# Solving Sparse Systems with the Block Wiedemann Algorithm

## Efficient Implementation over $\mathrm{GF}(2)$

Sonia Belaid Sylvain Lachartre

Séminaire SALSA

*2011, 8 July*

# Outline

# Outline

# Introduction

- Wiedemann's algorithm was **introduced** by Wiedemann in 1986 [?]
- It was **extended** by Coppersmith in 1994 [?] to perform parallel computations
- It was **improved and used** by Emmanuel Thomé [?] and an international team of researchers in 2009 to break a 768 bits RSA key [?]

# Scalar Wiedemann

- Input : **large sparse matrix** $B$
- Computation of the **linearly recurrent sequence**

$$x^T \cdot B^k \cdot z$$

  with $x$ and $z$ random vectors
- Computation of the **minimal polynomial** $\mu(X) = X \cdot \mu'(X)$ (because $B$ is singular) using Berlekamp-Massey
- Exhibition of the **kernel vector** $v$ such that

$$v = \mu'(B)$$

# Outline

# Outline

## Principle

**Block Wiedemann Algorithm :**

- is used for the resolution of linear systems on finite fields
- takes a **sparse matrix** and exhibits a **kernel vector**
- is organized into **three** steps :
    1. BW1 : computes a matrices series $A = (a_k)_{1 \leq k \leq L}$
    2. BW2 : computes a linear generator $F_g$ of $A$
    3. BW3 : exhibits a kernel vector $w$
- is **probabilistic** (depending on the matrix characteristics)

# Main Algorithm

**Algorithm 1** Wiedemann

**Input:** matrix $B \in \mathrm{GF}(2)^{N\times N}$
**Output:** vector $w \in \mathrm{GF}(2)^{N}$

1. $(A, z) \leftarrow$ BW1($B$) // $z \in \mathrm{GF}(2)^{N\times n}$ (random)
2. $F_g \leftarrow$ BW2($A$) // $F_g \in \mathrm{GF}(2)^{N}$ (linear generator)
3. $w \leftarrow$ BW3($F_g, B, A, z$)
4. **return** $w$

- **Input** : large sparse matrix $B$
- **Output** : kernel vector $w$
- **N** : input matrix dimension
- **m, n** : columns number of random blocks $x$ and $z$

# BW1 : Sequence $A$

- The first step consists in computing the series

$$\mathbf{A}(\mathbf{X}) \leftarrow \Sigma_k a_k \cdot X^k \quad \text{with} \quad a_k \leftarrow x^T \cdot B^k \cdot z \ \in \mathrm{GF}(2)^{m \times n}$$

- Only the first $\mathbf{L} = \frac{\mathbf{N}}{\mathbf{m}} + \frac{\mathbf{N}}{\mathbf{n}} + \mathbf{1}$ coefficients are needed

**Algorithm 2** BW1

**Input:** matrix $B$
**Output:** polynomial $A \in \mathrm{GF}(2)[[X]]^{m \times n}$, matrix $z \in \mathrm{GF}(2)^{N \times n}$

1. $(x, z) \leftarrow$ random matrices $\in \mathrm{GF}(2)^{N \times m} \times \mathrm{GF}(2)^{N \times n}$
2. $v \leftarrow B \cdot z$
3. for $k \leftarrow 0$ to $L$ do
4. $\quad A[k]$ [a] $\leftarrow x^T \cdot v$
5. $\quad v \leftarrow B \cdot v$
6. endfor
7. return $A$

a. $A[k]$ represents the degree $k$ coefficient of $A$

# BW2 : Linear Generator

**Coppersmith**'s generalization of **Berlekamp-Massey** algorithm :

- **Initialization** : $m+n$ candidates $F_j$ for the vectorial generator

$$A(X) \cdot F(X) = G(X) + X^t \cdot E(X)$$

  with $t$ the step number depending on $A$ at the beginning.

