2 b + (3+\phi)\mathsf{M}_1 m(m+n)^2 b\log b + O((m+n)^2 b\log b))$.*

PROOF. What this theorem says is that the generic result is not only specified using the complexity of the FFT for $\mathsf{M}(k)$, but that we also improve the complexity with respect to m and n. Let us show how this is obtained.

We refer to von zur Gathen and Gerhard (1999, Chapter 8) for an introduction to the FFT. In a few words, the FFT relies, on the one hand, on the ability to efficiently compute the evaluation discrete Fourier transform (DFT) of a polynomial at a bunch of points—the 2^lth roots of unity for some l—and, on the other hand, on the ability to interpolate equally fast a polynomial given its values at those same points (inverse DFT, or IDFT, operation). The DFTs of two polynomials can be multiplied pointwise (at a linear cost in the number of points) to obtain the DFT of the product polynomial. The latter can then be recovered by an IDFT operation.

We aim here at multiplying polynomial matrices. As above, we compute the DFT of each entry in the matrices involved (e, π_L, and π_R), forming the matrix DFTs $\widehat{e}$, $\widehat{\pi_L}$, and $\widehat{\pi_R}$ (these are matrices of scalar sequences, or also sequences of scalar matrices). These

DFTs can be multiplied pointwise, involving one scalar matrix multiplication per point, to obtain the DFTs of the products: $\widehat{e_R}$, and $\widehat{\pi}$.

The number of points at which the DFTs are computed is actually driven by the number k of unknown coefficients in the product: we take the smallest power of 2 above k. Therefore at most $2k$ points are needed. We know that $\deg e = b$, $\deg e_L = \deg e_R = \frac{b}{2}$, $\deg \pi_L = \deg \pi_R = \frac{m}{m+n}\frac{b}{2} = \phi\frac{b}{2}$, and $\deg \pi = \phi b$. Hence we need transforms using $2*\left(\frac{b}{2}+\phi\frac{b}{2}\right) = (1+\phi)b$ points at most for the computation $e_R \leftarrow (e\pi_L \bmod X^b) \operatorname{div} X^{\lfloor \frac{b}{2} \rfloor}$, and transforms using $2\phi b$ points at most for the computation of $\pi \leftarrow \pi_L\pi_R$. The cost of the computation of a DFT or IDFT using k points is below $\frac{k}{2}\log k$ multiplications in K (von zur Gathen and Gerhard, 1999, Theorem 8.15). Resulting upper bounds on the time required to compute all the transforms are summarized hereafter. Of course, the transform of π_L need not be computed twice, so we keep the largest figure ($(1+\phi)b$ points needed).

$$\begin{array}{lll}
\text{DFT}: & e \to \widehat{e} & \frac{1}{2}\mathsf{M}_1 m(m+n)(1+\phi)b\log((1+\phi)b),\\
\text{DFT}: & \pi_L \to \widehat{\pi_L} & \frac{1}{2}\mathsf{M}_1 (m+n)^2(1+\phi)b\log((1+\phi)b),\\
\text{IDFT}: & \widehat{e_R} \to e_R & \frac{1}{2}\mathsf{M}_1 m(m+n)(1+\phi)b\log((1+\phi)b),\\
\text{DFT}: & \pi_R \to \widehat{\pi_R} & \mathsf{M}_1 (m+n)^2\phi b\log(2\phi b),\\
\text{IDFT}: & \widehat{\pi} \to \pi & \mathsf{M}_1 (m+n)^2\phi b\log(2\phi b).
\end{array}$$

Additionally, the matrix products involved by the pointwise multiplication of the DFTs yield a complexity of $m(m+n)^2(1+\phi)b\mathsf{M}_1 + 2(m+n)^3\phi b\mathsf{M}_1$ operations. The cost equation for the algorithm MSLGDC for order b follows by summation:

$$\begin{aligned}
C(b) = 2C\left(\frac{b}{2}\right) &+ c(\phi)\mathsf{M}_1(m+n)^2 b\log b\\
&+ (3+\phi)\mathsf{M}_1 m(m+n)^2 b + O((m+n)^2 b),
\end{aligned}$$

hence $C(b) \le c(\phi)\mathsf{M}_1(m+n)^2 b\log^2 b + (3+\phi)\mathsf{M}_1 m(m+n)^2 b\log b + O((m+n)^2 b\log b)$, as claimed, where we have introduced as $c(\phi)$ the quantity $\phi^2 + 3.5\phi + 0.5$. For $\phi = 0.5$, which is a typical setting, $c(\phi)$ is 2.5. □

The complexity above is expressed with respect to the number of terms of the sequence that are used. For our interest, this number is $L = s + \left\lceil \frac{m+n}{n} d \right\rceil$. The generator obtained has degree $k = s + \frac{m}{n}d$. We will see in the next section that s is small and can safely be ignored. If we want to express the complexity required to compute a generator of degree k with respect to k, we obtain:

$$C\left(\frac{m+n}{n}\frac{n}{m}k\right) = c(\phi)\mathsf{M}_1\frac{(m+n)^3}{m}k\log^2 k + O((m+n)^3 k\log k).$$

So the actual[†] speedup obtained when we compare with Coppersmith's version is $\frac{n^2 k}{10(m+n)\log^2 k}$, as long as m and n stay relatively small (here we have simplified to $m = n$).

