Parallel Gaussian Elimination for Gröbner bases computations in finite fields

Jean-Charles Faugère
INRIA, Paris-Rocquencourt Center, SALSA Project
UPMC, Univ Paris 06, LIP6
CNRS, UMR 7606, LIP6
UFR Ingénierie 919, LIP6
Case 169, 4, Place Jussieu, F-75252 Paris
Jean-Charles.Faugere@inria.fr

Sylvain Lachartre
Thales Communications - Laboratoire Chiffre
160, boulevard de Valmy
92700 Colombes
Sylvain.Lachartre@fr.thalesgroup.com

ABSTRACT

Polynomial system solving is one of the important area of Computer Algebra with many applications in Robotics, Cryptology, Computational Geometry, etc. To this end computing a Gröbner basis is often a crucial step. The most efficient algorithms [6, 7] for computing Gröbner bases [2] rely heavily on linear algebra techniques. In this paper, we present a new linear algebra package for computing Gaussian elimination of Gröbner bases matrices. The library is written in C and contains specific algorithms [11] to compute Gaussian elimination as well as specific internal representation of matrices (sparse triangular blocks, sparse rectangular blocks and hybrid rectangular blocks). The efficiency of the new software is demonstrated by showing computational results fr well known benchmarks as well as some crypto-challenges. For instance, for a medium size problem such as Katsura 15, it takes 849.7 sec on a PC with 8 cores to compute a DRL Gröbner basis modulo $p < 2^{16}$; this is 88 faster than Magma (V2-16-1).

Categories and Subject Descriptors

I.1.2 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation—*Algorithms: Algebraic algorithms*; F.2.2 [**Theory of Computation**]: Analysis of algorithms and problem complexity—*Non numerical algorithms and problems: Geometrical problems and computation*; D.4.6 [**Software**]: Operating Systems—*Security and Protection: Cryptographic controls*

General Terms

Algorithms.

Keywords

Polynomial systems solving, Gröbner bases, Gaussian Elimination, High Performance Linear Algebra, Cryptography, Multi-core Programming.

1. INTRODUCTION

The most efficient algorithms [6, 7] for computing Gröbner bases [2] rely heavily on linear algebra techniques. More precisely, the main cost in Gröbner bases computation is the Gaussian reduction of matrices constructed from polynomials of the ideal generated by the input equations. The matrices generated by these algorithms have unusual properties: sparse, almost block triangular and not necessary full rank. Moreover, most of the pivots are known at the beginning of the computation.

Unfortunately, although `M4RI` [1] has good performances in $\mathbb{F}_2$, the best linear algebra packages such as `ATLAS` [13], `LinBox` [4], `FFLAS-FFPACK` [5] or `Sage` [12] are very efficient for dense linear algebra, but not tuned for F_4/F_5 matrices in word-size prime fields. In [11], we have presented a dedicated efficient algorithm for computing Gaussian elimination of such matrices. The main idea consists in decomposing the initial matrix in four submatrices obtained from both lists of pivot and non pivot rows and columns, and to treat them specifically. To benefit as much as possible from the cache memory, each matrix is split into small blocks and the reduction relies on three elementary block operations. To deal with the specific structures of the matrices occurring in a Gröbner basis computation we distinguish three block formats: sparse triangular blocks, sparse rectangular blocks and hybrid rectangular blocks (the internal representation can be sparse or dense, in adequation with the eventual rows densification occurring during the computation). At the end of this paper we report some timings and speedup to show the efficiency of the new library and to compare with existing linear algebra packages.

2. GRÖBNER BASES AND LINEAR ALGEBRA

Notions about Gröbner bases and how to compute them using linear algebra are not described here (see [3, 6, 7] for instance).

A list of polynomials $[f_1, \cdots, f_s]$ can be represented by a matrix as follows: columns correspond to all the monomials occurring in the polynomials (sorted with respect to a monomial ordering), and each row contains coefficients of a polynomial with respect to these monomials. The *leading* coefficient of a row denotes the column index of its first non zero coefficient.

$$\left\{\begin{array}{lcl} f_1 & = & \sum_{i=1}^{k} \alpha_{1,i}\, m_i \\ f_2 & = & \sum_{i=1}^{k} \alpha_{2,i}\, m_i \\ & \vdots & \\ f_s & = & \sum_{i=1}^{k} \alpha_{s,i}\, m_i \end{array}\right. \longrightarrow \begin{array}{c} \\ f_1 \\ f_2 \\ \vdots \\ f_s \end{array} \begin{array}{c} \begin{array}{cccc} m_1 & m_2 & \cdots & m_k \end{array} \\ \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} & \cdots & \alpha_{1,k} \\ \alpha_{2,1} & \alpha_{2,2} & \cdots & \alpha_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{s,1} & \alpha_{s,2} & \cdots & \alpha_{s,k} \end{pmatrix} \end{array}$$

To summarize, a Gröbner basis computation can be seen as a sequence of Gaussian eliminations of such matrices. In the next section, we present a new Gaussian elimination algorithm in which operations are performed in a different order. For that purpose, two types of columns in the matrix are distinguished: *pivot columns* (in which a row has its leading term), and *non pivot columns*. Similarly, a non null row chosen to reduce others is called a *pivot row*. The *column pivot set* (resp. *row pivot set*) is the set of all pivot column (resp. row).

The new algorithm use three elementary matrix operations:

- `Trsm`[1] : $Y \leftarrow X^{-1}Y$,
- `Axpy`[2]: $Y \leftarrow AX + Y$,
- `Gauss` : *classical* Gaussian elimination.

2.1 Structure of the matrices

The matrices occurring in the Gröbner basis computation have the following common properties:

- *sparse*: in degree δ a shift of a homogeneous degree d polynomial with n variables has less than $\binom{n+d-1}{d}$ non zero coefficients for $\binom{n+\delta-1}{\delta}$ total columns. For instance, if $d = 3$, $n = 5$ and $\delta = 10$, then the average density for this line is about 3.5%),
- several rows are monomial multiples of the same polynomial f: $(m_1 f, m_2 f, \ldots, m_k f)$,
- the matrices are not necessary full rank (this is the main difference between F_4 and F_5).
- *almost block triangular*: each matrix is constructed by pairwise combinations from a set of polynomials with distinct leading terms (*S-polynomial*), to exhibit new polynomials with new leading terms.