- **Iteration** : error $E(X)$ reduced by multiplying the previous equality by $P$ s.t. :

$$E[0] \cdot P = 0$$

- **Termination** : generator discovered when its error is zero

$$F_g(X) \;\text{ s.t. }\; A(X) \cdot F_g(X) = G_g(X) \;\; (E_g(X) = 0)$$

# BW2 : Initialization

| | |
|---|---|
| **d** | **constant** $d \leftarrow \lceil \frac{N}{m} \rceil$ |
| **s** | $F^{(t_0)}$ **degree** s.t. columns of $A[0], \ldots, A[s-1]$ form a basis of $\mathrm{GF}(2)^m$ |
| **t** | **step number** starting with $t \leftarrow s$ |
| **end** | **stop condition** end $\leftarrow 0$ |
| $\delta$ | **quantity** s.t. $\delta(f, g) = max(\deg f, 1 + \deg g)$ |
| $\Delta$ | **degrees bounds** $$\forall j, \quad \delta(F_j, G_j) \leq \Delta_j$$ |
| **F** | **generator candidates** $$\left( \begin{array}{c|ccc} I_n & X^{s-i_1} \cdot r_1 & \ldots & X^{s-i_m} \cdot r_m \end{array} \right)$$ |

# BW2 : Iteration (1/2)

- computation of polynomial $\mathbf{P}$ of degree 1

$$E[0] \cdot \mathbf{P} = 0$$

- the condition becomes

$$\begin{aligned} A \cdot F^{(t)} \cdot \mathbf{P} &= G^{(t)} \cdot \mathbf{P} + X^t \cdot E^{(t)} \cdot \mathbf{P} \\ A \cdot F^{(t+1)} &= G^{(t+1)} + X^{(t+1)} \cdot E^{(t+1)} \end{aligned}$$

- update of $F$ and $E$

$$\begin{aligned} \mathbf{F}^{(\mathbf{t+1})} &= F^{(t)} \cdot \mathbf{P} \\ \mathbf{E}^{(\mathbf{t+1})} &= E^{(t)} \cdot \mathbf{P} \cdot \frac{1}{X} \end{aligned}$$

**Algorithm 3** PMatrix

**Input:** matrix $E0 \in \mathrm{GF}(2)^{m\times(m+n)}$, tab $\Delta \in \mathbb{N}^{m+n}$
**Output:** polynomial $P \in \mathrm{GF}(2)[X]^{(m+n)\times(m+n)}$

/* Sort */
1. $(P[0], \Delta) \leftarrow Sort(\Delta)$
2. $E0 \leftarrow E0 \cdot P[0]$
/* Gaussian Elimination */
3. $(E0, P[0]) \leftarrow GaussElimE0(E0, P[0])$
/* Elimination of non zero columns */
4. for $i \leftarrow 1$ to $m+n$ do
5. if ($E0_i$ [a] is not null) then
6. $P \leftarrow MultByX(P, i)$
7. $\Delta_i \leftarrow \Delta_i + 1$
8. endif
9. endfor
10. return $P$

a. $E0_i$ represents the $i^{th}$ column of $E0$

# BW2 : Termination

- mean value $\overline{\Delta}$ of $\Delta_j$ coefficients increases by $\frac{m}{m+n} = \frac{1}{2}$

$$t - \overline{\Delta} = (t - s) \cdot \frac{n}{m+n}$$

- for $t = s + \lceil \frac{m}{m+n} d \rceil$, $t - \overline{\Delta} \geq d \quad \Rightarrow \quad \exists\, j, \quad t - \Delta_j \geq d$
- according to theorem 8.6 in [?]

$$\exists\, j, \quad t - \Delta_j \geq d \quad \Rightarrow \quad E_j(X) = 0$$

**Stop Condition**

$\mathbf{F_j}$ generator if $t - \Delta_j \geq d$ (before $s + \lceil \frac{m}{m+n} d \rceil$ steps)