5. Block Wiedemann Algorithm

We will now see how our approach of the computation of linear generators plugs well into the block Wiedemann algorithm. In this algorithm, we want to solve the equation $Bw = 0$ for a $N \times N$ matrix B defined over a finite field K.

[†]However, so many parameters are involved that this estimate is not really sharp.

5.1. principle of the block Wiedemann algorithm

In the original (non-block) algorithm from Wiedemann (1986), we focus on:

$$a_k = x^{\mathrm{T}} B^k y, \qquad k \geq 0,$$

where x and y are fixed elements of the vector space K^N acting as random inputs. A linear generator for this sequence is desired, and can be computed using only the first $2N$ coefficients a_k. If B has γ non-zero coefficients per line on average, those can be computed using $O(\gamma N^2)$ scalar multiplications in K. This computation is faster if γ is small, that is, if B is sparse. In addition, this evaluation is sequential by nature[†] since it involves repeated applications of B. Doing this computation in a parallel or distributed setting is infeasible without an important amount of communication between the different processors or machines taking part in the computation (it might be all right for an SMP crossbar, but it certainly is not for a network). Once a linear generator is obtained, one derives a solution to the equation $Bw = 0$.

Coppersmith (1994) brought the following interesting possibility: instead of vectors x and y, use *blocks* of vectors, of size $N \times m$ and $N \times n$, respectively, where m and n are chosen integers. We will concentrate on:

$$A(X) = \sum_{k=0}^{\infty} a_k X^k \in K[[X]]^{m \times n} \qquad \text{where } a_k = x^{\mathrm{T}} B^k y.$$

One "sample" $x^{\mathrm{T}} B^k y$ therefore contains more information because it is made up of several scalars. We will compute a vector generating polynomial for $A(X)$. For $m = n = 1$, this is the same computation as in the original Wiedemann algorithm. We will see that for all m and n, this generator yields a solution to our original linear system $Bw = 0$, and that it can be computed with the knowledge of approximately $\frac{N}{m} + \frac{N}{n}$ terms of the series $A(X)$. Designing a block version of the Wiedemann algorithm is interesting because it allows a partial distribution of the computation of the a_k's across several machines, each of them computing for instance a given column of all the a_k's. This achieves coarse-grain parallelization of the computation of the a_k's. Coppersmith was interested in the case of $\mathbb{F}_2$: an n-bit machine can compute a whole line of $B^k y$ from $B^{k-1} y$ in one single operation, performing n binary multiplications (that is, bitwise ANDs) at a time.

5.2. connections with vector generating polynomials

The inputs to the block Wiedemann algorithm are the matrix B, which is given, and the matrices x and y. We choose x at random, while y is chosen as Bz, for a random vector block z. This is necessary to ensure that a solution to the equation $Bw = 0$ will be produced. Let us see what are the expected values for the parameters s and d associated with $A(X)$. Our presentation does not pretend to give a full account on what types of degeneracy can show up. We refer to Coppersmith (1994), Kaltofen (1995) and Villard (1997a,b) for this matter.

We claim that we can choose d to be the first integer such that the span of the column vectors $(B^{\mathrm{T}})^l x_j,\ 0 \leq j < m, 0 \leq l < d$, is equal to the full span of these vectors, when

[†]It has been suggested to us that a baby-step/giant-step approach in the spirit of Kaltofen and Villard (2001) could help. However such a thing is not doable here since, B being large and sparse, the computation of $B^{\sqrt{N}}$ would have a prohibitive cost in time and space.

taken for all l. Indeed, for this integer, there exists for each k in $[\![0, m-1]\!]$ a collection of scalar coefficients $\lambda_{j,k,l}$ such that:

$$(B^{\mathrm{T}})^d x_k = \sum_{j=0}^{m-1} \sum_{l=0}^{d-1} \lambda_{j,k,l} (B^{\mathrm{T}})^l x_j. \tag{1}$$