The last point provides the predetermination of a part of the pivot columns. The efficiency of our new algorithm rely on this knowledge.

3. SKETCH OF THE SEQUENTIAL ALGORITHM

This algorithm has been introduced in [11] and it takes into consideration sparsity and almost block triangular shape, with different treatments and representations for the preselected pivot rows and columns, and the other ones. It inputs a $n_0 \times m_0$ matrix M_0 and performs its Gaussian elimination.

3.1 Analysis

The first stage consists in looking for columns clearly identified as pivot. For that purpose, it is enough to sweep the row leading terms. The list of the corresponding columns indices is called C_{piv}, and is of size N_{piv}. Then, one pivot row is chosen from several candidates (in fact all the rows which have the same leading index), to obtain the list R_{piv}, also of length N_{piv} : the coordinates of the i-th pivot will be $\left[R_{piv}[i], C_{piv}[i]\right]$. The list of non pivot rows (resp. columns) is denoted $\overline{R_{piv}}$ (resp. $\overline{C_{piv}}$).

[1]Trsm: TRiangular Solve with Multiple right-hand sides
[2]Axpy: "A X plus Y"

3.2 Decomposition into submatrices

M_0 can be decomposed in 4 submatrices A, B, C, D using the row and column pivot lists:

- A is made from the elements indexed by R_{piv} and C_{piv} (upper triangular $N_{piv} \times N_{piv}$ matrix with diagonal coefficients equal to 1).
- B of dimensions $N_{piv} \times (m_0 - N_{piv})$ contains the elements indexed by R_{piv} and $\overline{C_{piv}}$.
- C is a $(n_0 - N_{piv}) \times N_{piv}$ matrix built from the elements indexed by $\overline{R_{piv}}$ and C_{piv}. Its rows are sorted by increasing leading term indices and with leading coefficient equal to 1.
- D is obtained from the $(n_0 - N_{piv}) \times (m_0 - N_{piv})$ remaining elements (indexed by $\overline{R_{piv}}$ and $\overline{C_{piv}}$).

Figure 1 shows these four submatrices with their respective dimensions.

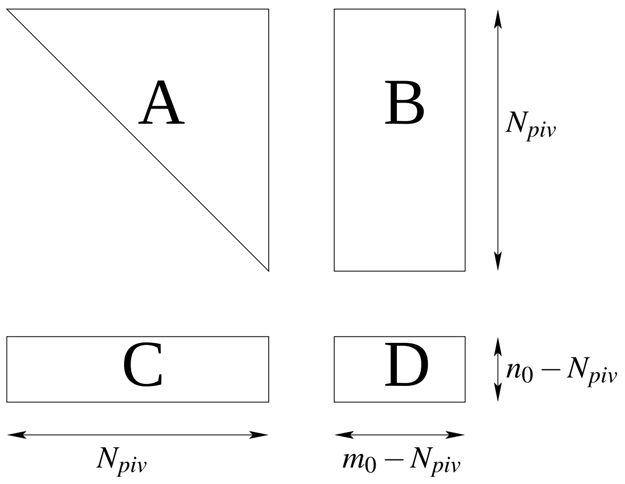

Figure 1: ABCD decomposition

3.3 Pivot row reduction (Trsm)

The third step of the algorithm consists in reducing the pivot rows by themselves. From linear algebra point of view, this means computing $B \leftarrow A^{-1}B$ since A is non singular (upper triangular with 1s on the diagonal). This operation is a basis change for the non pivot columns: it computes their expression in the vector space generated by the pivot columns. Each submatrix is treated differently: A is only read so it remains sparse, whereas the matrix B is accessed in read/write mode, so its density may increase. When B is a dense matrix, this computation can be made "in place" to save memory. At this step, the matrix M_0 is equivalent to:

$$M_0 \sim \left(\begin{array}{c|c} Id & A^{-1}B \\ \hline C & D \end{array}\right)$$

3.4 Non pivot rows reduction (Axpy)

Once the pivot rows are reduced, the non-pivot rows must be reduced by these new pivot rows by computing $D \leftarrow D - CB$ (here B denotes the *new* matrix $B \leftarrow A^{-1}B$) and C is set to zero, since all its coefficients are reduced by those of A. M_0 is now equivalent to (wrt. initial matrices A, B, C and D):

$$M_0 \sim \left(\begin{array}{c|c} Id & A^{-1}B \\ \hline 0 & D - CA^{-1}B \end{array}\right)$$

3.5 New pivot row computation (Gauss)

At this point, all the rows of M_0 have been reduced by pivot rows. The next step is to look and find new pivots in the matrix D, with a *Gaussian elimination* (row version of the classical and well-known algorithm): $D \leftarrow \texttt{Gauss}(D)$. Note that the leading terms of D are not necessary equal to 1 anymore, and some field inversions may be further required. Now:

$$M_0 \sim \left(\begin{array}{c|c} Id & A^{-1}B \\ \hline 0 & \texttt{Gauss}(D - CA^{-1}B) \end{array}\right)$$

3.6 Reconstruction

At last, the final matrix Gauss(M_0) is reconstructed from rows and pivots lists R_{piv} and C_{piv}, and from new matrices B and D.

REMARK 1. *The final matrix is not in row reduced echelon form. To obtain $rref(M_0)$ a second iteration of this new algorithm must be applied (see [11] for details) : the last step (3.5) gives two new lists of pivot rows and columns (so a new decomposition and the* `Trsm` *and* `Axpy` *steps can be performed once again before reconstructing the final rref matrix).*

4. PARALLEL IMPLEMENTATION

Operations on $\overline{C_{piv}}$ columns are independent, so they can be performed in parallel. In this section, we present the data structures we used to implement a parallel version of the new algorithm.