# BW2 : Algorithm

**Algorithm 4** BW2

**Input:** polynomial $A$
**Output:** polynomial $F \in \mathrm{GF}(2)[X]^n$

/* Initialization */
1. $(d, s, t, end, \Delta, F) \leftarrow Init(A)$
2. $E \leftarrow Error(A, F, t)$

/* Iteration */
3. while $(end = FALSE)$ do
4. $\quad (P, end, g) \leftarrow PMatrix(E[0], \Delta)$
5. $\quad F \leftarrow F \cdot P$
6. $\quad E \leftarrow E \cdot P \cdot \frac{1}{X}$
7. $\quad t \leftarrow t + 1$
8. endwhile

/* Termination */
9. return $F_g$

# BW3 : Kernel Vector

**Kernel vector** exhibition :

- coefficient $j$ of $A(X) \cdot F_g(X)$

$$(A \cdot F_g)[j] = x^T \cdot B^{j - \deg F_g + 1} \cdot v \quad \text{with} \quad v = \sum_{i=0}^{\deg F_g} B^{\deg F_g - i} \cdot z \cdot F_g[i]$$

- by construction

$$(A \cdot F_g)[j] = 0 \quad \textit{for } j \geq \delta(F_g)$$

- $B^{\delta(F) - \deg F_g + 1} \cdot v$ orthogonal to all vectors $(B^T)^i \cdot x_k$
- if these vectors form a basis of $K^N$

$$\boxed{B \cdot (\underbrace{B^{\delta(F_g) - \deg F_g} \cdot v}_{\text{Kernel Vector}}) \; = \; 0}$$

# BW3 Algorithm

**Algorithm 5** BW3

**Input:** polynomial $F_g$, matrix $B$, polynomial $A$, matrix $z$
**Output:** vector $w$

1. $w \leftarrow 0_N$
2. for $i \leftarrow 0$ to $\deg F_g$ do $w \leftarrow B \cdot w + z \cdot F_g[i]$ endfor
3. if $(w \neq 0_N)$ then
4. for $k \leftarrow 0$ to $\delta(F_g)$ [a] do
5. $u \leftarrow B \cdot w$
6. if $u = 0$ then return $w$ endif
7. $w \leftarrow u$
8. endfor
9. endif
10. return FAILED

a. $\delta(F_g) = max(\deg F_g, \deg(A \cdot F_g) + 1)$

# Outline

# RSA Attack

**2009**

Researchers broke a **768 bits** RSA key using **NFS** [?]

- NFS (Number Field Sieve) : factorization of large numbers
  - using Wiedemann's algorithm
  - input binary matrix :
    - 200 millions of rows
    - 150 non zeros elements by rows
  - 98 days of computations on a cluster of 576 cores

# Outline

# Outline

# Library M4RI [?]

- Linear algebra library in C language focused on dense matrices over $\mathrm{GF}(2)$
- Created by Gregory Bard
- Now maintained by Martin Albrecht

- We use it to compute operations on dense matrices
  - BW1 : to compute the products of $x \cdot (B^k \cdot z)$
  - BW2 : to perform all the operations involving blocks
  - BW3 : to compute the products $(B \cdot z) \cdot F$

# Sparse Matrix

- Structure for the input sparse matrix :
  - dimensions `m`, `n`
  - number of non zeros `nb`
  - number of non zeros by rows `sz`
  - positions of non zeros `pos`
  - rows structures with the same characteristics `l`

## Matrix Example

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

```
m,n = 5,   nb=6
sz = [1,0,2,2,1]
pos = [0,1,4,2,3,4]
l[0] : nb=1, pos=[0]
l[1] : nb=0, pos=[] ...
```

# Sparse Dense Operations

**Algorithm 6** Sparse Dense Product

**Input:** matrix $B$, matrix $v$
**Output:** matrix $res \leftarrow B \cdot v$

$res \leftarrow 0$
$p \leftarrow B_{pos}$
**for** $i \leftarrow 0$ **to** $N$ **do**
    **for** $j \leftarrow 0$ **to** $B_{inb}$ **do**
        $res_i \leftarrow res_i \oplus v_p$
        $p \leftarrow p + 1$
    **endfor**
**endfor**
**return** $res$

- Complexity : linear with the number of non zeros.