Equivalently, if $D(X)$ is the $m \times m$ matrix whose (j,k)th entry is the polynomial $\sum_{l=1}^{d} \lambda_{j,k,d-l} X^l$, we have:

$$(B^{\mathrm{T}})^d x_k = [X^d] \sum_{j=0}^{m-1} \sum_{l=0}^{d-1} (B^{\mathrm{T}})^l x_j X^l \lambda_{j,k,l} X^{d-l},$$

$$(B^{\mathrm{T}})^d x = [X^d] \left(\left(\sum_{l \geq 0} X^l (B^{\mathrm{T}})^l x \right) D(X) \right),$$

$$0 = [X^d] \left(\left(\sum_{l \geq 0} X^l (B^{\mathrm{T}})^l x \right) (D(X) - I_m) \right).$$

Multiplying equation (1) by any power of B^{T} on the left, we obtain more generally that all the coefficients after the d-th in the polynomial matrix product above are zero. We can also take the product with y on the left, and transpose the result, to obtain that $A(X)$ has a left rational form, with unimodular denominator $I_m - D(X)$, of degree less than or equal to d. The value of d will typically be $\lceil \frac{N}{m} \rceil$, since for this value the span envisioned above is the span of a collection of more than N vectors. Generically, if the projection incurred by x is not too bad, and if the matrix B^{T} does not have eigenvalues with large multiplicities, their span includes the full image of B^{T}.

As for s, which is such that the columns of $a_0, \ldots, a_{s-1}$ span all of $K^{m \times 1}$, its existence depends on good projection properties of x and y. It is highly likely as soon as N is big compared to m and n that s exists, and that in fact $s = \lceil \frac{m}{n} \rceil$, since this value of s gives us at least m vectors to choose from in order to span an m-dimensional space.

So we have typically $d = \lceil \frac{N}{m} \rceil$, and $s = \lceil \frac{m}{n} \rceil$. Of course, these values are not rigorously proven, and for real cases, they might be slightly greater. The theoretical analyses of the block Wiedemann algorithm in Coppersmith (1994), Kaltofen (1995) and Villard (1997a,b) study the deviation of s and d from their typical value, and recommend (in short) that we add an $O(1)$ component to these terms in order to avoid possible failure with very particular input, like matrices having many eigenvalues with large multiplicities. Applying Theorem 3.2, it follows that using $L = \frac{N}{m} + \frac{N}{n} + O(1)$ terms, we are able to compute a linear generator whose degree is $\frac{N}{n} + O(1)$ (terms like $\frac{m}{n}$ are included in the $O(1)$).

From now on, our context will be the one described in this paragraph. The quantities $B, N, d, s, L, m, n, x, z$, and y will correspond to the aforementioned.

5.3. DIFFERENT STAGES OF THE ALGORITHM

The computation of the coefficients of $A(X)$, which will be named "stage BW1", is done sequentially. A vector variable Y is repeatedly updated by $Y \leftarrow BY$, and dot products $x^{\mathrm{T}} Y$ are computed at each step. Once we have $A(X)$ at our disposal, we can infer a linear generator for this matrix sequence, using the tools we have already mentioned (for

example, we can use the MSLGDC algorithm). This will be the step BW2 of the block Wiedemann algorithm. We quickly show that such a linear generator yields a solution to the system $Bw = 0$. Suppose that we obtained a vector generating polynomial, that is, a pair $(u(X), v(X))$ and an integer δ satisfying:

$$A(X)u(X) = v(X), \qquad \delta(u, v) \leq d.$$

Writing down which coefficients are zero, we have:

$$\forall t, t \geq \delta, \qquad \sum_{k=0}^{\delta} ([X^{t-k}]A)([X^k]u) = 0,$$

$$\text{i.e. } \sum_{k=0}^{\delta} x^{\mathrm{T}} B^{t-k} y[X^k]u = x^{\mathrm{T}} B^{(t-\delta)} \sum_{k=0}^{\delta} B^{\delta-k} y[X^k]u = 0.$$

$$\forall t \geq 1, \qquad x^{\mathrm{T}} B^t \sum_{k=0}^{\delta} B^{\delta-k} z[X^k]u = 0.$$

Then, the quantity $w = \sum_{k=0}^{\delta} B^{\delta-k} z[X^k]u$ is orthogonal to the span of the $(B^{\mathrm{T}})^t x_i$'s, for $t \geq 1$. As said before, we can assume that this span is equal to the full image of B^{T} (this might fail if B has many eigenvalues with large multiplicities). This means, then, that Bw is necessarily zero, meaning that w is a solution to our linear system $Bw = 0$ if $w \neq 0$. In the case $w = 0$ but as a "polynomial" in B, w has a non-zero valuation ν, then we have $B^{1+\nu}\hat{w} = 0$ for $\hat{w} = \sum_{k=0}^{\delta-\nu} B^{\delta-k} z[X^k]u$, and some $B^t\hat{w}$ is guaranteed to be a solution if $\hat{w} \neq 0$. The computation of $\hat{w}$ and t such that $B^t\hat{w}$ is a solution is named step BW3.