4.1 Data structures

We take into account the architecture with last generation processors, while respecting the structure of matrices from F_4/F_5 algorithms. To benefit from the cache processor memory, and to maintain an optimal stream of data, matrices are reorganized by row and column blocks. We use three block matrix formats : *sparse*, *dense* and *hybrid* (rows are stored in sparse or dense format according to their density). Moreover, three blocks sizes have to be fixed:

- K_{AB} (resp. $K_{A_{rl}}$): row and column block (resp. incomplete block) size of matrix A (common block size of columns of A and C, and rows of B),
- K_{CX} (resp. $K_{C_{rl}}$): row block (resp. incomplete block) size of matrices C and D,
- K_{BY} (resp. $K_{B_{rc}}$): column block (resp. incomplete block) size of the matrices B and D,

where $K_{A_{rl}}$, $K_{C_{rl}}$ and $K_{B_{rc}}$ are the dimensions of incomplete blocks, respectively equal to:

$$\begin{cases} K_{A_{rl}} & \equiv \quad N_{piv} \mod K_{AB}, \\ K_{C_{rl}} & \equiv \quad n_0 - N_{piv} \mod K_{CX}, \\ K_{B_{rc}} & \equiv \quad m_0 - N_{piv} \mod K_{BY}. \end{cases}$$

Before giving a more formal description, figure 2 presents the global block layout: the numbered blocks in matrices and the dotted arrows of a block inner row symbolize the storage order of the elements in the memory.

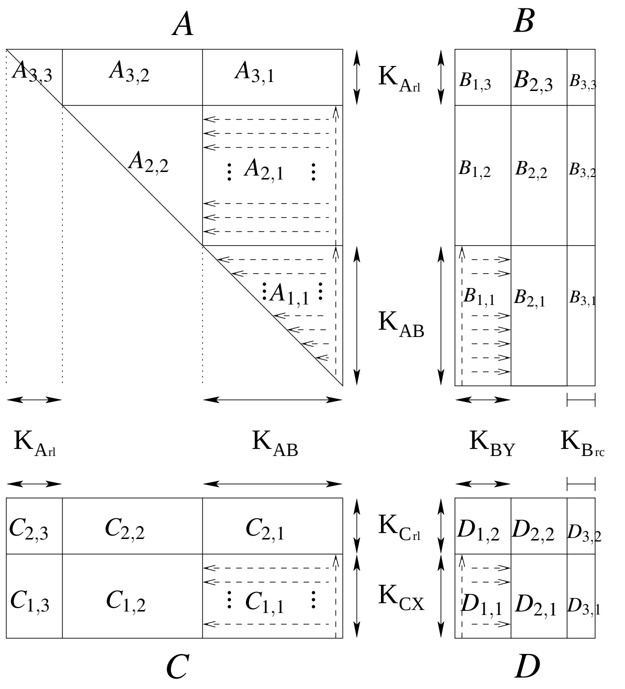

Figure 2: Matrices A, B, C and D block division

4.2 Block inner operations

This section deals with operations within a block. We distinguish three block formats:

1. *Sparse triangular block format*: applies to triangular blocks of the matrix A. It uses three lists: A_{val}, A_{pos} and A_{nb}, which represent respectively the values, the positions and the number of non zero elements in each row of the matrix A. Elements as well as rows are sorted by increasing order, from bottom to top. Row leading coefficients (equal to 1) and the last row of the block are not stored.
2. *Sparse rectangular block format*: this is the format of the rectangular blocks of matrices A and C. Three lists are also necessary to store the value, the position and the number of non-zero elements of each row in the block. Rows are sorted by decreasing order, from bottom to top. For the blocks of A, the positions of the non-zero elements are decreasing, from right to left, while for C, these are written in increasing order, from left to right.
3. *Hybrid rectangular block format*: used for the blocks of matrices B and D. Rows are stored in *hybrid* format: their representation is sparse or dense, according to the number of non-zero elements. Rows are ordered by decreasing indices, from bottom to top, while the row elements by growing indices, from left to right.

The layout of blocks in matrices is one of the following three formats:

1. *Block format of sparse triangular matrix:* uses sparse triangular and sparse rectangular blocks. Blocks are ordered by rows from right to left, and from bottom to top. Rectangular blocks have K_{AB} rows while triangular blocks have $K_{AB} - 1$

rows (since the leading coefficients, always equal to 1, are not stored).

2. *Block format of sparse rectangular matrix:* only contains *rectangular sparse blocks* stored by rows. The block layout is the same that the *sparse triangular matrix* format.

3. *Block format of hybrid rectangular matrix:* consists of hybrid rectangular blocks ordered from top to bottom, and from left to right.

EXAMPLE 1. *To illustrate each one of these three formats, we present three matrices A, B and C of dimensions $n \times m$ with block size K and density threshold d (for better legibility, a zero row or column is represented by the the empty set $\emptyset$ for value and position, and 0 for the number). For hybrid blocks, a threshold density d is chosen to determine whether a row has a sparse or a dense representation (ie. if the density is greater than the threshold):*

- *Sparse triangular block matrix format $n = m = 5$, $K = 2$:*

$$A = \left(\begin{array}{c|cc|cc} 1 & 5 & 2 & 0 & 0 \\ \hline 0 & 1 & 4 & 8 & 3 \\ 0 & 0 & 1 & 6 & 0 \\ \hline 0 & 0 & 0 & 1 & 7 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

$$A_{val} \quad \begin{array}{|c|cc|c|c|c|cc|} \hline 7 & 6 & 3 & 8 & 4 & \emptyset & 2 & 5 \\ \hline \end{array}$$

$$A_{pos} \quad \begin{array}{|c|ccc|c|c|cc|} \hline 1 & 2 & 1 & 2 & 1 & \emptyset & 1 & 2 \\ \hline \end{array}$$

$$A_{nb} \quad \begin{array}{|c|cc|c|c|c|} \hline 1 & 1 & 2 & 1 & 0 & 2 \\ \hline \end{array}$$

- *Sparse rectangular block matrix format, $n = 3$, $m = 5$ and $K = 2$:*

$$C = \left(\begin{array}{c|cc|cc} 8 & 6 & 1 & 9 & 0 \\ \hline 4 & 0 & 0 & 5 & 0 \\ 7 & 0 & 0 & 2 & 3 \end{array}\right)$$

$$C_{val} \quad \begin{array}{|ccc|c|cc|c|cc|c|} \hline 3 & 2 & 5 & \emptyset & 7 & 4 & 9 & 1 & 6 & 8 \\ \hline \end{array}$$

$$C_{pos} \quad \begin{array}{|ccc|c|cc|c|cc|c|} \hline 1 & 2 & 2 & \emptyset & 1 & 1 & 2 & 1 & 2 & 1 \\ \hline \end{array}$$

$$C_{nb} \quad \begin{array}{|c|c|cc|cc|c|c|c|} \hline 2 & 1 & 0 & 0 & 1 & 1 & 1 & 2 & 1 \\ \hline \end{array}$$

