
**UPMC - Pierre and Marie Curie University**
**MASTER OF SCIENCE AND TECHNOLOGY**
**IN**
**Distributed Systems and Applications**

# Faugère-Lachartre Parallel Gaussian Elimination for Gröbner Bases Computations Over Finite Fields

## Implementation and New Algorithms

by Martani Fayssal

Paris, September 2012

Supervisor: Jean-Charles Faugère
Guénaël Renault
Martin R. Albrecht

Examiner: Olivier Marin

# Contents

# Introduction

Computing Gröbner bases is one of the very important constituents of a computer Algebra system. Gröbner bases provide a unified approach to solving problems expressed in terms of sets of multivariate polynomials, this has many uses in cryptography, coding theory, statistics, robotics etc.

Faugère's F4 [3] and F5 [4] are the most efficient algorithms for computing Gröbner bases; they rely heavily on Linear Algebra. The computation of a Gröbner basis using these algorithms can be considered as succession of Gaussian eliminations over matrices constructed from polynomials generated by the system's input equations. Each row of the matrix corresponds to a polynomial: the columns of the matrix represent the monomials occurring in these polynomials sorted with respect to a monomial ordering; a row then corresponds to the coefficients of a polynomial with respect to these monomials. The list of polynomials $[f_1..f_n]$ is represented according to the monomials $[m_1..m_k]$ in a matrix of size $n \times k$ as follows:

$$\left\{\begin{array}{l} f_1 = \sum_{i=1}^{k} \alpha_{1,i} m_i \\ f_2 = \sum_{i=1}^{k} \alpha_{2,i} m_i \\ f_3 = \sum_{i=1}^{k} \alpha_{3,i} m_i \\ \vdots \\ f_n = \sum_{i=1}^{k} \alpha_{n,i} m_i \end{array}\right. \quad \rightarrow \quad \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{array} \quad \begin{array}{c} \begin{array}{cccc} m_1 & m_2 & \dots & m_k \end{array} \\ \left(\begin{array}{cccc} \alpha_{1,1} & \alpha_{1,2} & \dots & \alpha_{1,k} \\ \alpha_{2,1} & \alpha_{2,2} & \dots & \alpha_{2,k} \\ \alpha_{3,1} & \alpha_{3,2} & \dots & \alpha_{3,k} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{n,1} & \alpha_{n,2} & \dots & \alpha_{n,k} \end{array}\right) \end{array}$$

The matrices constructed from these polynomials have very interesting properties: sparse; almost block triangular; not necessarily full rank; all the rows are unitary; and the number of columns is usually greater than the number of rows. The Faugère-Lachartre algorithm [1] takes advantage of these properties in order to achieve the best efficiency when computing a Gaussian elimination over such matrices.

The best linear algebra packages like Atlas [14], Linbox [13] and Sage [12] are very efficient over dense matrices but not tuned for the matrices issued from the F4/F5 algorithms over word-size prime fields. To this end, the Faugère-Lachartre algorithm was designed having as the main criteria for its efficiency the structure of matrices occurring in Gröbner bases computations.

This internship was motivated basically by the need to provide an open source implementation of the Faugère-Lachartre algorithm with two main goals: being efficient on both CPU performance and memory usage. We have used the LELA [5] library all along our implementation; our work, once mature enough, would eventually be integrated into LELA which is available under the GNU General Public License (GPL).

The matrices issued from Gröbner bases computations have a size varying from several megabytes to dozens of gigabytes; this makes it crucial for any implementation of the Faugère-Lachartre algorithm to have a high CPU and memory usage efficiency. The performance of the program can be assured by the use of high tuned optimizations along with efficient use of the cache memory. Parallelization is also considered for speeding up the computations by means of performing simultaneous operations on many processors. Lastly, the efficient use of memory can be achieved by strategies like early release of data, avoiding memory fragmentation, etc.

The LELA library has a first implementation of the Faugère-Lachartre algorithm; however, LELA's implementation is not very efficient over word-size fields. Indeed, we will show that our implementation can be more than 20 times faster with 4 times less memory usage than LELA's. Moreover, our implementation is parallelizable, and with a new ordering of operations of the Faugère-Lachartre algorithm that we have proposed, our implementation can be even more efficient with less memory footprint.

In this report, we first start by presenting the well-known Gaussian and Gauss-Jordan elimination methods for computing row echelon and reduced row echelon forms of a matrix. We then introduce the Structured Gaussian Elimination method and show the first benefit of efficient memory usage by limiting writing operations to only a restrained memory area; the structured Gaussian elimination algorithm can be 80 times faster than the naïve methods on sparse matrices. We then introduce in details the Faugère-Lachartre algorithm. This algorithm decomposes the original matrix into four sub-matrices whose elements originate from the list of pivot and non-pivot rows and columns, this allows better separation of the elements following their use in the computations: some are read only while others are used in a read/write manner. The rows' elimination steps in the Faugère-Lachartre algorithm are based on the same idea of the structured Gaussian elimination. We then introduce briefly the block and parallel versions of the algorithm.

In the following chapter we list our contributions to the Faugère-Lachartre algorithm. This includes the changes to the original method to suit our implementation along with new algorithms. We present a new parallel algorithm for the structured Gaussian elimination method: this allows us to parallelize the third operation (*Gauss*) of the Faugère-Lachartre algorithm which was not parallelized previously.

Afterwards, a new ordering of operations of Faugère-Lachartre main steps is presented; in this new method, the reductions are performed directly and only on the non-pivot rows. We show that this new method is indeed more efficient than the standard Faugère-Lachartre and has less memory footprint when only a row echelon form is required. In the experimental results, we will see that the new method can be 5 times faster than the standard Faugère-Lachartre on some problems. This is due to the fact that initial matrix doesn't undergo a great density change while a row echelon for is computed.

In the third chapter we report about our implementation. We start by briefly mentioning some of the optimization techniques used and the validation process of the results. We then introduce LELA and its implementation of Faugère-Lachartre along with experimental results about its efficiency. Afterwards, we present the different versions we have developed and the data structures used. The "multiline" data structure is then introduced and we show how it enables us to achieve the best performance especially when coupled with the block representation of the matrices. The new method that we have proposed in the contribution section is then addressed; we show that it leads to better results than the standard method especially on slightly dense matrices. We finish by discussing the memory utilization and the possibility of distributed computations in the case of very large matrices.

CPU Performance and memory usage efficiency comparisons are made with Lachartre's original implementation; they are presented in the Appendix A.

We conclude by final remarks about the overall implementation and a list of improvements and issues to be addressed in future works.

## Related work

As we have mentioned, the LELA library contains an initial implementation of the Faugère-Lachartre algorithm but which is not very efficient on word-size prime fields.

Severin Neumann uses in [15] a different method for his parallelGBC [1] library which takes advantage of the structure of the matrices to minimize the set of operations as in Faugère-Lachartre, but differs from it by performing the parallelization over the rows of the matrix and not the columns. His method has the following main characteristics:

- Unlike Faugère-Lachartre, no pivot search is required; his algorithm does not require an analysis of the matrix to find pivots, since they are predetermined by the construction of the matrix.
- The parallelization is done on the rows and not the columns. The parallel operations by column are done by using SSE instructions.
- The matrix is not fully reduced, and the algorithm can stop after a simple row echelon form is computed.

parallelGBC is avaialable as an open source library at https://github.com/svrnm/parallelGBC.

***Note***: we couldn't make performance comparisons with parallelGBC since it performs the reduction of the matrix along with other steps while computing the Gröbner basis.

[1] Parallel Gröbner Basis Computation

# Gaussian Elimination in $F_p$

# Chapter 1

# Existing Gaussian elimination methods and Faugère-Lachartre

In this chapter we present different methods for computing row echelon forms and reduced row echelon forms using Gaussian elimination-like methods. We start by introducing the naïve Gauss and Gauss-Jordan algorithms in section 1.1; then, in section 1.2, we present the structured Gaussian elimination and its performance efficiency compared to the naïve methods. A detailed description of the Faugère-Lachartre algorithm is then presented in section 1.3 along with the block version in 1.4 and the parallelization in 1.5.

## 1.1 Gaussian and Gauss-Jordan Elimination

Gaussian elimination is an algorithm used for solving systems of linear equations. It can also be used to determine the rank of a matrix, compute its inverse (in case of nonsingular matrices), and calculate the determinant of a matrix among other things.

Using Gaussian elimination, a matrix is reduced to what is known as a "row echelon form" by means of elementary row operations: row multiplication, row addition and row interchanging. The resulting echelon form is not unique, for instance, any multiple of a matrix in an echelon form is also an echelon form. However, using a variant of the Gaussian elimination algorithm known as the Gauss-Jordan elimination, one can reduce a matrix to its "reduced row echelon form," or Rref; this form is unique for every matrix.

**Definition: Row Echelon Form and Reduced Row Echelon Form (Rref)**

A matrix $A \in M_{(n,m)}(K)$, where $K$ is a field, is in row echelon form if:

- Every empty row (a row in which all the elements are zeros) is below all non-zero rows (a row having at least one nonzero element).
- The column index of the leading coefficient of row $i$ (the first non-zero element of a row from the left, also called the pivot element) is strictly greater than the column index of the line just above (the row $i-1$).
- All elements below a leading entry of a row are zero elements (this is implied by the first two requirements).

A matrix is in **reduced row echelon** form if it satisfies, in addition, the following two conditions:

- Every leading coefficient is equal to 1.

- Every leading coefficient of a row is the only non-zero element in its column.

***Note:*** Every matrix has a unique reduced row echelon form.

### 1.1.1 Elementary row operations

Elementary row operations are used in the Gaussian algorithm (resp. Gauss-Jordan algorithm) to transform a matrix to its row echelon form (resp. reduced row echelon form). Elementary row operations preserve the row space of a matrix (and hence its rank), which means that the row space of a matrix is the same as that of its row echelon or reduced row echelon form.

There are three types of elementary row operations:

- Row interchanging $R_i \leftrightarrow R_j$ : any row $i$ can be switched with any other row $j$.
- Row multiplication $R_i \leftarrow c \times R_i$ : a row is multiplied by a non-zero constant (to make rows unitary for example).
- Row addition (or linear combination *axpy*) $R_i \leftarrow R_i + c \times R_j$: each row can be replaced by the sum of that row and the multiple of another (used basically to reduce rows by each other).

***Note:*** For every matrix, a row echelon form (resp. reduced row echelon form) can be obtained using a finite number of elementary row operations.