# Outline

# Counting Sort

array to sort

$\Delta$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 9 | 9 | 8 | 12 | 8 | 9 | 8 | 12 | 8 |

counting array

| | | | | | |
|---|---|---|---|---|---|
| values | 8 | 9 | 10 | 11 | 12 |
| occurrences | 4 | 3 | 1 | 0 | 2 |

sorted array

CountingSort($\Delta$)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 8 | 8 | 8 | 9 | 9 | 9 | 10 | 12 | 12 |

## Complexity

**Linear** in $O(m + n)$ with $m + n$ the size of $\Delta$

# Outline

# Practical Optimization : Operations Save

**Loop**

1. **for** $i \leftarrow 0$ **to** $a \times b$ **do**

**Loop**

1. $constant\ \ tmp \leftarrow a \times b$
2. **for** $i \leftarrow 0$ **to** $tmp$ **do**

**Product**

**BW1** : $\forall k \in [1..L]$,
$$v \leftarrow B^k \cdot z$$

**BW3** : $\forall k \in [1..\deg F_g]$,
$$v \leftarrow B^k \cdot z$$

**Product**

**BW1** : $\forall k \in [1..L']$,
$$v \leftarrow Bz_{save}[k] \leftarrow B^k \cdot z$$

**BW3** : $\forall k \in [1..\deg F_g]$,
$$v \leftarrow Bz_{save}[k]$$

# Practical Optimization : Reduction of the Number of Tests

**Test**

1. **for** $i \leftarrow 1$ **to** $n$ **do**
2. **if** $(i < \frac{n}{2})$ **then** Action1
3. **else** Action2 **endif**
4. **endfor**

**Test**

1. **for** $i \leftarrow 1$ **to** $\frac{n}{2}$ **do**
2. Action1
3. **endfor**
4. **for** $i \leftarrow \frac{n}{2}$ **to** $n$ **do**
5. Action2
6. **endfor**

# Practical Optimization : Adaptation of M4RI[1] Functions

- removing **initial tests**
  - $A \cdot B$ : number of $A$'s columns = number of $B$'s rows
  - equality to zero
- adapting to my matrices **constant dimensions**
  - BW1 : products of matrices $(64 \times N) \times (N \times 64)$
  - BW2 : products of matrices $(64 \times 128) \times (128 \times 128)$
  - global : most matrices whose dimensions are multiples of the size a machine word
- **improving** some functions
  - `mzd_is_zero` : stopping at the first non zero word
  - `mzd_transpose` : clearing before transposing

1. released 2009-05-12

# Practical Optimization : Parallelization

- **BW1** : parallel
- **BW2** : sequential $\Rightarrow$ parallelization
- **BW3** : sequential $\Rightarrow$ operations save

# Practical Optimization : Parallelization BW1

- sparse dense product : $B \cdot (B^k \cdot z),\ k \in [1..L]$
  - each thread dedicated to a number of rows
  - depending on number of non zeros by rows

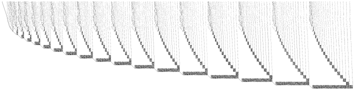


Figure: Representation of a $2688 \times 2688$ matrix

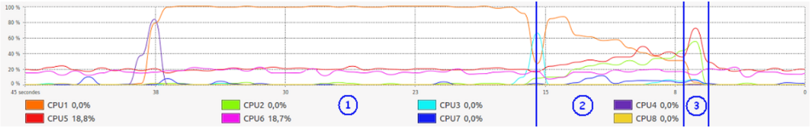
- computations of $a_k$ : $a_k \leftarrow x \cdot (B^k \cdot z),\ k \in [1..L]$
  - each thread dedicated to a number of $a_k$
  - same number of coefficients for each thread