- *Hybrid rectangular block matrix format, $n = 5$, $m = 3$, $K = 2$ and $d = 50\%$:*

$$B = \left(\begin{array}{cc|c} 0 & 2 & 5 \\ \hline 4 & 0 & 0 \\ 7 & 1 & 0 \\ \hline 0 & 0 & 3 \\ 6 & 8 & 0 \end{array}\right)$$

$$B_{val} \quad \begin{array}{|cc|cc|c|c|c|c|c|} \hline 6 & 8 & 7 & 1 & 4 & 2 & 3 & \emptyset & 5 \\ \hline \end{array}$$

$$B_{pos} \quad \begin{array}{|c|cc|c|c|c|} \hline \emptyset & 1 & 2 & \emptyset & \emptyset & \emptyset \\ \hline \end{array}$$

$$B_{nb} \quad \begin{array}{|cc|cc|c|cc|cc|c|} \hline 2 & 0 & 2 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ \hline \end{array}$$

This ordering is in perfect adequacy with the *double spacial and temporal principle* (see [13] for example) and so, benefits from the *cache memory* (small and fast memory taking advantage of two principles : a program is more likely to spend its time executing code around the same set of instructions, and tend to run in loops repeating the same instructions).

Algorithm 1 performs $B \leftarrow A^{-1}B$ between *sparse triangular block A* and *hybrid rectangular block B* ($D \leftarrow D - CB$ block algorithm follows the same philosophy). It uses a dense temporary row denoted `Temp` (rows must be converted from hybrid to dense format when copying rows from B to `Temp`, and from dense to hybrid format when updating B from `Temp`). Sparse or dense linear algebra (Axpy) is used according to the density of hybrid rows of B.

Algorithm 1: $B \leftarrow A^{-1}B$: block "hybrid" version

```
   Inputs   : sparse triangular block block $A$
              hybrid rectangular block $B$
   Output   : hybrid rectangular block $B = A^{-1}\ B$
   Local    : Temp is a $m_0 - N_{piv}$ temporary dense row
   Notation: $X[i,*]$ is the i-th row of $X \in \{A,B\}$
   /* $A$ rows loop                                          */
1  for $i \leftarrow N_{piv} - 1$ to 1 do
       /* Hybrid format to dense format                      */
2      Temp $\leftarrow$ Hybrid2Dense($B[i,*]$)
       /* $A$ i-th row loop ($A_{nb}[i] - 1$ elements)        */
3      for $j \leftarrow 2$ to $A_{nb}[i]$ do
4          $Av \leftarrow A_{val}[i,j]$, $Ap \leftarrow A_{pos}[i,j]$
5          if Density($B[Ap,*]$) $\leq Threshold$ then
               /* Sparse :  Temp $\leftarrow$ Temp $- Av * B[Ap,*]$   */
6              Temp $\leftarrow$ SparseAxpy(Temp,$Av$,$-1$,$B[Ap,*]$)
7          else
               /* Dense :  Temp $\leftarrow$ Temp $- Av * B[Ap,*]$    */
8              Temp $\leftarrow$ DenseAxpy(Temp,$Av$,$-1$,$B[Ap,*]$)
       /* Dense format to hybrid format                      */
9      $B[i,*] \leftarrow$ Dense2Hybrid(Temp)
10 return $B$
```

4.3 Block outer operations

The outer operations are performed on matrix blocks: each operation $B \leftarrow A^{-1}B$ and $D \leftarrow D - CB$ uses *block hybrid* algorithms. It is also possible to use a temporary dense block to store the results of the partial block products.

4.4 Block hybrid Gaussian elimination

The search of new pivots (see 3.5) has to be adapted to the *block hybrid* format of the matrix D (Gauss algorithm operating on *hybrid blocks*). Here, the Gaussian elimination is performed on successive blocks (by increasing indices) of the new matrix D obtained in 3.4. A $\#\overline{R_{piv}} \times \#\overline{R_{piv}}$ matrix P, equivalent to a *pseudo inverse*, is introduced to keep a track of the successive row operations. In the i-th stage, the i-th block D_i is updated by left-product by P, and then Gaussian elimination is performed on $D_i \mid P$ (D_i concatenated with P), from rows of indices greater than the partial rank $r_D^{(i)}$. Note that a temporary block is used to store the rows of D_i and P which have to be reduced.

Initially, P is equal to the identity matrix. The Gaussian reduction of the first block concatenated with P is computed. Then, in the i-th stage, the i-th block is updated by a simple left matrix multiplication by P, and then, the a Gaussian reduction is performed on this block concatenated with P. We denote $nz^{(i)}$ the number of non-null rows in the i-th block D_i. Identically, $r_D^{(i)}$ is the rank of the i-th block after Gaussian elimination.

On figure 3, the matrix is represented after the reduction of the first block and has "$nz^{(1)}$" non-null rows. After the first Gaussian block reduction, the first block contains the up-triangular matrix of rank $r_D^{(1)}$. The $nz^{(1)}$ first rows of P contain the linear operations needed by the Gaussian reduction of the first block.

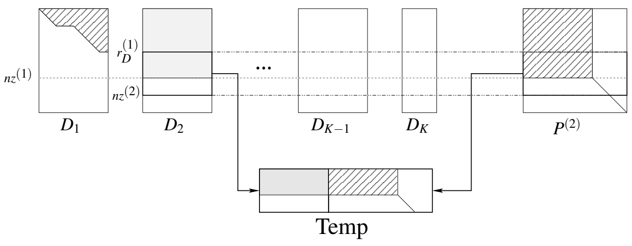

Figure 3: Gaussian block reduction

Then, the non null rows from index $\left(r_D^{(i)}+1\right)$ of D_2 and of P are copied in the temporary block. Finally, the temporary block is reduced by Gaussian elimination, and the submatrices are updated. The temporary rank is then denoted $r_D^{(2)}$.

This process is iterated to obtain the Gaussian reduction of the matrix, which final rank is denoted r_D. Therefore, $\text{rank}(M_0) = N_{piv} + r_D$. The efficiency relies on the number N_{piv} (trivial pivots in M_0): if $\#\overline{R_{piv}} = n_0 - N_{piv}$ is small with respect to n_0, the cost of this hybrid Gaussian block elimination is negligible both in time and memory, comparing to the whole process cost.