#### Example

Consider the following matrix in $\mathbb{Z}/7\mathbb{Z}$:

$$A = \begin{pmatrix} 1 & 2 & 6 & 0 & 4 \\ 0 & 0 & 5 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This matrix is in a row echelon form because it satisfies the three conditions mentioned above:

- The second row's entry is at column index 3 (assuming that the first column index is 1) which is greater than the column index of the first row's entry: 1.
- All the empty rows (the $3^{\text{rd}}$one in this case) are below the non-zero rows.
- All the elements below the leading coefficients are zeros.

However, $A$ is not in a ***reduced row echelon form*** since the leading coefficient of the second row is not equal to 1 and there is an element not equal to zero above the leading coefficient of row 2. We will show, after stating the Gauss-Jordan algorithm, how to obtain the *Rref* of this matrix.

### 1.1.2 Naïve Gaussian and Gauss-Jordan Elimination

In this section we present the naïve Gauss and Gauss-Jordan algorithms used to calculate the row echelon form and the reduced row echelon form respectively.

#### Gaussian Elimination

As we have already mentioned, the Gaussian elimination algorithm is used to calculate the row echelon form of a matrix; only elementary row operations are used to achieve this. Since we are interested in Gaussian elimination over a finite field, *we will only focus on producing row echelon forms where the rows are normalized (i.e. unitary)*; this makes the computations very easy since one has to take only the additive inverse of an element to reduce one row by another.

**Algorithm 1.1** Naïve Gaussian Elimination

```
r ← 0
for i = 1 to m do          /* for all the columns of A */
    piv_found ← false
    forj = r + 1 to n do /* search the next pivot, place it in the right position*/
        ifA[j, i] ≠ 0 then
            r ← r + 1
            A[j, ⋆] ↔ A[r, ⋆]
            A[r, ⋆] ← A[r, i]^{-1} × A[r, ⋆]        /* make row unitary */
            piv_found ← true
        end if
    if piv_found = true then
        /* eliminate all the rows below by the pivot r */
        for j = r + 1 to n do
            A[j, ⋆] ← A[j, ⋆] − A[j, i] × A[r, ⋆]
    end if
```

This algorithm is pretty straight forward: we start by sweeping the matrix searching for pivots from the left to the right; this is achieved throughout the outer for loop. Then we perform the row reduction in two steps:

1. Searching for a pivot at column index $i$: we check all the rows below the last found pivot row: of index $r$. The first row that has a leading coefficient in that column is chosen and placed in its final position (held in the $r$ variable). Notice that we can also choose other rows with leading coefficients in that column index. After that, the row is made unitary, by means of multiplication with the inverse of its leading coefficient; the variable $piv\_found$ is set to true indicating that the rows below can be reduced. The row at position $r$ is now called a pivot row.

2. The second step is performed only if the flag $piv\_found$ is set to $true$, which means that there is indeed a pivot at column index $i$ by which subsequent rows are to be reduced.

Reducing rows of indices greater than $r$ is performed by a simple [1]$axpy$ operation: replacing a row by the sum of that row and the multiplication of the additive inverse of the element at index $i$ by the pivot row at index $r$.

This algorithm leads indeed to a row echelon form using only elementary row operations.

From now on we consider the two operations:

- $make\_unitary(row)$: makes a row unitary by multiplying it by the inverse of its leading coefficient. This operation can be also referred to as $normalize$.

- $axpy(row_1, \alpha, row_2)$: reduces $row_1$by $row_2$, by replacing $row_1$by the sum of $row_1$and the multiplication of $row_2$with $\alpha$. We may sometimes omit $\alpha$, in this case, $\alpha$ is considered to be the additive inverse of the element of $row_1$which is located at the column index of the leading entry of $row_2$.

More specifically:

$$make\_unitary(row) = leading\_coefficient(row)^{-1} \times row$$

$axpy$: which is equivalent to $y \leftarrow \alpha \times x + y$ where $x$ and $y$ are vectors and $\alpha$ is a scalar. When $\alpha$ is omitted, its value is computed as following:

$$h = leading\_column\_index(row_2);\ \alpha = -(row_1[h]).$$

***Note***: In case $row_2$ is empty, $axpy$ and $normalize$ have no effect on their input rows.

[1] axpy: A X plus Y

**Gauss-Jordan Elimination**

To obtain *the reduced row echelon form* of a matrix, the Gauss-Jordan algorithm is used. Unlike the Gaussian-elimination which places zeros below the pivot elements working from the first row down to the last row, the Gauss-Jordan algorithm places zeros below and above pivot elements. It differs only slightly from the algorithm we have presented earlier in **??**.

**Algorithm 1.2** Gauss-Jordan Elimination

```
r ← 0
for i = 1 to m do          /* for all the columns of A */
    piv_found ← false
    forj = r + 1 to n do /* search the next pivot, place it in the right position*/
        ifA[j, i] ≠ 0 then
            r ← r + 1
            A[j, ⋆] ↔ A[r, ⋆]
            A[r, ⋆] ← A[r, i]^{-1} × A[r, ⋆]        /* make row unitary */
            piv_found ← true
        end if
    if piv_found = true then
        /* eliminate all the rows above and below by the pivot r */
        for j = 0 to n do
            A[j, ⋆] ← A[j, ⋆] − A[j, i] × A[r, ⋆]
    end if
```

Notice that the only difference from the Gaussian-Elimination algorithm is that we have only changed the bounds of the second inner for loop; not only reducing (placing zeros) the rows below the current pivot row, but going a step further and reducing the pivot rows above the current row too.

**Complexity** Both the Gaussian and the Gauss-Jordan elimination algorithms have a time complexity of order $O(n^3)$ for $n \times n$ full rank matrices.

These two algorithms are not efficient because at each time a pivot row is discovered, all the rows (except the current pivot) in the matrix are reduced by this pivot which implies performing writes all over the memory area where the matrix is stored. When working over the $\mathbb{Z}/p\mathbb{Z}$ field for example, the number of modular reductions becomes very limiting also. These algorithms are poorly designed to work over sparse matrices as well, as we will see in the following sections.

## 1.2 Structured Gaussian Elimination

In structured Gaussian elimination, we intend to minimize writing to the whole matrix while we reduce by a pivot row. We consider specifically sparse matrices for this algorithm, but it can also be perfectly applied to dense matrices as well with slight changes.

The idea of this algorithm is to reduce a row by all the pivot rows discovered so far. If at the end of this process, the current row is not empty, it is added to the list of pivot rows; a new row is then processed and the same steps are repeated till all the rows are handled.

At this level, the rank of the matrix is known since all the pivots have been identified.

One advantage of this method is that the writing area is limited to a small array that can fit into the processor's cache. While reducing a row, we use a dense array to represent it. On sparse matrices, the number of multiplications and additions while reducing the dense array by a pivot row is directly proportional to the number of the non-zero elements of that pivot.

The penalty of copying the current sparse row to a dense temporary array and back is negligible compared to the running time of the *axpy* operations. In the case of matrices over fields of a characteristic $p$ that can be represented over 16 bits, one can use in place operations over a dense array of

16 bits which implies performing modulo operations after each multiplication and addition. On the other hand, a dense array of 64 bits elements can be used to "accumulate" the results of additions and delay the modulo operations just before the dense array is copied back to the sparse row in the original matrix. Over 16 bits, one can perform up to $2^{31}$ additions in a 64 bit array accumulator before performing the modulo reduction. The number of modulo reduction falls back to $n^2$ because each row is reduced only once.

To obtain the reduced row echelon form, we need to reduce the newly discovered pivot rows by each other. The last step consists in sorting the rows by ascending column entry to obtain the final reduced row echelon form.

The above steps are described in the algorithm 1.3.

**Algorithm 1.3** Structured Gaussian Elimination

$rank \leftarrow 0$
/* loop over the rows of A, reduce rows by the newly discovered pivots */
for $i = 1$ to $n$ do
    copy_row_to_dense_array($A[i, \star]$, $temp$)
    for $j = 1$ to $i - 1$ do
        /* $temp = temp$–$temp[head(A[j, \star])] * A[j, \star]$ */
        axpy($temp$, $A[j, \star]$)
    copy_dense_array_to_row($temp$, $A[i, \star]$)
    if not_empty($A[i, \star]$) then
        $rank \leftarrow rank + 1$

/* reduce pivots by each other */
for $i = 1$ to $n$ do
    copy_row_to_dense_array($A[i, \star]$, $temp$)
    /* all pivots with index greater than $i$ and which have a column index entry greater than that of row $i$*/
    $S = j : j > i \wedge not\_empty(A[i, \star]) \wedge not\_empty(A[j, \star]) \wedge head(A[j, \star]) > head(A[i, \star])$
    /* note: $S$ is sorted incrementally */
    for each $j$ in $S$ do
        axpy($temp$, $A[j, \star]$)
    copy_dense_array_to_row($temp$, $A[i, \star]$)

sort_rows($A$)
return $A$

The *copy_row_to_dense_array* function copies a sparse/dense row of $A$ to a dense temporary array, this array can be, as we have already stated, of size 64 bits in the case of matrices with elements that can be represented with 16 bits.

*axpy*$(temp, row)$ performs the reduction of the dense array *temp* by *row* (cf. 1.1.2); if the pivot row at input to this function is empty, then the function has no effect. If the dense array *temp* is represented over 64 bits, *axpy* would only accumulate the products without performing the modulo reduction each time.

*copy_dense_array_to_row*$(temp, row)$ copies the elements of the dense array *temp* to the sparse row *row*. If the elements of *temp* are 64 bits, then a modulo operation is performed before the elements are stored back in *row*. *head*$(row)$ returns the column index of the leading element of *row*, it returns $-1$ if row is empty. *not_empty*$(row)$ returns true if row contains at least one non-zero element, false otherwise.

In the second for loop, a pivot $i$ is reduced only by pivots of greater indices (because it was already reduced by the rows of smaller indices in the first loop.) and whose leading element's column index is greater than the leading column index of the current row's leading element.