6. Complexity Analysis and Optimization

Having a block version of the Wiedemann algorithm introduces a new flexibility: we can play with parameters m and n. Nevertheless, these parameters do have some optimal value that we had better use: obviously, the bigger m and n, the shorter the computation of the a_k's, but also the more tedious the computation of a solution from these. We will therefore detail the complexity of the different stages (BW1, BW2, BW3) of the algorithm with respect to m, n, and N. For step BW2, we will give complexities for both Coppersmith's and our algorithm.

The block approach allows coarse grain parallelization (see Coppersmith, 1994 or Kaltofen and Lobo, 1999). In a parallel or distributed setting, distributing the columns of a vector block Y across several machines allows one to compute $Y \leftarrow BY$ in a real time that does not depend on n (if we have that many machines available). It is important to note here that this distribution requires no communication at all between the machines taking part in the computation. Steps BW1 and BW3 can take advantage of this, and therefore the real time is the appropriate measure for the algorithm. One can also regard this as the parallel complexity using a given number of computers, and a communication complexity in $O(1)$. Now the question is: provided that the hardware we have access to allows us several values for m and n, how to choose them in order to achieve the lowest total real time? This is answered in Theorems 6.2 and 6.3. Since m and n are typically limited by the available hardware, it is reasonable to assume that m and n are bounded by a constant. Therefore, at least for the complexity of step BW2

using our recursive algorithm, we will incorporate this in the complexity equation, and focus on the dominating term.

In order to obtain complexity measurements we will use the constant M_1 which has been defined previously (the time for multiplying together two elements of K), as well as an additional constant, M_0, which is the time needed for multiplying a coefficient of the matrix B (typically of size equal to one machine word) by an element of K. Also, we denote by γ the average number of non-zero entries of rows of B (B is expected to be sparse, so γ is small). We do not take additions into account in our analysis. This is an excessive simplification over small fields like $\mathbb{F}_2$, but reasonable over larger fields. We prove the following results:

THEOREM 6.1. *The different steps of the block Wiedemann algorithm require the following real time:*

BW1 $(\gamma\mathsf{M}_0 + m\mathsf{M}_1)\frac{m+n}{mn}N^2$ *using* n *computers* (*see also remark below*).
BW2 $\mathsf{M}_1\frac{m+n}{2}N^2 + O(N)$ *using Coppersmith's algorithm*
$c(\phi)\mathsf{M}_1\frac{(m+n)^3}{mn}N\log^2 N + O(N\log N)$ *using our algorithm* (*provided that* m *and* n *are in* $O(\log N)$), *using one computer.*
BW3 $\gamma\mathsf{M}_0\frac{1}{n}N^2$, *using* n *computers.*

PROOF. As said before, step BW1 is accomplished by repeating the operation $Y \leftarrow BY$, where $Y = y$ initially. This sums up as nL matrix times vector product, but since the n columns of Y are assumed to be treated on separate computers, the real time is the time needed for L applications of B: $\gamma N\mathsf{M}_0 L$. Furthermore, we have to add the cost of the dot products $({x_i}^{\mathrm{T}}Y_j)$. These cost $m\mathsf{M}_1 N$ at each step, hence the result (see also remark below). As for step BW2, the result follows from Coppersmith (1994) for Coppersmith's algorithm, and from Theorem 4.2 for our algorithm, specialized to m and n bounded. The third result follows from the degree of $\hat{w}$ as a "polynomial" in B being $\frac{N}{n}$. □

REMARK. In practice, the real time needed for step BW1 can be lowered down to $\gamma\mathsf{M}_0\frac{m+n}{mn}N^2$ by using vectors of the canonical basis for the x_i's. Indeed, the dot products which account for the $m\mathsf{M}_1\frac{m+n}{mn}N^2$ term become trivial (one operation instead of $m\mathsf{M}_1 N$ with random x's). It should be noted however that when we do so, x is no longer truly random, and the correctness analyses of Kaltofen (1995) and Villard (1997a,b) do not necessarily apply.

We will now write the overall cost of the block Wiedemann algorithm, in the light of Theorem 6.1. Our analysis is valid over fields other than $\mathbb{F}_2$, since the numerous possible tricks in that case tend to shape the results differently.

THEOREM 6.2. *Using Coppersmith's algorithm to handle step* BW2, *the real time for the block Wiedemann algorithm is lowest for* $n_{\mathrm{opt}} = 2\sqrt{\frac{\gamma\mathsf{M}_0}{\mathsf{M}_1}}$, *and* $m_{\mathrm{opt}} = 0.7n_{\mathrm{opt}}$. *The total time needed in this case is* $\mathsf{W}_{\mathrm{opt}} = 3.4\sqrt{\gamma\mathsf{M}_0\mathsf{M}_1}N^2$.