Algorithm 2 present this hybrid Gaussian algorithm. The function `FistNonZeroRow`(l,M) returns the index of the first non-null row in the list l of rows of the matrix M. The function `Update`(Temp) copies the temporary rows of Temp in the corresponding rows of D_i and P in a hybrid format. At the end of this algorithm, both matrices P and Temp can be freed from memory.

4.5 Parallelization

During the computation of $B \leftarrow A^{-1}B$ thus $D \leftarrow D - CB$, the operations on the columns of matrices B and D are independent. They can be realized in parallel. For that purpose, matrices B and D must be considered from columns blocks point of view (noted B_i and D_i), and the two elementary parallelizable operations are:

- `Trsm`(i): inputs the block index i and outputs

$$B_i \leftarrow A^{-1}B_i,$$

- `Axpy`(i): inputs the block index i and outputs

$$D_i \leftarrow D_i - CB_i.$$

The hybrid Gaussian elimination algorithm is applied to D_i (denoted `Gauss`(i)) to search for new pivots (after both previous reductions).

During the whole process, Gaussian elimination must be performed as soon as possible. So, we define priority rules between

Algorithm 2: Block hybrid Gaussian elimination

```
Input   : Block hybrid matrix $D$ (dimension $n_D \times m_D$).
Outputs: Block hybrid matrix Gauss($D$) and its rank $r_D$.
   /* Init parameters                                              */
1  $P \leftarrow Id_{n_D}, r_D^{(1)} \leftarrow 0, N \leftarrow \lceil m_D/K_{BY} \rceil$
   /* D blocks loop                                                */
2  for $i \leftarrow 2$ to $N-1$ do
3      $nz^{(i)} \leftarrow$ FirstNonZeroRow($\{r_D^{(i-1)}+1,\dots,n_D\}, D_i$)
4      Temp $\leftarrow$ Gauss(SubMatrix($\{nz^{(i-1)},\dots,nz^{(i)}\}, D_i|P$))
5      $r_D^{(i)} \leftarrow r_D^{(i-1)} +$ Rank(Temp)
6      $(D_i, P) \leftarrow$ Update(Temp)
   /* Last block of D                                              */
7  Temp $\leftarrow$ Gauss(SubMatrix($\{r_D^{(N-1)}+1,\dots,n_D\}, D_N$))
8  $r_D \leftarrow r_D^{(N-1)} +$ Rank(Temp)
9  $D_N \leftarrow$ Update(Temp)
10 return $D$ and $r_D$
```

the three operations `Trsm`, `Axpy` and `Gauss`. Four priority constraints and synchronization points (denoted S_i for i from 1 to 4) are introduced for the parallel algorithm (see figure 4):

- S_1 (from `Analysis` to `Trsm`): no constraint of synchronization,
- S_2 (from `Trsm` to `Axpy`): to compute `Axpy`(i), the computation of `Trsm`(i) must be completed,
- S_3 (from `Axpy` to `Gauss`): to process the reduction `Gauss`(i), `Axpy`(i) must be completed as well as the operation `Gauss`(j) for j between 1 and $i-1$,
- S_4 (from `Axpy` step to the reconstruction step): all the operations of type `Axpy` must be completed.

To keep track of all the operations on reduced blocks by each of the operations, the list of remaining tasks is shared by all processors.During its update, we make sure that no other processor has access to this *critical section*. For that purpose, we use *Mutex* (MUTual exclusion). Algorithm 3 presents a way of parallelizing the computation in order to lower the latency. It uses four lists:

- Function: list of the three block operations (`Trsm`, `Axpy` and `Gauss`),
- Todo: list of the lists of not treated yet block indices for each of the three functions,
- Done: list of the block indices for which the three operations have been performed,
- Pr: list of priorities of each function (since Gauss is sequential, it must be computed as soon as possible, so its priority is 1 and the priority of Axpy is 2; Trsm is the function with less priority).

This algorithm is executed by all the threads and ends when the blocks of all the matrices have been treated by the three operations. At the beginning of the while loop, the thread looks for a task (searching first in the most priority list – ie. Todo$_3$, then Todo$_2$, etc – and denoting $ind = \text{Todo}_{Pr[i]}$ this block index), locks the mutex to update Todo (ie. remove ind from Todo$[i]$: the chosen task has no longer to be treated by the other threads), and performs the

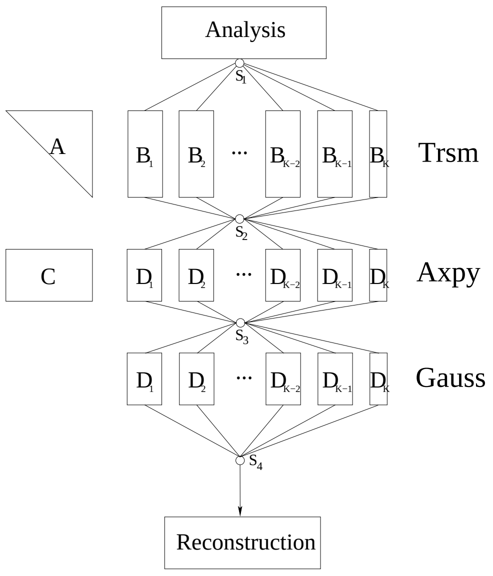
Analysis
S_1
A
B_1 B_2 ··· B_{K-2} B_{K-1} B_K
Trsm
S_2
C
D_1 D_2 ··· D_{K-2} D_{K-1} D_K
Axpy
S_3
D_1 D_2 ··· D_{K-2} D_{K-1} D_K
Gauss
S_4
Reconstruction

Figure 4: New Gaussian algorithm (parallel version)