**Algorithm 1.4** Swap sorting

```
column_index_to_pivot_row_set ← Ø   /* a map of column index to row index */
permutations_set ← Ø   /* the permutations used to keep track of rows */
r ← 1
real_r ← 1
for i = 1 to n do
    column_index_to_pivot_row_set.add(head(A[i, ⋆]), i)

for each pair in column_index_to_pivot_row_set do
    row_index ← column_index_to_pivot_row_set[i]
    swap(A[permutations_set[row_index]], A[r, ⋆])
    /* A[r, ⋆] is in last position now */

    /* update the permutations */
    real_r ← permutations_set[r]
    while r ≠ permutations_set[real_r] do
        real_r ← permutations_set[real_r]
    t ← perm[row_index]
    permutations_set[row_index] ← r
    permutations_set[real_r] ← t
    r ← r + 1
```

Finally the function $sort\_rows(A)$ sorts the rows of the matrix according to their incrementing leading element's column index. We use an analyze phase to construct a map of each column index and its corresponding pivot row, and then an inplace swap between rows (useful with *C++ STL* vectors for example and on matrices where it is not possible to acquire a pointer on a row but it is possible to swap the contents of two rows efficiently without moving the actual data around.)

This method is shown in algorithm 1.4

The above algorithm performs sorting on the rows by incrementing leading element's column index using only swap operations between the rows. The set $column\_index\_to\_pivot\_row\_set$ is a map where the key is the column index of a row's leading element the corresponding row's index. This map is ordered incrementally on to the keys. The $permutations\_set$ is a map keeping track of the rows' positions. At the beginning of the algorithm, this map is simply the identity: pointing each row index $i$ to $i$ (at the beginning the row at position 5 is at position 5 indeed.)

During the actual sorting, a variable $r$ keeps track of the position where the next pivot row is to be inserted. The index of the next row to move to its final position is read: $row\_index = column\_index\_to\_pivot\_row\_set[i]$; then this row is swapped to its final position $r$. Notice that the actual swap is done between the row at position $r$ and the row at position $permutations\_set[row\_index]$; this is due to the fact that when $row\_index$ is read, the row pointed by this value could have already been swapped before: hence is at another position; this is where the $permutations\_set$ is useful: it is used to track where each row is located at throughout the algorithm.

The second part of the algorithm takes care of updating the $permutations\_set$ map. We indicate simply that the row located at $row\_index$ is now at the position $r$, however, the previous row that was at position $r$ is not necessarily the same at the map $permutation\_set$; in order to get the corresponding row in the $permutation\_set$, we have to perform a backtracking until we reach the position of original row, and update it to indicate that its corresponding row is now located at $row\_index$.

### 1.2.1 Implementation and experimental results

We have implemented the Structured Gaussian Elimination algorithm using the *LELA* library (cf. Chapter 2). We measure the running time of *LELA*'s naïve Gaussian Elimination implementation and

two structured Gaussian Elimination implementations: the first one performs modulo reductions after each multiply/add operation in the *axpy* routine ($O(n^3)$ modulo reductions); the second uses a 64 bits accumulator and performs the modulo reductions only once a row is fully reduced ($O(n^2)$modulo reductions).

The matrices used are sparse matrices issued from actual Gröbner bases problems over the $\mathbb{Z}/65521\mathbb{Z}$ field. The type $SparseMatrix$ is used to represent the matrices with elements presented with 16 bits (*unsigned short int*). The rows of the matrix – or vectors –, of type $SparseVector$, are internally represented with two vectors: one holding the actual values of the elements of the sparse vector; and the other keeping hold on the positions of these elements. The positions' vector is a vector of 32 bits elements since the column size of the some of the matrix exceeds easily 65535. Thus, using $LELA$'s $SparseMatrix$ type with these type parameters, each element requires 6 bytes of memory (2 bytes for the value and 4 bytes for the position).

The modulo operations are performed using the the *C++ "%"* operator and the inverse operations are done by $LELA$'s internal $Modular < T > .invert()$ routines.

The experiments were performed on an Intel(R) Core(TM) i7 CPU with a clock speed of 2.93GHz, 8MB of L3 cache and 8GB of RAM. All the running times are in seconds.

As shown in the table 1.1, the structured Gauss algorithm outperforms widely the naïve implementation (about 85 times faster). Also, using a 64 bit array accumulator to delay modulo operations increases significantly performance (~40% of running time).

Table 1.1: Structured Gaussian elimination vs naïve Gaussian elimination (seconds)

| matrix | dimensions | density | naïve Gauss (LELA) | Structured Gauss | Structured Gauss with accumulator |
|---|---|---|---|---|---|
| kat11/mat1 | 716 x 1218 | 7,81% | 0,79 | 0,05 | 0,04 |
| kat11/mat2 | 2297 x 3015 | 6,76% | 13,33 | 0,64 | 0,36 |
| kat11/mat3 | 4929 x 5703 | 6,15% | 84,32 | 3,25 | 1,86 |
| kat13/mat3 | 10014 x 14110 | 2,97% | **675,45** | **12,19** | **8,14** |
| kat13/mat4 | 19331 x 25143 | 2,69% | **3577,00** | **58,19** | **36,64** |
| kat13/mat5 | 28447 x 35546 | 2,66% | - | 156,85 | 91,68 |
| minrank minors 9_9_6/mat1 | 1296 x 11440 | 100,00% | - | 66,02 | 26,24 |
| minrank minors 9_9_6/mat2 | 5380 x 22400 | 45,29% | - | 220,68 | 92,57 |

***Note***: notice that if these matrices were represented as dense matrices, the memory required becomes very limiting. For example, in the Katsura 13 problem, the $4^{\text{th}}$ matrix, of size ($19331 \times 25143$), has a size of 78 MB when represented in a sparse form. If it was represented with a dense matrix, it would require no less than 927 MB which is almost 12 times the size of the sparse matrix.

In the table 1.2 we measure the time that is spent on each part of the structured Gauss algorithm: *copy_row_to_dense_array*, *axpy*, *copy_dense_array_to_row* and *sort_rows*. All the modulo operations are made the moment the dense array is copied back to the matrix, which makes this operation takes more time compared with the other operations.

It is clear that the copying of the rows to and from the dense temporary array are lighter operations compared to *axpy*. Furthermore, the running time of the sorting operation at the end of the algorithm is negligible compared to the other operations.

Table 1.2: Inner operations running time (seconds)

| matrix | copy row to dense array | copy dense array to row | axpy | sort rows | total |
|---|---|---|---|---|---|
| kat13/mat3 | 0,12 | 1,80 | 3,54 | 0,0002 | 8.14 |
| kat13/mat4 | 0,41 | 6,32 | 17,28 | 0,0004 | 36.64 |
| minrank minors 9_9_6/mat1 | 0,04 | 0,25 | 19,88 | 0,0000 | 26.24 |

Although the Structured Gauss algorithm achieves significant performance, it doesn't fully take advantage of the structure of the matrices issued from Gröbner bases computation. Indeed, in addition to the fact that matrices issued from Gröbner bases computation are generally very sparse with unitary rows and not necessarily full rank, they are also almost block triangular which makes most of the pivots known at the beginning of the computation.

In the following sections we present the *Faugère-Lachartre* algorithm which takes advantages of all these characteristics in order to perform efficient Gaussian elimination over such matrices.

## 1.3 The Faugère-Lachartre algorithm

In this section we will present the Faugère-Lachartre algorithm used to compute row echelon forms and reduced row echelon forms. The Faugère-Lachartre algorithm was designed specifically to be efficient on matrices generated by the *F4* [3] and *F5* [4] algorithms for Gröbner bases computations and which have very special properties:

- Sparse;
- Almost block triangular;
- Not necessarily full rank (this is the main difference between F4 and F5);
- All the rows are unitary (i.e. first element from the left is equal to 1);
- The number of columns is generally greater than the number of rows.

Unlike the structured Gaussian elimination algorithm, Faugère-Lachartre (we will use the acronym *FGL* from now on) takes advantage of the fact that at the beginning of the computation, a considerable number of pivots is already known; the efficiency of *FGL* relies mainly on this knowledge.

*FGL* separates the coefficients of the input matrices and performs computations only on areas where the computations are known ahead *not* to lead zeros elements. This is possible due to the fact that most of the pivots are known and that the pivot columns in the ***Rref*** form would eventually become the columns of the identity.

At the same time, this separation makes it possible to perform parallel computations on a big part of the *FGL* algorithm (2 parts of 3 were paralyzed in [2] and [1]). We will show in 2.2 how to fully parallelize the 3rd part, namely the structured Gaussian elimination.

### 1.3.1 Sequential Faugère-Lachartre algorithm

The *FGL* algorithm consists of 6 steps: 1) matrix analysis; 2) decomposition or splicing; 3) $Trsm$; 4) $Axpy$; 5) $Gauss$; and finaly 6) reconstruction of the final matrix. We will explain these steps in full details in the following sections.

***Definition***: a pivot column is a column in which a row has its leading element.

On a given matrix, two types of columns can be identified: *pivot columns* and *non-pivot columns*. In the reduced row echelon form, the pivot columns become the columns of the identity, and the non-pivot columns are expressed with respect to the basis of the pivot columns.

In the following sections, we denote as $M_0$ the input matrix to *FGL*, of size $n_0 \times m_0$.

#### 1.3.1.1 Analysis

The first step of the FGL algorithm is to identify the evident pivot columns and at the same time the corresponding pivot rows. Indeed, for a given pivot column, there might exist several rows that have the same leading index corresponding to the index of that column. One has the choice to use a pivoting strategy by which one row is chosen; the chosen row is called a pivot or a pivot row. For example, one can chose the first row encountered, the least/most dense row etc.

A simple one-pass sweeping of the rows' leading coefficients is enough to determine both lists. The list of the pivot columns indices is called $C_{piv}$ of size $N_{piv}$. The corresponding pivot rows indices is called $R_{piv}$ of size $N_{piv}$ too. The coordinates of the i$^{\text{th}}$ pivot are then $[R_{piv}[i], C_{piv}[i]]$. The list of the non-pivot columns (resp. the non-pivot rows) will be called $\overline{C_{piv}}$ (resp. $\overline{R_{piv}}$).

The algorithm 1.5 describes the analysis phase.

This algorithm is very lightweight if applied correctly on sparse data structures. For instance, if the head function runs in constant time (which is the case on sparse vectors), then the algorithm requires no more than $m_0$ iterations to finish the analysis step.