PROOF. Applying Theorem 6.1, we obtain directly (recall that $\phi = \frac{m}{m+1}$):

$$\mathsf{W} = \gamma\mathsf{M}_0\frac{m+n}{mn}N^2 + \mathsf{M}_1\frac{m+n}{2}N^2 + \gamma\mathsf{M}_0\frac{1}{n}N^2,$$

$$\mathsf{W} = \left(\gamma \mathsf{M}_0 \left(1 + \frac{1}{\phi}\right)\frac{1}{n} + \mathsf{M}_1 \frac{n}{2(1-\phi)}\right) N^2.$$

If we minimize W for a given ϕ, the optimal values W_{opt} and n_{opt} are:

$$n_{\text{opt}} = \sqrt{\frac{\gamma \mathsf{M}_0 2(\phi+1)(1-\phi)}{\mathsf{M}_1 \phi}},$$

$$\mathsf{W}_{\text{opt}} = 2N^2 \sqrt{\gamma \mathsf{M}_0 \mathsf{M}_1 \frac{\phi+1}{2\phi(1-\phi)}}.$$

The minimum value of the quantity $\frac{\phi+1}{\phi(1-\phi)}$ is obtained for $\phi = \sqrt{2}-1$. Specializing n_{opt} and W_{opt} to this yields the announced values. □

THEOREM 6.3. *Using algorithm* MSLGDC *for step* BW2*, the real time for the block Wiedemann algorithm is lowest for* $n_{\text{opt}} = 0.6\sqrt{\frac{\gamma \mathsf{M}_0 N}{\mathsf{M}_1 \log^2 N}}$*, and* $m_{\text{opt}} = 0.5 n_{\text{opt}}$*. The total time needed in this case is* $W_{\text{opt}} = 13.8\sqrt{\gamma \mathsf{M}_0 \mathsf{M}_1} N\sqrt{N}\log N$*.*

PROOF. Following Theorem 6.1, W writes down as:

$$\mathsf{W} = \gamma \mathsf{M}_0 \left(\phi + \frac{1}{\phi}\right)\frac{1}{n}N^2 + c(\phi)\mathsf{M}_1 \frac{1}{\phi(1-\phi)^2} nN \log^2 N.$$

Following the same reasoning as before, we obtain the optimum at $\phi \approx 0.3$, and hence the announced W_{opt} and n_{opt}. □

It should be noted that Theorem 4.2 yields a low complexity for step BW2 with respect to m and n because of the introduction of the FFT which hides the cubic dependency on $m+n$. If we had used algorithm MSLGDC with a generic multiplication algorithm, W_{opt} would certainly be higher.

The optimal value n_{opt} is not necessarily acceptable, because we are limited by the available hardware. We will see in the following section that for realistic examples, n_{opt} is still reasonable.

7. Implementation Concerns

7.1. INTEREST OF THE BLOCK VERSION

The consequences of our analysis depend on the base field. We excluded $\mathbb{F}_2$ due to the extreme particularity of this case. For linear algebra problems encountered for example within discrete logarithm computations, the base field is large. M_1 is then typically much bigger than M_0: indeed, the coefficients of the input matrix are usually kept bounded to a size of one machine word (see below), so when a generic element of K has size about 10 words, M_0 is a dozen machine cycles, whereas M_1 can reach several hundreds of machine cycles. Therefore, the second part of the algorithm could end up dominating the overall cost. If we include these considerations in the computation of the optimal value n_{opt} for the parameters m and n, we see that if one uses Coppersmith's version of step BW2, n_{opt} is very small (sometimes hardly above 1). In other words, there is not much

interest in using the block version of the Wiedemann algorithm. On the other hand, our algorithm MSLGDC yields a bigger optimal value. In the experiments we did, it turned out to be worthwhile to have n strictly greater than 1.

7.2. INFLUENCES ON INPUT FILTERING

Our computations also have an interesting consequence on the input given to the block Wiedemann algorithm, when it comes out of a structured Gaussian elimination program (Pomerance and Smith, 1992), or more generally any filtering stage like in Cavallar (2000). Such algorithms aim at reducing the matrix size with minimal fill-in—we want the matrix to remain sparse—, as far as this is possible. Their output is then given to an algorithm like Wiedemann's, or alternatively a block version. Depending on the context, reduction rates going from one half to one tenth are achieved. When the base field is not simply $\mathbb{F}_2$, the matrices given on input to the filtering program have small coefficients. Therefore, coefficients of the matrix are stored in a single machine word and not allowed to go beyond this in order to reduce the memory storage. In the course of this filtering, one usually arranges for stopping it as soon as the estimated subsequent cost (of the Wiedemann algorithm for instance) starts to rise up again, after having been diminished. See Weber and Denny (1998) for an example of such an estimation. The estimated cost is generally something like γN^2, where N is the number of rows of the matrix, and γ the average number of non-zero entries per row. As the filtering proceeds, γ grows while N gets smaller.