# Practical Optimization : Parallelization BW2

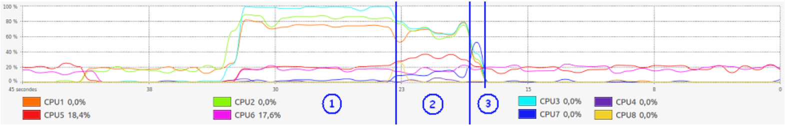
- polynomials products $F \cdot P$ and $E \cdot P$
  - each thread dedicated to a number of coefficients of $F$ or $E$
  - same number of coefficients for each thread
- product of $P$ by $X$
  - each thread dedicated to a number of columns of $P$
  - same number of columns for each thread

# Practical Optimization : Parallelization : CPU Use

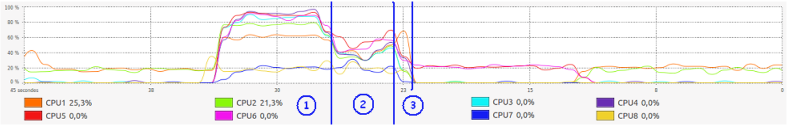
1 thread




3 threads

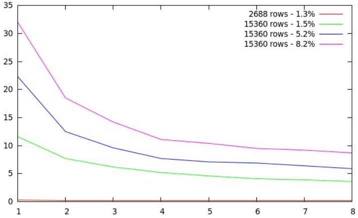



6 threads



# Practical Optimization : Parallelization : Execution Time

Execution times according to the number threads



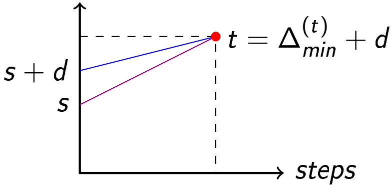

| Dim | $\delta(\%)$ | 1th | 2th | 3th | 4th | 5th | 6th | 7th | 8th |
|---|---|---|---|---|---|---|---|---|---|
| 2688 | 1.3 | 0.3 | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |
| 15360 | 1.5 | 11.6 | 7.7 | 6.2 | 5.2 | 4.6 | 4.1 | 3.9 | 3.6 |
| 15360 | 5.2 | 22.3 | 12.5 | 9.6 | 7.7 | 7.1 | 6.9 | 6.4 | 5.9 |
| 15360 | 8.2 | 32.0 | 18.5 | 14.2 | 11.1 | 10.4 | 9.5 | 9.2 | 8.7 |

# Theoretical Optimizations : Termination Tests

## Stop Condition

$$\exists j \in [1..m+n], \quad t \geq \Delta_j + d$$

$\Delta_{min}^{(t)} + d$ and t




## New Stop Condition

$$\exists\, j \in [1..\frac{m+n}{2}+1] \;\; s.t. \;\; \Delta_j^{(t)} = \Delta_j^{(t-1)}, \quad t \geq \Delta_j + d$$

# Theoretical Optimizations : Error Degree Update (1/2)

- for large matrices
  - high error degree in BW2
  - product $E \cdot P$ **expensive**
- but **only E**[**0**] used in each step
- knowing the number of reminding steps, we can :
  - determine the **useful degree** of $E$
  - decrease the number of coefficients to update

# Theoretical Optimizations : Error Degree Update (2/2)

- Stop condition : $t_f = min(\Delta)^{(t_f)} + d$
- Worst case
  - $min(\Delta)$ increases by 0.5 at each step
  - from step $t$, we still have $\delta_e^{(t)}$ iterations :

$$\begin{aligned} t + \delta_e^{(t)} &= min(\Delta)^{(t)} + \frac{\delta_e^{(t)}}{2} + d \\ \Rightarrow \quad \delta_e^{(t)} &= 2 \cdot (min(\Delta)^{(t)} + d - t) \end{aligned}$$