**Algorithm 1.5** Analysis of the matrix A of size $n \times m$

$N_{piv} \leftarrow 0$
for $i = 1$ to $m$ do
    $list_{condidates} \leftarrow \{j : head(A[j, \star]) = i\}$
    if $list_{condidates} \neq \emptyset$ then
        $piv \leftarrow$choose_condidate($list_{condidates}$)
        $N_{piv} \leftarrow N_{piv} + 1$
        $C_{piv}[N_{piv}] \leftarrow i$
        $R_{piv}[N_{piv}] \leftarrow piv$
    end if

return $C_{piv}$, $R_{piv}$, $N_{piv}$

#### 1.3.1.2 Matrix decomposition

Now that the pivot (resp. non-pivot) columns and rows are identified, the initial matrix $M_0$ can be decomposed into 4 sub-matrices $A$, $B$, $C$ and $D$ as follows:

- $A$ will hold elements from the pivot rows and columns only; these elements are indexed by the two lists $C_{piv}$ and $R_{piv}$. It will be upper triangular of size $N_{piv} \times N_{piv}$ and all of its diagonal elements are equal to 1.
- $B$ is composed of the elements of the pivot rows and the non-pivot columns: the elements indexed by $\overline{C_{piv}}$ and $R_{piv}$. B will be of dimensions $N_{piv} \times (m_0 - N_{piv})$.
- $C$ is the sub-matrix that will hold the elements of non-pivot rows and the pivot columns: these elements are indexed by $\overline{R_{piv}}$ and $C_{piv}$. Its rows are unitary and will have the dimensions $(n_0 - N_{piv}) \times N_{piv}$.
- $D$ which will contain the elements of the non-pivot rows and the non-pivot columns will have the dimensions $(n_0 - N_{piv}) \times (m_0 - N_{piv})$.

The decomposition in 4 sub-matrices is represented in the figure 1.3.1.

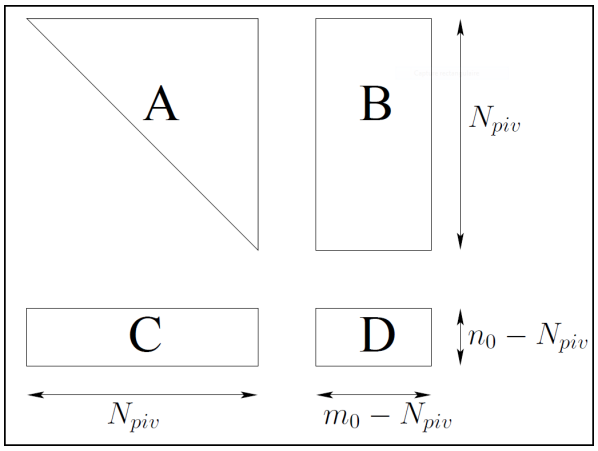


Figure 1.3.1: Decomposing into sub-matrices

This decomposition offers a twofold benefit: first, it separates the parts of the matrix according to nature of the operations to be performed on them; matrices $A$ and $C$ are only read throughout the algorithm, while we perform reads/writes on matrices $B$ and $D$. Assembling the elements in $B$ and

$D$ allows us also to make efficient computations on the vectors. Indeed, reductions can now operate on data elements that are packed contiguously in memory; this eliminates the jumps we would have if the elements of non-pivot columns were mixed with those of pivot columns. Second, we will see that this configuration is perfectly parallelizable since there are no dependencies between the columns of $B$ and $D$.

Figure 1.3.2 shows how the second matrix of the *Katsura 12* problem looks like after the splicing step. Black pixels represent non-zero elements, and white pixels represent zero elements.

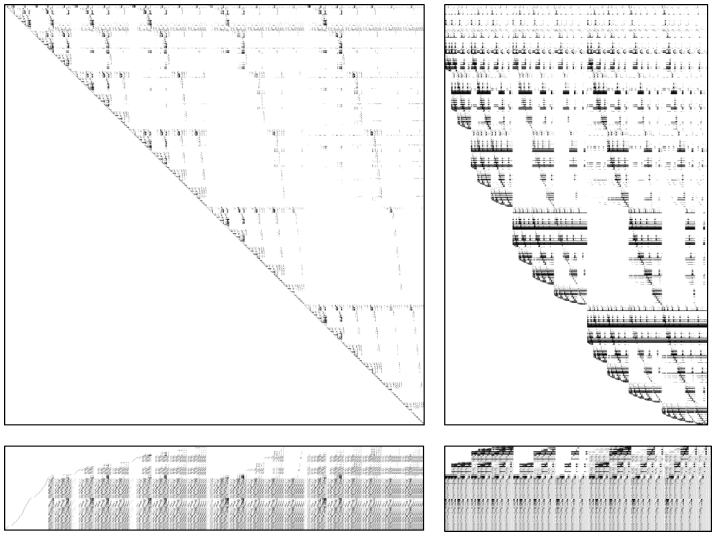

Figure 1.3.2: sub-matrices of Katsura 12/matrix 2

We can notice that the matrices $A$ and $C$ are sparser than $B$ and $D$. Furthermore, the number of pivot rows (rows of $A$ and $B$) is greater than the number of non-pivot rows.

The cost of this decomposition is of the number of non-zero elements of the original matrix; this cost is negligible compared to the overall complexity of the algorithm -we will see however that this can be penalizing on the block version.

After this decomposition, $M_0$ is now equivalent to:

$$M_0 \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

#### 1.3.1.3 Reducing the pivot rows by each other (Trsm[2])

In this step, the pivot rows are reduced by each other. From a Linear Algebra point of view, this is equivalent to computing $B \leftarrow A^{-1} \times B$; $A$ is invertible since it is upper triangular with 1s in the diagonal. Notice that the matrix $A$ is accessed in a read only fashion so it keeps its original density and remains sparse, while $B$ is accessed in a read/write mode; there is a considerable chance that $B$ will become denser throughout this step.

The algorithm to compute the pivot row reduction step is showed in the algorithm 1.6.

[2]Trsm: TRiangular Solve with Multiple right-hand sides

**Algorithm 1.6** Reducing the pivot rows by each other$A^{-1}B$

```
/* loop through rows of A starting from the end */
for i = N_piv − 1 downto 1 do
    /* loop through elements of A[i, ⋆] starting from the end */
    for j = N_piv downto head(A[i, ⋆]) + 1 do
        if A[i, j] ≠ 0 then
            /* B[i, ⋆] ← B[i, ⋆]–A[i, j] × B[j, ⋆] */
            axpy(B[i, ⋆], A[i, j], B[j, ⋆])
        end if
    end for

return B
```

In the outer loop, it is mandatory to start from the end of the matrix down to the beginning. When a row is reduced in $B$, the corresponding row in $A$ is implicitly reduced to the identity row (although we do not perform these operations.) Reducing a row at index $i$ means performing *axpy* operations with all the rows of indices greater than $i$; these rows have already become the rows of identity as the algorithm proceeds. Notice that we do not reach the leading coefficient in the inner loop because otherwise, each row would get reduced by itself.

After this step, $M_0$ is now equivalent to:

$$M_0 \begin{pmatrix} Id_{N_{piv}} & A^{-1} \times B \\ C & D \end{pmatrix}$$

#### 1.3.1.4 Reducing non-pivot rows by pivot rows (Axpy[3])

Once the pivots are reduced by each other, the next step consists in reducing the non-pivot rows by the new pivot rows: the rows of $C$ and $D$ by the rows of the new matrices $A$ and $B$. Once this step is achieved, the matrix $C$ becomes 0 since all its coefficients get reduced by those of $A$. As in the previous step, $C$ is accessed in a read only mode, its coefficients are used to perform the reductions on the rows of $D$ which is accessed in a read/write mode. As in the case of $B$, there is a high probability that the density of $D$ increases throughout this step. From a Linear Algebra point view, this step is equivalent to computing $D \leftarrow D - C \times (A^{-1} \times B)$.

The algorithm 1.7 shows the non-pivot rows reduction.

**Algorithm 1.7** Reducing non-pivot rows by pivot rows $D \leftarrow D - C \times B$

```
/* loop through rows of C */
for i = 1 to n_0 − N_piv do
    /* loop through elements of C[i, ⋆] */
    for j = 0 to N_piv do
        if C[i, j] ≠ 0 then
            /* D[i, ⋆] ← D[i, ⋆]–C[i, j] × B[j, ⋆] */
            axpy(D[i, ⋆], C[j, i], B[j, ⋆])
        end if
    end for

return D
```

An example of the reduction of the second row in the matrix $D$ (and $C$) is presented in the figure

[3] Axpy : A X plus Y

1.3.3. Notice that since we started the reduction from the first row of $C$, the first row is now equal to 0; as we stated before, we do not actually perform this computation on the rows of $C$.

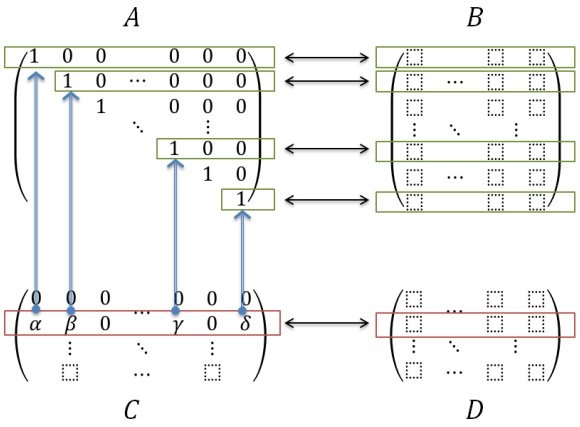


Figure 1.3.3: An example of the non-pivot rows reduction

The original matrix $M_0$ is now equivalent to (*w.r.t* the original sub-matrices $A$, $B$, $C$ and $D$):

$$M_0 \begin{pmatrix} Id_{N_{piv}} & A^{-1} \times B \\ 0 & D - C \times (A^{-1} \times B) \end{pmatrix}$$

#### 1.3.1.5 Identification of new pivots (Gauss)

Now that the entire matrix has been reduced by the pivot rows (though, not the same pivot rows), new pivots must be identified among the non-pivot rows and columns: the matrix $D$. For this purpose, we use a slightly modified version of the structured Gaussian elimination algorithm we have presented in 1.2. We do not compute the reduced row echelon form of the matrix $D$, but only an echelon form. Notice that the rows of $D$ are not unitary and field inversions are thus required.