Our point here is that when using the block version with optimal parameters m and n, we can focus on the quantity $\sqrt{\gamma}N^2$ instead. This means that we are able to continue the Gaussian elimination a bit further. If we plan to use our subquadratic alternative, the relevant figure is $N\sqrt{\gamma N}\log N$, but this is only valid as long as m and n remain small. Experiments with matrices coming from discrete logarithm problems showed that the filtering can actually be brought substantially further.

7.3. MEMORY REQUIREMENTS

We hardly addressed the memory concerns for the block Wiedemann algorithm. However, these are important because the memory storage needed for the matrix B is usually huge. For this very reason, parallelization or distribution is hampered by the relative scarcity of computing resources available that can handle such a big object: if we plan to distribute step BW1 among several machines with no communication overhead, these have to work on a local copy of the matrix B. This is why having n_{opt} reasonable was crucial. This being said, while the memory is definitely an issue for step BW1, things do not get worse with step BW2, since at this point one can consider that the sequence $A(X)$ has been computed, and that memory storage for the matrix B is no longer needed. Therefore, the increased memory requirements of algorithm MSLGDC compared to Coppersmith's algorithm are not very important.

An important point in the ability, explained in the previous subsection, to carry out the filtering or structured Gaussian elimination further than what we used is that this helps in reducing the storage needed for B; the memory requirements for step BW1 are driven by two quantities: γN for the matrix size, and N for the size of all the linear storage data and such. Continuing the filtering further than before makes these quantities lower,

Table 1. Timings for experiments with MSLGDC.

Field	L	m	n	Coppersmith	MSLGDC	Threshold
$\mathbb{F}_{2^{127}-1}$	1 000	4	4	35 s	36 s	958
	10 000			1 h 01 min	14 min	
	100 000			≈4 d	6 h 10 min	
$\mathbb{F}_{2^{607}-1}$	1 000	4	4	112 s	118 s	923
	10 000			3 h 03 min	45 min	
	100 000			≈12 d	19 h 34 min	
	242 304			≈75 d	47 h 48 min	
$\mathbb{F}_{2^{607}-1}$	10 000	10	20	≈5 d	1 h 57 min	880
$\mathbb{F}_{2^{1279}-1}$	1 000	4	4	267 s	292 s	916
	10 000			7 h 15 min	1 h 50 min	
	100 000			≈30 d	47 h 38 min	

and therefore the algorithm could become more usable if its memory requirements are reduced.

8. Experiments With Algorithm MSLGDC

Algorithm MSLGDC has been implemented in ANSI C, using the big integer multiplication library GMP (Granlund, 1996). The input sequences were all chosen arising from runs of the block Wiedemann algorithm, with base fields which were large prime finite fields. Our FFT code used extensions of the base field to obtain roots of unity. Restricting ourselves to base fields of the form $\mathbb{F}_p$, where $p = 2^k - 1$ is a Mersenne prime, we had sufficiently many roots of unity available in a degree two extension. We mentioned before that another approach that works pretty well in practice consists in doing multiplications in $\mathbb{F}_p[X]$ via integer FFT, using a packing/padding technique. As alluded to before, the algorithm actually implemented did not descend recursively to the tiniest sub-problem. Instead, computation of the $\pi^{(0,b)}$ matrices for b below a certain threshold was delegated to a variant of the quadratic algorithm proposed by Coppersmith. The evaluation of this threshold was done by simple trial and error. Obviously, this value depends on m, n, the base field, and, above all, the exact implementation. Table 1 shows that our algorithm performed well, even for quite small examples. For each example, we give the definition field (it is understood that the Fourier transform operations all take place in a degree two extension), the matrix sequence length (L), its dimension (m and n), the time demanded by our implementation of Coppersmith's quadratic algorithm, and the time required by our algorithm. The (indicative) threshold is in the last column. All timings express runtime on one 667 MHz alpha ev67 CPU.

For the record, we give the timings that we obtained for the real life experiment attached to the large computation of length 242 304. We had $N = 484\,603$, $\gamma = 106.7$, $m = n = 4$. Using four 2-CPUs alpha ev67's at 667 MHz, we did step BW1 in 39 days, step BW2 in 2 days, and BW3 in 20 days. While the setting for m and n was probably not optimal, it is clear that using MSLGDC saved us a lot on this computation.

Several additional points deserve noting concerning the experiments with our algorithm. First, we found it satisfying to remark that the running times could easily be extrapolated with almost no error to obtain estimates for the running times for larger examples (one can check that the timings here fit well with the theory). In the recursive

Table 2. Comparison with results in Lobo (1995).