Algorithm 3: Parallel Gaussian algorithm

```
   Inputs    : matrices $A$, $B$, $C$ and $D$
   Outputs   : matrices $B$ and $D$ after reduction
   Notations: Todo: lists of blocks to be treated by functions,
              Pr: list of function priorities.
1  Todo $\leftarrow$ [ $[1,\ldots,K]$, [ ], [ ] ], Done $\leftarrow$ [ ]
2  Function $\leftarrow$ [Trsm, Axpy, Gauss], Pr $\leftarrow [3,2,1]$
   /* Something to do */
3  while Done $\neq [1,\ldots,K]$ do
       /* search a task from high to low priority */
4      for $i \leftarrow 1$ to 3 do
5          if $\mathrm{Todo}_{\mathrm{Pr}[i]} \neq$ [ ] then
6              Lock()
7              $ind \leftarrow \mathrm{Todo}_{\mathrm{Pr}[i]}[1]$
8              $\mathrm{Todo}_{\mathrm{Pr}[i]} \leftarrow \mathrm{Todo}_{\mathrm{Pr}[i]} \setminus [ind]$
9              Unlock()
               /* The computation is performed */
10             Function$[i](ind)$
11             Lock()
12             if $i \leq 2$ then
                   /* Next operation must be performed on
                      this block */
13                 $\mathrm{Todo}_{\mathrm{Pr}[i+1]} \leftarrow$ Sort($\mathrm{Todo}_{\mathrm{Pr}[i+1]} \cup [ind]$)
14             else
                   /* All the operations are done */
15                 Done $\leftarrow$ Done $\cup\, [ind]$
16             Unlock()
```

computation Function$[i](ind)$. If $i \leq 2$, ind is added to the next list Todo$[i+1]$, else the ind-th block is added to Done (nothing to do with it anymore). Then, the thread goes on until all the blocks have been treated by the three operations.

5. PRACTICAL EXPERIMENTS

We have implemented a small finite field version ($\mathbb{F}_p$ with $3 \leq p \leq 65521$) of this new algorithm in C language (approximately 15000 lines of code) using POSIX threads.

5.1 Comparison with existing linear algebra packages

First, we compute the row echelon form (in [11] we have also described a `Rref` algorithm to compute a row echelon form of matrix) of small matrices occurring in some Gröbner bases applications. We compare the computations in $\mathbb{F}_{65521}$ with several linear algebra tools: Maple 13 (function `RowReduce` from LinearAlgebra and Modular packages), Magma 2.16.1 (function `Nullspace-OfTranspose` on sparse matrices), Sage 3.0.5 (echelon_form on sparse matrices) and Linbox 1.1.6 (`rowReducedEchelon` on SparseMatrix), on the six matrices:

Name	Dimension	Density	Rank
robot	404×302	12.39%	262
katsura7	694×738	7.44%	611
f855	2456×2511	2.78%	2331
cyclic8	4562×5761	9.37%	3903
katsura12	18285×19607	10.50%	15810
cyclic9	72552×93913	0.70%	71872

The tests are run on a pc with two Intel Xeon E5420 processors (with four 2.5 GHz cores each), and 6 Go of RAM, and obtain the following table (*MT* refers to the case of a *memory trash*):

Name (version)	New library	Maple (13)	Sage (3.0.5)	Magma (2.16.1)	Linbox (1.1.6)
robot	<0.1	6.4	2.4	<0.1	<0.1
katsura7	<0.1	40.8	20.92	0.2	0.2
f855	<0.1	841.2	257.11	3.3	4.3
cyclic8	**1.8**	$> 10^5$	$> 10^5$	54.9	33.0
katsura12	**28.5**	MT	MT	1036.81	1166.8
cyclic9	**46.6**	MT	MT	MT	MT

Although these matrices are sparse, for Maple and Sage dense linear algebra is more efficient. Our `Rref` version is more efficient (wrt. to memory and time) than the other tools.

At last, the results of the parallel version of the new algorithm using POSIX threads:

Name	Seq. (s)	Thread number / SpeedUp				
		1	2	4	8	12
cyclic8	**1.8**	1.0	1.8	3.1	**4.7**	4.4
cyclic9	**46.6**	1.0	1.9	3.4	**5.7**	5.4

Note that with two threads, latency periods are almost null, both processors are used at full capacity. The best real times are obtained with eight threads using the eight cores of the machine. However, sequential hybrid last blocks computations and/or bus memory engorgement prevent from optimal performances.

5.2 Comparison with existing Gröbner bases tools

All the timings given in this section are in elapsed seconds and are obtained using our library on a 64 bit Intel Xeon CPU X5570 @ 2.93GHz with 8 cores.

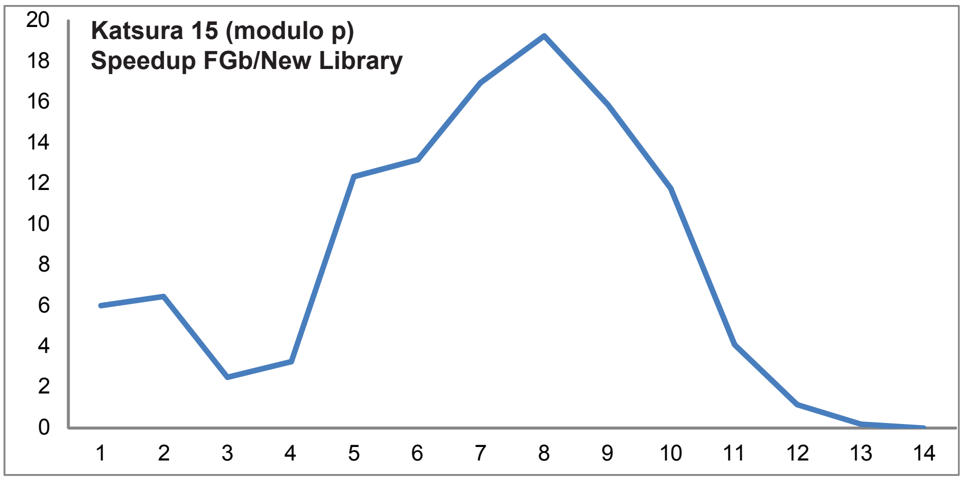

Fig.5.2: relative speedup for the Katsura 15 problem over $\mathbb{F}_{65521}$
(the abscissa corresponds to the stage of the GB computation and the ordinate to the speedup).

The goal now is to try to estimate the real speedup that we can achieve using the new library. In contrast with the previous subsection we have thus to perform Gaussian elimination on several matrices. To start with a well known benchmark we run our new library on the Katsura n problem [10]: since this system is a set of n quadratic equations we know that we have to perform $n+1$ Gaussian eliminations (this is the Macaulay bound for regular systems). In figure 5.2, we compare the new implementation with our reference library FGb. The conclusion is that the new library is always more efficient than the original implementation in FGb except for the last two computations: in that cases the matrices are *quasi-triangular* (triangular with few more rows) the new algorithm is not optimal (the cost of `Trsm` is too important with respect to a classical Gaussian elimination performed in FGb). The same phenomenon occurs for the steps 3 and 4 and that is why the speedup decreases.