  - $\Rightarrow \begin{cases} \delta_e^{(t+1)} = \delta_e^{(t)} & \text{if} \quad min(\Delta)^{(t+1)} = min(\Delta)^{(t)} + 1 \\ \delta_e^{(t+1)} = \delta_e^{(t)} - 2 & \text{otherwise} \end{cases}$

## Error Degree Bound

$$\forall t, \quad \deg E^{(t)} \leq \delta_e^{(t)}$$

## Theoretical Optimizations : Candidates Degree Update (1/2)

- for large matrices
  - candidates degree increases by 1 at each step (becoming high)
  - product $F \cdot P$ becomes **expensive**
- but **only $\mathbf{F_g}$** (linear generator) will be kept
- knowing the number of reminding steps, we can :
  - determine the **useful degree** of $F$ (that is the one of $F_g$)
  - limit the number of coefficients to update

# Theoretical Optimizations : Candidates Degree Update (2/2)

- $F$ degree is limited to $\delta_f$ such that

$$\delta_f^{(t)} = min(\Delta)^{(t)} + r^{(t)}$$

with $r^{(t)} = 2 \cdot (min(\Delta)^{(t)} + d - t)$ (maximum reminding steps)

$$\Rightarrow \quad \delta_f^{(t)} = min(\Delta)^{(t)} + 2 \cdot (min(\Delta)^{(t)} + d - t)$$

- $\Rightarrow \begin{cases} \delta_f^{(t+1)} & = & \delta_f^{(t)} + 1 & \text{if } min(\Delta)^{(t+1)} = min(\Delta)^{(t)} + 1 \\ \delta_f^{(t+1)} & = & \delta_f^{(t)} - 2 & \text{otherwise} \end{cases}$

## Candidates Degree Bound

$$\forall t, \quad \deg F^{(t)} \leq \delta_f^{(t)}$$

# Outline

# Outline

# Execution Times

| Dim | $\delta(\%)$ | BW1 | BW2 | BW3 | Global |
|---|---|---|---|---|---|
| 738 | 7.00 | 10% | 90% | 0% | $0''100$ |
| 3422 | 7.57 | 20% | 73% | 7% | $0''450$ |
| 10000 [2] | 5.97 | 46% | 43% | 11% | $1''960$ |
| 27000 [1] | 2.20 | 60% | 28% | 12% | $14''060$ |
| 73674 | 0.61 | 65% | 30% | 5% | $1'12''760$ |
| 93913 | 0.20 | 51% | 41% | 8.3% | $1'38''710$ |

Figure: Performances on different matrix sizes

## Computer Details : PC Xeon (64 bits)

- 8 Go RAM
- 2.40 GHz
- 8 processors Intel(R) Xeon(R) CPU

2. from Faugère, 2011-06-30

# Outline

# Comparisons with Magma and Sage

| Matrices | | Our Implementation | | Speed up (1th) | |
|---|---|---|---|---|---|
| Dim | $\delta$ | 8 th | 1 th | Magma | Sage |
| 2688 | 1.27% | 0″200 | 0″360 | × 5.8 | × 150.0 |
| 15360 | 1.54% | 3″580 | 11″300 | × 85.8 | — [3] |
| 15360 | 5.20% | 5″870 | 21″670 | × 62.4 | — [2] |
| 15360 | 8.21% | 8″690 | 31″480 | × 62.3 | — [2] |

Figure: Temporal comparisons with Magma[4] and Sage[5]

3. not enough memory ($>$ 8 Go)
4. Magma version 2.17-1, released 2010-12-02
5. Sage version 4.6.2, released 2011-02-25

# Outline

- Summary
  - Efficient implementation of the Block Wiedemann algorithm over $\mathrm{GF}(2)$ in C language
  - Practical and theoretical optimizations
  - Encouraging results compared to existing methods

- Further Work
  - *Algebraic Cryptanalysis of HFE Cryptosystems Using Gröbner Bases* [?] using Block Wiedemann algorithm
  - Use of Gröbner bases algorithm [?]
  - Comparisons with LinBox
  - Work on applications
  - Open source ?

# Bibliography