Field	N	m, n	BW1	Coppersmith	MSLGDC	BW3	Threshold
$\mathbb{F}_{32479}$	10 000	2	4 h 01 min	1 h 12 min		1 h 57 min	
		4	2 h 02 min	2 h 02 min		1 h 04 min	
		8	1 h 05 min	4 h 06 min		34 min	
	20 000	2	29 h 05 min	4 h 38 min		14 h 30 min	
		4	14 h 44 min	8 h 15 min		7 h 17 min	
		8	8 h 07 min	16 h 29 min		3 h 48 min	
$\mathbb{F}_{65537}$	10 000	2	1 h 16 min	52 min	3 min 50 s	38 min	147
		4	38 min	1 h 27 min	7 min 47 s	19 min	132
		8	19 min	2 h 20 min	18 min 56 s	10 min	74
	20 000	2	8 h 58 min	3 h 07 min	8 min 45 s	4 h 31 min	161
		4	4 h 41 min	5 h 10 min	18 min 32 s	2 h 22 min	132
		8	2 h 19 min	9 h 12 min	52 min 01 s	1 h 10 min	80

steps, the cost of the convolution products, which is linear, never became really negligible compared to the cost of the Fourier transforms. For the biggest transforms on the large experiment of length 242 304 over $\mathbb{F}_{2^{607}-1}$, the actual DFT (of order 18) cost was 4 h 11 min, while the convolution cost was 1 h 40 min. We make a final remark on the memory requirements for our program. Of course, the introduction of the FFT tends to make these requirements a bit large. At its peak, the large computation on the sequence of length 242 304 used 11 GB of virtual memory. The machine on which we ran the experiments only had 4 GB of memory, and coped gracefully with this large virtual memory size (we had to add a little disk swap space, though). This is due, of course, to the good locality properties of the FFT algorithm.

Other experiments with the block Wiedemann algorithm are reported in Penninga (1998), Lobo (1995) and Kaltofen and Lobo (1999). Only Lobo's thesis (Lobo, 1995) contains experiments on fields other than $\mathbb{F}_2$. Lobo's results are the experiments over $\mathbb{F}_{32479}$ quoted in Table 2. For the comparison, we tried to solve problems of similar size, on similar hardware. Lobo had 107 MHz sparc processors. We used 143 MHz sparcs, and $\mathbb{F}_{65537}$ as the base field. Apart from steps BW1 and BW3 which do not scale proportionally to the clock ratio, everything appears to fit well with the theory.

9. Conclusion and Further Work

We have presented in this paper a new algorithm, and our experiments seem to indicate that it is rather competitive in comparison to the one proposed by Coppersmith, even for sizes that we consider small, or in any case not unrealistic. We hope that our contribution will help in improving the competitiveness of the block Wiedemann algorithm over large fields.

Several directions can be studied by further work. Of course, it would be highly interesting to make a precise comparison of the running time of our MSLGDC algorithm and the algorithm of Beckerman and Labahn (1994), or other methods. We did not implement these algorithms, and know of no implementation of them (at least in the subquadratic version).

Also, algorithm MSLGDC uses products of matrix formal power series that can be regarded as *short products*. Namely, when the product $e(X)\pi_L(X)$ is computed, we are interested in only part of the result. Recent work showed that a constant factor can be

gained for the complexity of such computations for scalar formal power series (Hanrot *et al.*, 2002). A matrix generalization of this work could help make our algorithm more efficient.

Acknowledgements

I would like to thank Franois Morain who helped me in preparing this paper. I am also grateful to Gilles Villard, Erich Kaltofen, as well as the anonymous referees, for their valuable questions and comments. This research was partially supported by INRIA Action COURBES and the French Ministry of Research—ACI CRYPTOLOGIE.

References

Aho, A. V., Hopcroft, J. E., Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*, Reading, MA, Addison-Wesley.

Beckerman, B., Labahn, G. (1994). A uniform approach for the fast computation of matrix-type Padé approximants. *SIAM J. Matrix Anal. Appl.*, **15**, 804–823.

Bitmead, R. R., Anderson, B. D. O. (1980). Asymptotically fast solution of Toeplitz and related systems of linear equations. *Linear Algebr. Appl.*, **34**, 103–116.

CABAL. 233-digit SNFS factorization. Available online at `ftp://ftp.cwi.nl/pub/herman/SNFSrecords/SNFS-233`, November 2000.

Cavallar, S. (2000). Strategies in filtering in the number field sieve. In Bosma, W. ed., *Proceedings of the 4th Algorithmic Number Theory Symposium, ANTS-IV*, LNCS **1838**, pp. 209–231. Berlin, Springer.

Cavallar, S. *et al.* (2000). Factorization of a 512-bit RSA modulus. In Preneel, B. ed., *Proceedings of EUROCRYPT 2000*, LNCS **1807**, pp. 1–18. Berlin, Springer.

Coppersmith, D. (1984). Fast evaluation of logarithms in fields of characteristic two. *IEEE Trans. Inf. Theor.*, **IT-30**, 587–594.