Dimension	FGb	New lib(8)	SpeedUp FGb/New	New Seq library	New (seq) /New (8)
1042 x 2135	0.02	0.01	2.00	0.0	2.4
3827 x 6207	0.29	0.06	4.83	0.3	4.7
10014 x 14110	2.10	0.33	6.36	1.8	5.2
19331 x 25143	9.30	1.27	7.32	7.6	6.0
28447 x 35546	23.36	2.87	8.14	18.1	6.3
34501 x 42315	36.38	4.5	8.08	29.4	6.4
38165 x 46265	34.79	5.78	6.02	38.0	6.5
39590 x 47768	19.28	5.94	3.25	38.7	6.5
39965 x 48156	5.90	5.65	1.04	36.3	6.4
40035 x 48227	1.08	1.08	1.00	35.5	6.3
40042 x 48234	0.07	0.07	1.00	35.4	6.3
Total	191.69	27.56	**6.96**		

Katsura 13 modulo 65521 with 8 cores

In the current state of the implementation we have to devise the following strategy: by default to perform Gaussian elimination we call the new library except when the matrix is *quasi-triangular* (there is a threshold to find). When the matrix is *quasi-triangular* we call the old sequential implementation. Note that in practice the previous restriction is not a big deal: the CPU needed to perform Gaussian elimination on the first/last matrices occurring in the computation is negligible compared with the total CPU time. In the rest of the paper, we assume that we always apply this strategy.

5.2.1 *Katsura modulo p*

We present here the detailed results of the Katsura n problems for n from 13 to 16. In some table we also include a comparison between the sequential version of the library (`New Seq Library`) and the 8-cores version of the library (`New Seq Library (8)`). All the timings are in seconds.

Dimension	FGb	New lib(8)	SpeedUp FGb/New	New Seq library	New (seq) /New (8)
1333 x 2804	0.04	0.01	4.00	0.05	4.8
5559 x 9032	0.63	0.11	4.50	0.65	5.9
11683 x 18005	0.52	0.36	1.33	1.8	5.0
21717 x 30783	5.85	1.31	4.50	7.7	5.8
39001 x 50484	93.65	7.11	13.12	49.9	6.7
67933 x 82582	322.19	27.42	12.08	182.85	5.8
70411 x 85376	218.69	21.46	10.33	141.4	6.6
81277 x 97202	332.68	30.72	10.99	215.4	7.0
86547 x 102826	258.66	30.64	8.58	208.5	6.8
88417 x 104786	105.31	28.18	3.76	189.3	6.7
88874 x 105257	28.70	26.92	1.08	176.5	6.6
88954 x 105338	4.72	4.72	1.00	175.1	6.6
88962 x 105346	0.32	0.32	1.00	175.2	6.7
Total	1881.29	180.68	**10.55**		

Katsura 14 modulo 65521 with 8 cores

Dimension	FGb	New lib(8)	SpeedUp FGb/New	New Seq library	New (seq) /New (8)
1667x3608	0.05	0.01	6.00	0.1	8.8
7312x12257	1.40	0.21	6.43	1.2	5.6
17248x26575	2.05	0.82	2.46	4.6	5.6
32109x46154	9.36	2.85	3.25	17.3	6.1
60801x79831	289.77	23.39	12.33	177.4	7.6
114563x140832	830.56	62.93	13.17	422.8	6.7
142062x170248	1454.32	85.78	16.92	558.2	6.5
170221x201111	2351.63	121.53	19.26	858.9	7.1
187664x219868	2275.85	142.23	15.87	865.1	6.1
195325x227973	1513.69	127.6	11.75	871.4	6.8
197778x230530	533.53	129.95	4.06	790.8	6.1
198335x231102	133.15	115.03	1.14	760.1	6.6
198426x231194	20.40	20.40	1.00	729.8	6.5
198434x231202	1.25	1.25	1.00	738.6	6.5
Total	11948.14	849.68	**14.06**		

Katsura 15 modulo 65521 with 8 cores

Steps	Dimension	FGb	New library (8)	SpeedUp FGb/New
1	271 x 968	0	0	
2	2048 x 4565	0.08	0.02	4.00
3	9953 x 16839	2.58	0.38	6.79
4	23290 x 36757	3.86	1.50	2.57
5	45844 x 67046	18.70	6.25	2.99
6	83046 x 114252	108.55	23.96	4.53
7	160426 x 204782	3326.63	186.06	17.88
8	175286 x 214892	3822.91	194.53	19.65
9	328980 x 385905	11295.82	700.92	16.12
10	373624 x 432524	16441.15	890.49	18.46
11	401429 x 464523	19090.29	733.58	26.02
12	426807 x 491659	15294.66	728.41	21.00
13	437603 x 503003	8912.21	867.45	10.27
14	440754 x 506273	3035.39	622.11	4.88
15	441423 x 506958	603.01	595.60	1.01
16	441525 x 507061	84.23	84.23	1.00
17	441534 x 507070	4.84	4.84	1.00
Total		103180.96	5687.29	**18.14**

Katsura 16 modulo 65521 with 8 cores

We can deduce from the previous table that the new library is very efficient. Better results can still probably obtained since we have sometimes a maximal speedup of 26 and sometimes a much lower speedup.

5.2.2 *Minrank*

The Minrank problem is a fundamental linear algebra problem (generalisation of the eigenvalues problem) as was studied recently in Cryptology [8] or in Computer Algebra [9]. In that case, the polynomial system is a list of polynomials of degree 4.