**Algorithm 1.8** Compute a pseudo-row echelon for of matrix $D$ of size $(n_0 - N_{piv}) \times (m_0 - N_{piv})$

```
rank_D ← 0
/* loop over the rows of D, reduce rows on the go */
for i = 1 to (n_0 − N_piv) do
    normalize(D[i, ⋆])
    copy_row_to_dense_array(D[i, ⋆], temp)
    for j = 1 to rank_D do
        if head(temp) = head(D[j, ⋆]) then
            axpy(temp, D[j, ⋆])

    copy_dense_array_to_row(temp, D[i, ⋆])
    normalize(D[i, ⋆])
    if not_empty(D[i, ⋆]) then
        rank_D ← rank_D + 1
        sorted_insertion(i)
return rank_D
```

The $rank_D$ variable holds the rank of the matrix $D$ at the end of the algorithm.

*normalize* makes its input row a unitary row by dividing the entire row by the multiplicative inverse of its leading coefficient. The *copy_row_to_dense_array* function copies a sparse/dense row of the matrix to a dense temporary dense array. *copy_dense_array_to_row*(*temp*, *row*) copies the elements of the dense array *temp* to *row*. *axpy*(*temp*, *row*) performs the reduction of the dense array temp by row; if the pivot row at input to this function is empty, then the function has no effect. *not_empty*(*row*) returns true if row contains at least one non-zero element, false otherwise.

In this algorithm, a sorting the rows is needed each time a row is fully reduced, the subsequent rows are reduced only by pivots of the same leading column index.

After this step, the original matrix $M_0$ is now equivalent to (*w.r.t* the original sub-matrices $A$, $B$, $C$ and $D$):

$$M_0 \begin{pmatrix} Id_{N_{piv}} & A^{-1} \times B \\ 0 & Gauss(D - C \times (A^{-1} \times B)) \end{pmatrix}$$

***Note***: Upon this point, all the rows are pivot unitary rows. The rank of the original matrix $M_0$ is henceforth known and equal to $N_{piv} + rank_D$. An echelon form of the matrix $M_0$ can be reconstructed at this point; refer to the section 1.3.1.7 for an idea about how this can be done.

#### 1.3.1.6 Second iteration

In order to obtain the reduced row echelon form, a second iteration of the algorithm must be applied. During this second iteration the newly discovered pivot rows in the matrix $D$ are reduced by each other; this corresponds to the $Trsm$ step we have presented earlier. Furthermore, the rows in the new matrix $B$ (which were part of the original pivots discovered in the matrix $M_0$) must now be reduced by the newly discovered pivots in the matrix $D$; this step is the $Axpy$ step we have presented before.

We start by performing the analysis step on the matrix $D$; we now have lists $C_{piv_D}$ and $R_{piv_D}$ corresponding the column and row indices of the pivots in $D$. Notice that $D$ contains only pivot rows or empty rows: because we have reduced it to an *echelon form* in the previous step.

Now we proceed to the splicing step, in which the matrix $D$ is spliced to two matrices $D1$ and $D2$. $D1$ is the matrix containing the elements from the pivot columns and the pivot rows of $D$; these elements are located by the tuple $[R_{piv_D}[i], C_{piv_D}[i]]$. $D2$ contains the elements issued from the pivot rows and the non-pivot columns of $D$. Likewise, $B$ is decomposed according to the list $C_{piv_D}$ to $B1$ and $B2$. $B1$ contains the elements of $B$ whose column indices are in $C_{piv_D}$ and $B2$ contains the elements of $B$ whose columns are not in $C_{piv_D}$.

We can now apply the same reductions $Trsm$ and $Axpy$ that we have presented earlier to these new matrices. This whole process corresponds to the reduction of the pivot rows by each other.

#### 1.3.1.7 Matrix reconstruction

At this point it is possible to reconstruct the reduced row echelon form of the original matrix $M_0$ (or a row echelon form in case we have stopped in the $Gauss(D)$ step). The final columns of the ***Rref*** of the matrix $M_0$ are now in the sub-matrices $B2$ and $D2$ in addition to the identity columns of $A$ and $D1$. The algorithm 1.9 shows the process of reconstructing the reduced echelon form from the sub-matrices $B2$ and $D2$ ($C$ is now equal to 0 and $A$ to $Id_{N_{piv}}$).

**Algorithm 1.9** Reduced echelon form reconstruction

// $C_{piv_D}, R_{piv_D} \leftarrow Analyse(D)$
$piv\_col\_to\_row \leftarrow \emptyset$
/* update the map $piv\_col\_to\_row$ to contain pivot rows from the matrix $A$ and the new pivots of $D1$ */
for $i = 1$ to $N_{piv}$
    /* in $A$, the next pivot is at row $i$ and is originated from column $C_{piv}[i]$ in $M_0$ */
    $piv\_col\_to\_row[C_{piv}[i]] \leftarrow i$

/* Add the new pivots in $D1$ */
for $i = 1$ to $rank_D$ do
    /* get where in $M_0$, the column $i$ in $B$ (and $D$) has originated */
    $idx\_in\_B = C_{piv_D}[i]$
    $idx\_in\_A = \overline{C_{piv}[idx\_in\_B]}$
    $piv\_col\_to\_row[idx\_in\_A] = N_{piv} + i$

/* remap the non-pivot columns from $B2$, $D2$ to the original matrix $M_0$ */
for $i = 0$ to $\#\overline{C_{piv}}$ do
    $idx\_in\_B = \overline{C_{piv}[i]}$
    $idx\_in\_B2 = \overline{C_{piv_D}[idx\_in\_B]}$
    $\overline{C_{piv}[idx\_in\_B]} = idx\_in\_B2$  /* points the non-pivot columns of $M_0$ to the final column in $B2$ */

/* reconstruct the actual matrix */
$M_{res} \leftarrow 0_{(n,m)}$
$next_{pivot} \leftarrow 1$
$L \leftarrow (\frac{B2}{D2})$  /* $L$ is the concatenation of the rows of $B2$ and $D2$ in this order */

for each *pair* $p$ in $piv\_col\_to\_row$ do
    /* $p$ is a tuple of column index $\rightarrow$ row index */
    $M_{res}[next_{pivot}][p.first] \leftarrow 1$  /* adds identity column */

/* add the actual elements of the rows of $B2$ and $D2$ */
for $i = 1$ to $(m_0 - N_{piv} - N_{piv_D})$ do
    $M_{res}[next_{pivot}][\overline{C_{piv}[i]})] \leftarrow L[p.second][i]$

return $M_{res}$

These lists of pivot columns and rows in the matrix $D$ are denoted $C_{piv_D}$, $R_{piv_D}$ respectively as explained in the analysis section. Then a new map $piv\_col\_to\_row$ is created to map the positions of the original columns in the matrix $M_0$ to the corresponding pivot rows in matrices $B2$ and $D2$ which are concatenated to form a bigger matrix $L$ of size $n_0 \times (m_0 - N_{piv} - N_{piv_D})$.

The non-pivot columns list is updated so that it points to the column index in the matrix $B2$ rather than that of $B$; indeed, the new elements of the ***Rref*** form are in the matrices $B2$ and $D2$ and not in $B$ and $D$ anymore. The number of writes performed is proportional to the number of non-zero elements in the matrices $B2$ and $D2$.

The following schema in figure 1.3.4 shows the steps of the algorithm. On the second iteration, the matrices $A$, $B$, $C$ and $D$ are replaced by the matrices $D1$, $D2$, $B1$ and $B2$ respectively. At the end of the second iteration, the reduced row echelon form can be reconstructed.

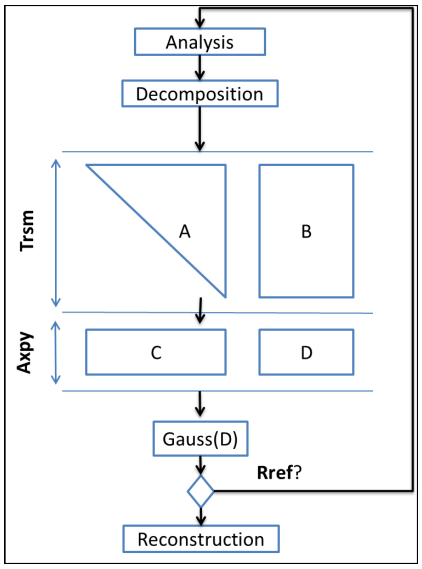


Figure 1.3.4: Steps of the Faugère-Lachartre algorithm

The figure 1.3.5 shows how the reduced row echelon form of the matrix *katsura12 mat2* looks like.

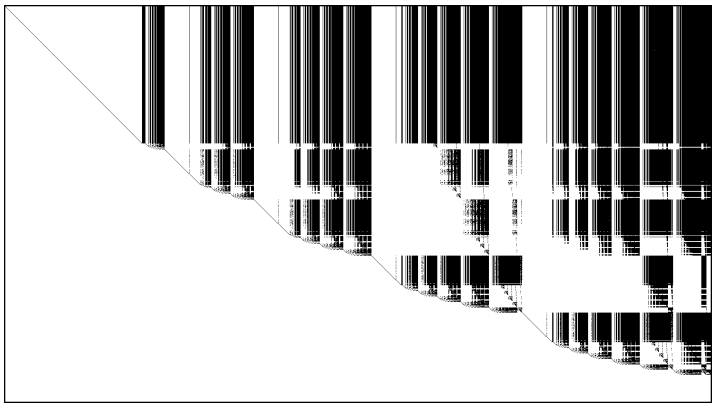

Figure 1.3.5: Reduced row echelon form of katsura 12 matrix 2

## 1.4 Block version of Faugère-Lachartre

Having a block version of the *FGL* algorithm has multiple benefits: taking advantage of modern processors architecture by packing data into blocks that can fit into the processor's cache memory, and maintaining an optimal stream of data. Also, the operations on the $\overline{C_{piv}}$ columns are independent which means that these columns can be handled in parallel provided the data in these columns are separated; this leads naturally to a block representation which separates the elements of the matrix on a columns basis.