Coppersmith, D. (1993). Solving linear equations over GF(2): block Lanczos algorithm. *Linear Algebr. Appl.*, **192**, 33–60.

Coppersmith, D. (1994). Solving linear equations over GF(2) via block Wiedemann algorithm. *Math. Comput.*, **62**, 333–350.

Gaudry, P. (2000a). An algorithm for solving the discrete log problem on hyperelliptic curves. In Preneel, B. ed., *Proceedings of the EUROCRYPT 2000*, LNCS **1807**, pp. 19–34. Berlin, Springer.

Gaudry, P. (2000b). Algorithmique des courbes hyperelliptiques et applications à la cryptologie. Thèse, École polytechnique.

Granlund, T. (1996). `GMP`, the GNU multiple precision arithmetic library. Homepage at `http://www.swox.com/gmp`.

Gustavson, F. G., Yun, D. Y. (1979). Fast algorithms for rational Hermite approximation and solution of Toeplitz systems. *IEEE Trans. Circuits Syst.*, **CAS-26**, 750–755.

Hanrot, G., Quercia, M., Zimmerman, P. (2002). Speeding up the division and square root of power series, manuscript in preparation.

Kaltofen, E. (1995). Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Math. Comput.*, **64**, 777–806.

Kaltofen, E., Lobo, A. (1999). Distributed matrix-free solution of large sparse linear systems over finite fields. *Algorithmica*, **24**, 331–348.

Kaltofen, E., Villard, G. (2001). On the complexity of computing determinants. In *Proceedings of the Fifth Asian Symposium on Computer Mathematics (ASCM 2001), Singapore*, pp. 13–27. Singapore, World Scientific Publishing Company.

LaMacchia, B. A., Odlyzko, A. M. (1990). Solving large sparse linear systems over finite fields. In Menezes, A. J., Vanstone, S. A. eds, *Proceedings of CRYPTO '90*, LNCS **537**, pp. 109–133. Berlin, Springer.

Lobo, A. (1995). Matrix-free linear system solving and applications to symbolic computations. Ph.D. Thesis, Rensselaer Polytechnic Institute.

Montgomery, P. L. (1995). A block Lanczos algorithm for finding dependencies over GF(2). In Guillou, L. C., Quisquater, J.-J. eds, *Proceedings of EUROCRYPT '95*, LNCS **921**, pp. 106–120. Berlin, Springer.

Morf, M. (1980). Doubling algorithms for Toeplitz and related equations. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 954–959. New York, IEEE.

Odlyzko, A. M. (1985). Discrete logarithms in finite fields and their cryptographic significance. In Beth, T., Cot, N., Ingemarsson, I. eds, *Proceedings of EUROCRYPT '84*, LNCS **209**, pp. 224–314. Berlin, Springer.

Penninga, O. (1998). Finding column dependencies in sparse matrices over $\mathbb{F}_2$ by block Wiedemann. In Report MAS-R9819, Amsterdam, The Netherlands, Centruum voor Wiskunde en Informatica, available from `http://www.cwi.nl/`.

Pomerance, C., Smith, J. W. (1992). Reduction of huge, sparse matrices over finite fields via created catastrophes. *Exp. Math.*, **1**, 89–94.

Rissanen, J. (1972). Realizations of matrix sequences. Technical Report RJ-1032, IBM Research, Yorktown Heights, New York, NY, T. J. Watson Research Center.

Thomé, E. (2001a). Fast computation of linear generators for matrix sequences and application to the block Wiedemann algorithm. In Mourrain, B. ed., *Proceedings of the ISSAC '2001*, pp. 323–331. New York, ACM Press.

Thomé, E. (2001b). Computation of discrete logarithms in $\mathbb{F}_{2^{607}}$. In Boyd, C., Dawson, E. eds, *Proceedings of ASIACRYPT '2001*, LNCS **2248**, pp. 107–124. Berlin, Springer.

Thomé, E. (2002). Discrete logarithms in GF(2^{607}). Email to the `NMBRTHRY` mailing list, available at `http://listserv.nodak.edu/archives/nmbrthry.html`.

Villard, G. (1997a). A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Research Report 975, Grenoble, France, LMC-IMAG.

Villard, G. (1997b). Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems. In Küchlin, W. W. ed., *Proceedings of the ISSAC '97*, pp. 32–39. New York, ACM Press.

Weber, D., Denny, T. (1998). The solution of McCurley's discrete log challenge. In Krawczyk, H. ed., *Proceedings of CRYPTO '98*, LNCS **1462**, pp. 458–471. Berlin, Springer.

Wiedemann, D. H. (1986). Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theor.*, **IT-32**, 54–62.

von zur Gathen, J., Gerhard, J. (1999). *Modern Computer Algebra*, Cambridge, Cambridge University Press.

Received 12 November 2001
Accepted 27 February 2002