Steps	Dimension	FGb	New library (8)	SpeedUp FGb/New
1	441 x 2002	0.47	0.17	2.76
2	1676 x 4231	0.70	0.10	7.00
3	3657 x 7058	2.06	0.31	6.65
4	5089 x 8985	4.54	0.53	8.57
5	6204 x 10265	4.88	0.85	5.74
6	6594 x 10700	2.06	0.87	2.37
7	6720 x 10835	0.63	0.63	1.00
8	6753 x 10869	0.14	0.14	1.00
9	6758 x 10874	0.02	0.02	1.00
Total		28	3.62	**7.73**

Minrank (9,7,4) with 8 cores

Steps	Dimension	FGb	New library (8)	SpeedUp FGb/New
1	784 x 5005	3.89	0.07	2.76
2	3145 x 10201	7.43	0.68	5.99
3	6989 x 16880	24.25	2.44	7.01
4	11160 x 23270	51.36	4.88	6.94
5	14947 x 28344	96.73	10.72	6.31
6	17421 x 31313	109.04	15.52	5.54
7	18420 x 32477	52.34	15.59	2.85
8	18810 x 32912	20.08	15.52	1.11
9	18936 x 33047	5.92	5.92	1.00
10	18969 x 33081	1.3	1.3	1.00
11	18974 x 33086	0.14	0.14	1.00
Total		512.32	72.78	**7.04**

Minrank (9,8,5) with 8 cores

Dimension	FGb	New lib(8)	SpeedUp FGb/New	New Seq library	New (seq) /New (8)
784 x 5005	3.89	0.07	2.76	1.7	1.4
3145 x 10201	7.43	0.68	5.99	3.9	5.9
6989 x 16880	24.25	2.44	7.01	17.1	7.1
11160 x 23270	51.36	4.88	6.94	35.7	7.4
14947 x 28344	96.73	10.72	6.31	83.1	7.9
17421 x 31313	109.04	15.52	5.54	123.0	8.0
18420 x 32477	52.34	15.59	2.85	122.7	8.0
18810 x 32912	20.08	15.52	1.11	119.6	7.7
18936 x 33047	5.92	5.92	1.00	116.5	7.9
18969 x 33081	1.3	1.3	1.00	108.3	7.6
18974 x 33086	0.14	0.14	1.00	115.4	7.8
Total	512.32	72.78	**7.04**		

Minrank (9,8,5) with 8 cores

Dimension	FGb	New lib(8)	SpeedUp FGb/New	New Seq library	New (seq) /New (8)
1296 x 11440	25.26	7.19	3.51	0.0	1.3
5380 x 22400	55.48	4.16	13.34	23.1	5.6
12224 x 36784	225.41	14.74	15.29	106.9	7.3
21066 x 52502	567.18	46.77	12.13	322.2	6.9
30519 x 67094	1119.4	91.61	12.22	643.8	7.0
38109 x 77687	1724.84	192.08	8.98	1436.6	7.5
43162 x 84027	1808.62	259.87	6.96	2071.4	8.0
45441 x 86801	956.11	245.61	3.89	1953.4	8.0
46440 x 87965	420.85	264.91	1.59	1813.8	6.8
46830 x 88400	154.03	154.03	1.00	1732.6	7.8
46956 x 88535	45.79	45.79	1.00	1697.6	7.2
46989 x 88569	10.03	10.03	1.00	1695.2	5.8
46994 x 88574	1.11	1.11	1.00	1673.5	7.5
Total	8757.62	1337.90	**6.55**		

Minrank (9,9,6) with 8 cores

Even if the new library is less efficient on this example than for the Katsura *n* problem we observe a non linear speedup for huge computations. The 8-core version is also always 6 to 8 times more efficient than the sequential version showing that the parallelization of the algorithm is quite efficient.

5.2.3 *Comparison with Magma 2.16.1*

We compare now our new algorithm with a recent version of the F_4 implantation in Magma.

	F_4 Kat11	F_4 Kat12	F_4 Kat 13
Magma	19.5	151.2	1091.4
FGb	40.6	342.6	2550.65
New library	**2.85**	**19.45**	**149.6**

	F_5 Kat 12	F_5 Kat 13	F_5 Kat 14
Magma	151.2	1091.4	9460.35
FGb	32.8	191.7	1881.3
New library	**4.6**	**27,6**	**180,7**

6. CONCLUSIONS AND PERSPECTIVES

We have shown a parallelized algorithm to perform Gaussian elimination in in order to compute efficiently Gröbner bases. We have applied our implementation on real size and difficult problems (for instance the Minrank problem in Cryptology). Hence our approach is very effective for computing Gröbner bases on a multicore PC. Some work is still necessary to obtain a maximal speedup and to decrease the memory requirement of the new library.

Acknowledgements: The authors would like to thank Olivier Orcière for his helpful remarks.

7. REFERENCES

[1] M. Albrecht and G. Bard. *The M4RI Library – Version 20090409*. The M4RI Team, 2009.

[2] B. Buchberger. An Algorithmical Criterion for the Solvability of Algebraic Systems. *Aequationes Mathematicae*, 4(3):374–383, 1970. (German).

[3] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms : An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, 7 1997.

[4] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. Saunders, W. J. Turner, and G. Villard. Linbox: A Generic Library For Exact Linear Algebra, 2002.

[5] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense Linear Algebra over Word-Size Prime Fields: the FFLAS and FFPACK Packages. *ACM Trans. Math. Softw.*, 35(3):1–42, 2008.

[6] J.-C. Faugère. A New Efficient Algorithm for Computing Groebner bases (F_4). *Journal of Pure and Applied Algebra*, 139(1-3):61–88, 1999.

[7] J.-C. Faugère. A new efficient algorithm for computing Groebner bases without reduction to zero F_5. In *Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation*, 2002.

[8] J.-C. Faugère, F. Levy-dit Vehel, , and L. Perret. Cryptanalysis of Minrank. In D. Wagner, editor, *Advances in Cryptology CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 280–296, Santa-Barbara, USA, 2008. Springer-Verlag.

[9] J.-C. Faugère, M. Safey El Din, and P.-J. Spaenlehauer. Computing Loci of Rank Defects of Linear Matrices using Gröbner Bases and Applications to Cryptology. In S. Watt, editor, *ISSAC '10: Proceedings of the 2010 international symposium on Symbolic and algebraic computation*, New York, NY, USA, 2010. ACM.

[10] K. Katsura. Theory of spin glass by the method of the distribution function of an effective field. *Progress of Theoretical Physics*, 87:139–154, 1986. Supplement.

[11] S. Lachartre. *Algèbre linéaire dans la résolution de systèmes polynomiaux Applications en cryptologie*. PhD thesis, Université Paris 6, 2008.

[12] W. Stein et al. *Sage Mathematics Software (Version 3.3)*. The Sage Group, 2009. `http://www.sagemath.org`.

[13] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).