The data structures of this version are well presented in [2] section 4. We mention here only that the blocks are a list of rows, where every row may have two formats: a "sparse format," or a "hybrid format." A row in a sparse format is a list of position-value tuples; whereas a hybrid row is simply a dense array containing the elements (zero and non-zero elements) of that row. There is a ***threshold*** that is used to define if a row should be in a sparse or hybrid format. For example, if the threshold

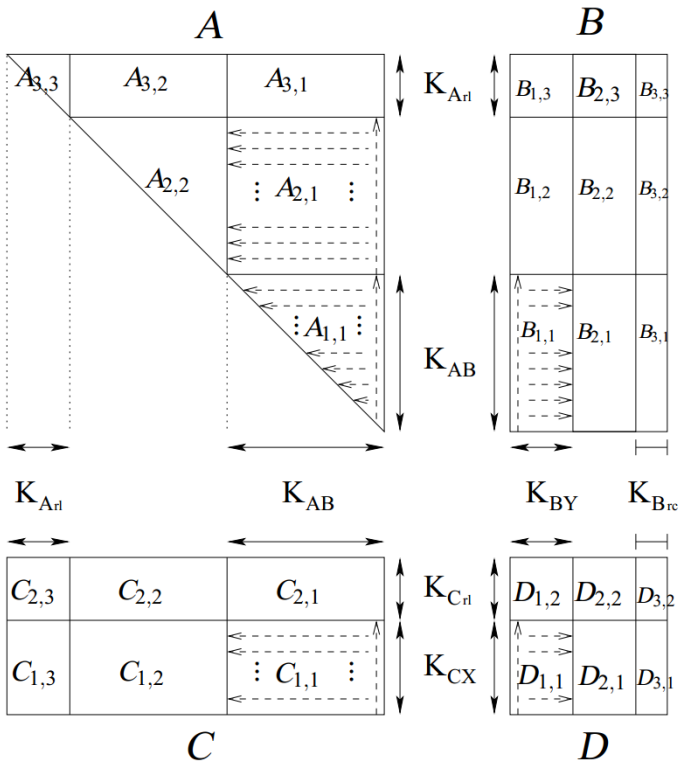


Figure 1.4.1: Block representation of the sub-matrices A, B, C and D

is 50%, then a row that has more elements than 50% of the matrix column size is stored in a dense format.

Blocks of the matrix $A$ and $C$ are represented from right to left then from down to top, while those of $B$ and $D$ are from down to top then from left to right as shown in the figure 1.4.1. Elements inside the blocks themselves follow a similar configuration: from right to left the from down to top for the blocks of $A$ and $C$, and from left to right then from down to top for the blocks of $B$ and $D$.

Performing the $Trsm$ and $Axpy$ operations on block matrices is achieved throughout two levels: matrix level operations operating on blocks (outer block operations), and block level operations that operate on the elements inside blocks (inner block operations).

### 1.4.1 Inner block operations

Inner block operations are of two types: reducing a block by another block ($Axpy_{block}$), or reducing the block by itself ($Trsm_{block}$). These are exactly the same operation we have already presented in 1.3.1.3 and 1.3.1.4. In fact, the blocks themselves can be considered as small matrices; a reduction between two different blocks is analogous to the $Axpy$ operation performed between the sub-matrices $A$, $B$, $C$ and $D$. The only difference is that one has to take into account now the nature of the rows being reduced: sparse or hybrid, and the disposition of these rows inside a block.

The algorithm 1.10 is a modified version of the $Trsm$ algorithm 1.6 that acts on the block scale with hybrid rows.

**Algorithm 1.11** Block $Trsm$ $(A^{-1}B)$

```
/* loop through the columns of blocks of B */
for i = 1 to nb_block_columns_B do
    /* loop through the rows of blocks of A */
    for j = 1 to nb_block_rows_A do
        /* reduce block B[i,j] by all the blocks before in the same column */
        for k = 1 to j − 1 do
            /*B is accessed in column major order*/
            Axpy_block(A[j,k], B[i,k], B[i,j])

    /* reduce B[i,j] by itself */
        Trsm_block(A[j,j], B[i,j])

return B
```

**Algorithm 1.10** $Trsm_{block}$: reduce $Block_B$ by upper tringular $Block_A$

```
for i = 2 to Block_height(Block_A) do
    /* rows are listed from down to top */
    temp ←hybrid_to_dense(Block_B[i,⋆])
    for j = 1 to head(block_A[i,⋆])+1 do
        /* elements liste from right to left in block_A */
        if A[i,j] ≠ 0 then
            /* temp ← temp–Block_A[i,j] × Block_B[j,⋆] */
            if density(Block_B[j,⋆])< threshold then
                sparse_axpy(temp, Block_A[i,j], block_B[j,⋆])
            else
                dense_axpy(temp, Block_A[i,j], block_B[j,⋆])
    Block_B[i,⋆] ←dense_to_hybrid(temp)

return Block_B
```

The function *hybrid_to_dense* copies a hybrid row to a dense array, whereas *dense_to_hybrid* performs the opposite operation: copies a dense array back to a hybrid row with a sparse representation if the density of temp is less than the threshold. *sparse_axpy* and *dense_axpy* perform rows reduction in a sparse or a dense mode. Some optimizations can be taken advantage of in the case of *dense_axpy* since the data is contiguous and no index computations or jumps are needed throughout the operation. Likewise, the $Axpy_{block}$ operation differs from the original $Axpy$ by the hybrid aspect of the rows; the way the rows are organized inside a block must be taken into account also.

### 1.4.2 Outer block operations

The organization of the blocks inside the sub-matrices $A$, $B$, $C$ and $D$ allows us to perform the $Trsm$ and $Axpy$ operations on these matrices in a very intuitive manner. The algorithm 1.11 shows the $Trsm$ operation considering only block operations over the sub-matrices $A$ and $B$.

The $Axpy$ operation considering only block operations is shown in the algorithm 1.12.

As in the case of rows reduction, in both algorithms, a dense block can be used during the reduction by other blocks and then written back to matrix once fully reduced.

**Algorithm 1.12** Block *Axpy* $(D - CB)$

/* loop through the columns of blocks of $D$ */
for $i = 1$ to $nb\_block\_columns\_D$ do
    /* loop through the rows of blocks of $C$ */
    for $j = 1$ to $nb\_block\_rows\_C$ do
        /* reduce block $D[i, j]$ by all the blocks in the column $i$ of $B$ */
        for $k = 1$ to $nb\_block\_columns\_C$ do
            $Axpy_{block}(C[j, k],\ B[i, k],\ D[i, j])$

return D

### 1.4.3 Block hybrid Gaussian elimination

The authors in [2, 1] have adapted the Gaussian elimination algorithm over the matrix $D$ that we have presented in 1.3.1.5 to the block representation of the matrices.

The idea is to consider the matrix $D$ as a matrix with one row of blocks and as many columns as the original one: namely $\lceil \#\overline{C_{piv}}/block\_width\_D \rceil$. A dense matrix $P$ of size $\#\overline{R_{piv}} \times \#\overline{R_{piv}}$ and initially equivalent to the *identity* is used to keep track of the operations performed on the blocks as the algorithm proceeds. A sequential Gaussian elimination method is then used over the blocks one by one starting from the left to the right. The matrix $P$ is used to keep track of the different reductions performed on the blocks $D_i$; then, when reducing the block $D_{i+1}$, we first update it by a left multiply with the matrix $P$ in order to *propagate* the *pending operations* from the previous blocks; $P$ is then concatenated again to the block $D_{i+1}$ and the Gaussian elimination is once again performed on this block one and so on.

Check [1] for a full description of this method.

## 1.5 Parallelization

The $Trsm$ $(A^{-1} \times B)$ and $Axpy$ $(D - C \times (A^{-1} \times B))$ operations can be made parallel since there is no data dependency between the columns of the matrices $B$ and $D$. The block organization of the matrices makes this perfectly adequate to parallelize. We denote the columns of blocks of $B$ as $B_i$ and those of $D$ as $D_i$ then the basic parallelizable operations are:

- $Trsm_i$: takes as input a column of blocks $B_i$, the matrix $A$ and returns as output $A^{-1} \times B_i$.
- $Axpy_i$: computes $D_i - C \times B_i$ given the matrix $C$ and the columns of blocks $B_i$ and $D_i$.

Clearly there is an ordering on these operations; the $Trsm_i$ (resp. $Axpy_i$) operations are completely independent one another for different $i$; however, for a given column block index $i$, the $Trsm_i$ operation must precede $Axpy_i$ because $Axpy_i$ uses as input the output of $Trsm_i$.

For the Gauss operation on the matrix $D$, a dense matrix $P$ is used to keep track of the operations performed on the blocks as they get reduced. Before each reduction, the operations are propagated on the current block using the pseudo inverse matrix $P$ then the block concatenated to $P$ is reduced. The same sequential block hybrid Gaussian elimination presented in 1.4.3 is used.

Priority rules are defined between the operations $Trsm_i$, $Axpy_i$ and $Gauss_i$ along with synchronization points used enforce the priority rules. The synchronization points are:

- $S1$(from Analysis and decomposition to $Trsm$): the analysis and decomposition steps must be finished before $Trsm$ can start;
- $S2$(from $Trsm$ to $Axpy$): to compute $Axpy_i$, the computation of $Trsm_i$ must be completed;
- $S3$(from $Axpy$ to $Gauss$): to apply the reduction $Gauss_i$, $Axpy_i$ must be completed as well as the operation $Gauss_j$ for all $j$ between 1 and $i - 1$;

- $S4$(from the$Gauss$ step to the reconstruction step): all the operations of type $Gauss$ must be completed before the final matrix can be reconstructed.

For the full version of the parallel algorithm, refer to [2].

***Note***: In [2, 1], the Gauss step is sequential, the operations $Gauss_i$ are interposed with $Trsm_i$ and $Axpy_i$ but they are sequential nonetheless.

# Chapter 2

# Contributions

We present in this chapter a new parallel algorithm for the structured Gaussian elimination method and show some experimental results about its efficiency in 2.2. In section 2.3 we present a new ordering of operations of Faugèure-Lachartre and we show that this method is generally more efficient than the standard algorithm when computing row echelon forms on full rank matrices.

## 2.1 Modifications to the Standard Faugère-Lachartre Method

Throughout our implementation we have used the multiline data structure that we will present in chapter 3. The multiline data structure is very efficient but not suited when rows' ordering is involved; this is the case of the Gauss step in the Faugère-Lachartre algorithm. To this end, we propose a modification to this algorithm that does not require the reordering of the rows.

### 2.1.1 Identification of new pivots (Gauss)

We use a slightly modified version of the structured Gaussian elimination algorithm we have presented in 1.2 to compute an echelon form of the matrix $D$ in order to identify the new pivots. Notice that the rows of $D$ are not unitary and field inversions are thus required.

**Algorithm 2.1** Compute a pseudo-row echelon for of matrix $D$ of size $n \times m$

$rank_D \leftarrow 0$
/* loop over the rows of $D$, reduce rows on the go */
for $i = 1$ to $n$ do
    normalize($D[i, \star]$)
    copy_row_to_dense_array($D[i, \star]$, *temp*)
    for $j = 1$ to $i - 1$ do
        axpy(*temp*, $D[j, \star]$)

    copy_dense_array_to_row(*temp*, $D[i, \star]$)
    normalize($D[i, \star]$)
    if not_empty($D[i, \star]$) then
        $rank_D \leftarrow rank_D + 1$

return $rank_D$

We do not sort the rows to obtain a correct echelon form because this sorting step will be performed during the next step of the algorithm involving analysis. The $rank_D$ variable holds the rank of the

matrix $D$ at the end of the algorithm.

***Note***: In [1], the author performs this Gaussian elimination only over rows which have the same leading coefficient index as a pivot row. This has indeed an advantage if only the echelon form of the original matrix is required. In the case where a reduced echelon form is required, the computations that we perform when reducing by all the pivots in the algorithm 2.1 are simply subtracted from the overall work to be performed in the next steps.

### 2.1.2 Block hybrid Gaussian elimination

We have dropped using this Gaussian elimination over hybrid blocks (c.f. 1.4.3) and chosen to copy the block matrix back to a row representation and apply our classical structured Gaussian elimination presented in the previous section. Our choice is motivated by the following arguments:

- The fact that the matrix $P$ requires more space $(n_0 - N_{piv})^2$ of the size of the elements.
- Our special data structure (the multiline vector that we will present in chapter 3) which makes ordering individual rows very penalizing.
- This algorithm is not parallelizable; we show in 2.2 how our structured Gaussian elimination over rows can be parallelized.

We should also notice that the copying time compared to the overall structured Gaussian elimination computation is generally negligible.

## 2.2 Parallel Structured Gaussian elimination

In this section we present a parallel algorithm of the sequential structured Gaussian elimination we have presented in 1.2. In the original Faugère-Lachartre parallel algorithm, the two steps $Trsm$ and $Axpy$ are parallelized but not the $Gauss$ step. To this end, we have developed a parallel version of the structured Gaussian elimination. This algorithm is designed to work over shared memory architectures.

The idea behind this algorithm is that in the structured Gaussian elimination, a row $i$ is replaced by a linear relation in which the rows of index $j$ (where $j \leq i$) are involved. This can be expressed as follows:

$$row_i \leftarrow row_i + \sum_{j=1}^{i-1} \alpha_j row_j$$

At step $i$ of the algorithm, the rows from 1 to $i-1$ have already been reduced and are in their final image *w.r.t.* the row echelon form we are computing.

If we dispose of $p$ processors (or threads) then at step $i$, the rows from $i$ to $i+p$ can, in a large part, be reduced in parallel by pivots up to $i-1$ since these pivots are only read and are already in their final form. Once all the rows between $i$ and $i+p$ have been reduced by pivots of indices 1 to $i-1$, the row $i + offset$ must wait for all the rows from $i$ to $i + offset - 1$ in order be fully reduced. We solve this problem by having a priority queue, $waiting_q$ , that holds the rows which have been reduced by the pivots $\{0..i-1\}$ but are waiting for other pivots to be fully reduced.

All the threads have a shared variable indicating the last fully reduced pivot: *last_pivot*; any row can be reduced by the pivots from 1 to *last_pivot*. The threads then spin over all the rows, fetching the next available row to reduce and reducing it up to *last_pivot*. If the current row has an index equal to *last_pivot*$+1$, it means that the current row is fully reduced and hence it is added to the list of fully reduced rows; otherwise, it is added to $waiting_q$. The other threads can now fetch elements from the $waiting_q$ and finish their reduction up to *last_pivot* which should have changed its value since the last time the current row was added to $waiting_q$.

These steps are described in the algorithm 2.2.

We start by reducing several rows sequentially to avoid high contention on the waiting queue. This is done by a call to *echelonizeRowsUpTo_Sequential* which is a simple call to the sequential structured Gaussian elimination. Then several shared variables between all the threads are declared:

**Algorithm 2.2** Parallel structured Gaussian elimination of a matrix $A_{n,m}$

```
/* echelonize several rows at the beginning */
echelonizeRowsUpTo_Sequential(A, 0, p);

shared:
   last_pivot    /* index of the last fully reduced pivot */
   next_row_to_reduce    /* index of the next available row to reduce in the matrix*/
   waiting_q    /* min-priority heap: list of not fully reduced rows */

private:
   local_last_pivot    /* rows are reduced up to this index */
   row_to_reduce    /* candidate row to reduce */
   start_from    /* pivot from which to start the reduction */

while (true) do in parallel
      if last_pivot = n then    /* termination condition */
            break

local_last_pivot ← last_pivot    /* no synchronization needed */
      lock(mutex)    /* start of critical section */
      if not_empty(waiting_q) then
            /* fetch smallest row index from waiting list */
            elt ← waiting_q.fetch_smallest()
            row_to_reduce ← elt.row
            start_from ← elt.last_pivot_reduced_by + 1
      else        /* fetch next available row in the matrix */
            row_to_reduce ← next_row_to_reduce
            next_row_to_reduced ← new_row_to_reduce + 1
            start_from ← 0
      unlock(mutex)        /* end of critical section */

copy_row_to_dense_array(A[row_to_reduce], temp)
      for i = from_pivot to local_last_pivot do
            /* temp = temp–temp[head(A[i, ⋆])]^{-1} × A[i] */
            axpy(temp, A[i]);
      copy_dense_array_to_row(temp, A[row_to_reduce])

lock(lutex)
      if row_to_reduce = local_last_pivot + 1 then    /* row_to_reduce fully reduced */
            last_pivot ← last_pivot + 1
      else
           /* adds this row to the waiting list specifying the row index and the last pivot it was reduced by */
            waiting_q.add({row_to_reduce, last_local_pivot})
      unlock(mutex)
end wile
sort_rows(A)
```

- *last_pivot*: this variable keeps track of the index of the last fully reduced pivot.
- *next_row_to_reduce*: the next not reduced row in the original matrix. This is incremented atomically each time a row is fetched (goes from 1 to $n$).
- $waiting_q$: a minimum priority heap that holds track of the rows on which we have started reduction but are not fully reduced yet. These are rows of indices $[last\_pivot+2..next\_row\_to\_reduce[$.

Each thread has a list of private variables also:

- *local_last_pivot*: holds the local value of the shared variable *last_pivot*.
- *row_to_reduce*: this variable can be assigned the shared variable *next_row_to_reduce* if $waiting_q$ is empty, otherwise, it holds the index of the smallest waiting row in $waiting_q$.
- *start_from*: indicates the index of the pivot from which to start the reduction. If the row to be reduced is fetched from the matrix $A$, then this variable is assigned the value 1 (indicating to start the reduction from the first row); however, if the row is fetched from the waiting list, then this variable indicates that the reduction should start from the last pivot index that the waiting row was reduced by.

The threads can now fetch rows to reduce from the matrix or the waiting queue. Each thread starts by checking if there are other rows not fully reduced, indicated by the condition $last\_pivot < n$; this is also the termination criteria for each thread.

If the termination condition is not met, the thread fetches the index of the next row to handle. In case the waiting queue is not empty, then the thread "helps out" by finishing the reduction of the waiting rows. The function $waiting_q.fetch\_smallest()$ pops an element from $waiting_q$ (removing it from the queue); the fetched element contains the index of the row to reduce and the index of the last pivot it was reduced by. If, on the other hand, $waiting_q$ is empty, the index of the row to reduce is fetched from the shared variable *next_row_to_reduce* and the *start_from* is set to 1 indicating to start reducing this row by all the pivots up to *last_pivot*. The shared variable *next_row_to_reduce* is then incremented so that the other threads can pick a new row from the matrix.

All the above operations must be synchronized since they handle shared resources: $waiting_q$, *next_row_to_reduce*. The functions *lock* and *unlock* are used to achieve this synchronization over a shared *mutex* passed as input.

Once the index of the row to reduce and the interval of the pivots to be used in the reduction are known, the actual reduction can be now performed. We first start by copying the row $A[row\_to\_reduce]$ to a temporary dense array *temp*. Then we perform *axpy* operations over temp and all the rows in the interval $[start\_from, local\_last\_pivot]$. *temp* is then copied back to the row $A[row\_to\_reduce]$.

Notice that there is no need of any synchronization during this step. Indeed, any pivot with an index less that *last_pivot* is accessed in a read only mode and hence can be used by all the threads during the *axpy* computations. This has also an advantage since the pivots can be kept in the processors' cache for faster access.

Furthermore, the rows pointed by the private variable *row_to_reduce* are also ensured to be different for each thread since the access to $waiting_q$ and *next_row_to_reduce* is synchronized; each time a thread extracts the next row to reduce, it whether removes it from the $waiting_q$ or increments the *next_row_to_reduce* variable: this ensures that at any given time, no thread is writing to a row used by another thread.

Once a row is reduced and written back to the matrix, it can now be in two states: fully reduced or not fully reduced. A row $i$ is fully reduced only if it was reduced by pivots up to $i-1$; to this end, we check if *row_to_reduce* is equal to $local\_last\_pivot + 1$, if it is the case, then we increment the value of *last_pivot* to indicate that reduction can now include the current row as a pivot. However, if the index of the row is not equal to $local\_last\_pivot + 1$, then we add its index and the index of

the last pivot it was reduced by to the waiting queue. These operations must also be synchronized to ensure the integrity of the shared variables.

Sorting the rows according the leading entries is then required to obtain the row echelon form of the matrix. As we have shown in 1.4, this is negligible compared to the total running time of the overall algorithm.

#### Note on load balancing

The actual algorithm as presented can entail a very severe unequal load balancing between the threads. Indeed, a thread can get "stuck" reducing rows in the $waiting_q$ while the other threads advance over the rows of the matrix and pile them up in the $waiting_q$ . To avoid such situation, we add flags to indicate when a thread should check the waiting list and when it can reduce rows from the matrix. For example, a thread that reduced a row completely is set to check the waiting list immediately since it has just added a new pivot and other rows could be waiting for this pivot. We also impose that a thread which adds $t$ consecutive times to the $waiting_q$ should help with the waiting rows rather than fetching new rows from the matrix.

#### Note on synchronization

Ideally, more fine-grained synchronization should be used to minimize waiting times. The $waiting_q$ would be implemented as an independent object assuring synchronization over its *add*, $fetch\_smallest$ and *not_empty* methods using *spin locks* preferably; these operations take very short time to execute and spin locks can ensure that the threads would simply spin waiting for the lock to be released instead of enduring a context switch had *Mutexes* been used. The same goes for the other synchronization parts of the algorithm.

### 2.2.1 Experimental results

We have implemented this algorithm in our parallel Faugère-Lachartre version. We used the *multi-line* data structure to represent the rows of the matrices. Notice that at the end of the *Axpy* step, the matrix $D$ becomes very dense; in the following experiment, the matrix $D$ of the problem *minrank_minors_9_9_6* becomes almost 100% dense after the *Axpy* step. This means that most of the threads' running time is spent on the *axpy* operations and hence less thread communication/waiting is entailed. On sparse matrices, communication time could decrease greatly the performance of this algorithm.

The speedups achieved with different number of threads are showed in table 2.1 and the corresponding efficiency in table 2.2.

***Note***: $speedup_p = \frac{Sequential\,running\,time}{Parallel\,running\,time_p}$ ,$efficiency_p = \frac{speedup_p}{p}$ where $p$ is the number of threads.

The experiments were performed on an Intel(R) Xeon(R) X5677 CPU with a clock speed of 3.47GHz, 12MB of L3 cache and 144GB of RAM.

Table 2.1: Speedup of parallel structured Gaussian elimination

| matrix | 2 threads | 4 threads | 8 threads | 12 threads | 16 threads |
|---|---|---|---|---|---|
| minrank minors 9_9_6/mat1 | 1.90 | 3.69 | 6.80 | 5.62 | 5.88 |
| mr 9_9_6/mat2 | 1.85 | 3.59 | 5.68 | 4.68 | 4.55 |
| mr 9_9_6/mat3 | 1.87 | 3.38 | 5.70 | 4.95 | 5.19 |
| mr 9_9_6/mat4 | 1.89 | 3.70 | 5.57 | 5.03 | 5.28 |
| mr 9_9_6/mat5 | 1.90 | 3.69 | 5.55 | 4.99 | 5.36 |
| mr 9_9_6/mat6 | 1.88 | 3.86 | 5.83 | 5.23 | 5.66 |
| mr 9_9_6/mat7 | 1.88 | 3.72 | 5.57 | 4.97 | 5.40 |
| mr 9_9_6/mat8 | 1.91 | 3.63 | 5.30 | 4.58 | 4.46 |

Table 2.2: Efficiency of parallel structured Gaussian elimination

| matrix | 2 threads | 4 threads | 8 threads | 12 threads | 16 threads |
|---|---|---|---|---|---|
| minrank minors 9_9_6/mat1 | 94.85% | 92.14% | **84.94%** | 46.81% | 36.74% |
| mr 9_9_6/mat2 | 92.65% | 89.79% | **70.94%** | 39.01% | 28.41% |
| mr 9_9_6/mat3 | 93.31% | 84.44% | **71.30%** | 41.26% | 32.44% |
| mr 9_9_6/mat4 | 94.31% | 92.48% | **69.59%** | 41.91% | 33.01% |
| mr 9_9_6/mat5 | 94.84% | 92.22% | **69.39%** | 41.56% | 33.51% |
| mr 9_9_6/mat6 | 94.07% | 96.47% | **72.82%** | 43.60% | 35.37% |
| mr 9_9_6/mat7 | 93.97% | 92.90% | **69.57%** | 41.45% | 33.77% |
| mr 9_9_6/mat8 | 95.44% | 90.64% | **66.19%** | 38.15% | 27.85% |

From the results in table 2.1 we notice that the best speedup is achieved by 8 threads, and then it decreases beyond that. This can be explained by the high concurrency and waiting times due to the use of coarse-grained synchronization used in the above algorithm: threads finish faster and contend over the waiting list.

The efficiency, however, is around 87% and 95% with 2 or 4 threads but decreases when we use more threads. Indeed, with 2 or 4 threads, the amount of work spent on the actual *axpy* operations is more important than that spent on the critical sections. The best efficacity/speedup is obtained with 8 threads. Again, to avoid such speedup degradation, it is paramount to use more fine-grained synchronization constructs and a more robust load balancing strategy so that the threads would have equal amount of work between the reduction of fresh rows from the matrix on one hand, and the reduction of the rows in the waiting queue on the other hand.

## 2.3 A New Ordering Of Operations For Faugère-Lachartre

In the original Faugère-Lachartre algorithm, when the number of the evident pivots (rows of the sub-matrices $A$ and $B$) is large compared to the number of the non-pivots (rows of $C$ and $D$), most of the work is spent reducing the evident pivots by each other: the $Trsm$ step ($A^{-1} \times B$). For instance, in [2], we can notice that the FGb library [6] outperforms significantly Faugère-Lacharte on the last matrices which are usually almost quasi-triangular (triangular with few more rows) with a running time scale of 10 to 50.

Applying the $Trsm$ step is not strictly required if one is only interested by computing a row echelon form. Indeed, computing $A^{-1} \times B$ is only required when we are interested by the reduced row echelon form: inserting zeros below and above the pivots' leading elements. For the row echelon form, only

inserting zeros below the pivot's leading elements is required; this is equivalent to performing reductions over the rows of sub-matrices $C$ and $D$ only.

We present in what follows a method that is based on the original Faugère-Lachartre algorithm, but in which we perform reductions directly on the non-pivot rows. We show that, in case only a row echelon form is required, this new method outperforms the old method ($Trsm - Axpy$ - $Gauss$) on full rank matrices: e.g. matrices issued from the F5 algorithm.

Besides the performance benefits, the density of the row echelon form computed by this new method does not vary greatly from that of the input matrix leading to rather small size matrices; this method has also *less memory footprint* than the original Faugère-Lachartre algorithm.

### 2.3.1 Sketch of the new method

As in Faugère-Lachartre, we separate the elements of the original matrix $M_0$ into 4 sub-matrices $A$, $B$, $C$ and $D$. Afterwards, we reduce the rows of $C$ by the pivots of $A$ while reflecting these reductions on $D$ at the same time. Notice that the reduction of $C$ and $D$ in this case is different from the $Axpy$ step of the original algorithm; in the original $Axpy$ step, $A$ was already equivalent to $Id_{N_{piv}}$ which is not the case this time.

#### 2.3.1.1 Non-pivot reduction ($C$ and $D$)

After the analysis and decomposition steps 1.5, we perform the reduction of $C$ and $D$ at the same time as presented in the algorithm 2.3. The matrix is supposed to be represented by rows.

**Algorithm 2.3** Reduce C and D

```
/* loop over the rows of C and D */
for i = 1 to n_0 - N_piv do
    copy_row_to_dense_array(C[i,⋆], temp_C)
    copy_row_to_dense_array(D[i,⋆], temp_D)
    /* loop over the elements of row temp_C */
    for j = 1 to N_piv do
        if temp_C[j] ≠ 0 then
            /* temp_C = temp_C - temp_C[j] × A[j,⋆] */
            axpy(temp_C, temp_c[j], A[j,⋆])

            /* temp_D = temp_D - temp_C[j] × B[j,⋆] */
            axpy(temp_D, temp_C[j], B[j,⋆])

    copy_dense_array_to_row(temp_D, D[i,⋆])
```

Unlike the original $Axpy$ algorithm, reductions are now performed on the rows of $C$ at the same time as those of $D$. We loop through the rows of $C$ and $D$ (order is not important) and start by copying the rows to temporary dense arrays, $temp_C$ and $temp_D$. After that, we step through all the non-zero elements in $temp_C$: these are the coefficients by which we will reduce the rows of $D$. Notice, however, that these scalars change as we advance through the inner loop: $temp_C$ is now being reduced at the same time as $temp_D$. *axpy* is then applied on both rows of $C$ and $D$, the coefficient $temp_C[j]$ is used in both operations.

At the end of the inner loop, we write back the $temp_D$ array to the corresponding row in $D$ using the function *copy_dense_array_to_row*. We do not perform any writing on $C$ because its elements are not of any use anymore.

#### 2.3.1.2 Non-pivot block reduction of $C$ and $D$

Over block matrices, it is natural to make the reduction of $C$ and $D$ conform to the way matrices are represented. However, taking advantage of this block disposition comes at a cost: we cannot make use of the "ephemeral" nature of the elements of $C$ anymore. In the previous algorithm over rows, we never write back to $C$; this is not the case anymore if the matrix $C$ is represented by blocks.

In what follows, we consider the same block disposition as in the original block version of Faugère-Lachartre as mentionned in 1.4.1.

In the block version, we have to reduce completely all the rows in a given block before reducing the rows of the next block. However, unlike blocks of $D$, where there is no dependency on the columns, on $C$, the operations on the blocks at the left affect all the blocks on their right; this means that operations must be carried out as we go through blocks from left to right. This is close to the technique used in the Gauss step of the original Faugère-Lachartre algorithm where a dense matrix $P$ is used to save the operations to carry out while the algorithm advances from left to right.

Furthermore, reducing matrices $C$ and $D$ at the same time involves a lot of writing to the matrix $D$: a block of $D$ is reduced on average $nb\_cols\_C$ times. Thus, we have to reduce the matrix $C$ separately and save the coefficients back in order to use them for the reduction of $D$. The reduction of $D$ is then equivalent to performing the block version of $Axpy$ .

The algorithm 2.4 describes the block reduction of $C$.

**Algorithm 2.4** Reduce C - block

```
/* for all the rows of blocks in C */
for i = 1 to nb_block_rows_C do
    /* for all the blocks in the row C[i] */
    /* C blocks are numbered from right to left, must start from the end */
    for j = nb_block_cols_C down to 1 do
        /* carry out the reductions from the previous block */
        for k = nb_block_cols_C down to j + 1 do
            Axpy'_block(A[k, j], C[i, k], C[i, j])

        /* reduce the actual block */
        Trsm'_block(A[j, j], C[i, j])
return C
```

$nb\_block\_rows\_C$(resp. $nb\_block\_cols\_C$) represents the number of block rows (resp. the number of block columns) of the matrix $C$. The $Axpy_{block}$' and $Trsm_{block}$' are a slight variation of the $Axpy_{block}$ and $Trsm_{block}$ algorithms; the elements inside the blocks of $C$ are not organized the same way as those of $B$ and $D$ (they are listed from right to left whereas those of $B$ and $D$ are from left to right), this is the only modification in the $Axpy_{block}$' and $Trsm_{block}$' that is required for this algorithm. The matrix $C$ becomes very dense after this step (from 70% to 95% in our experiments.) This is an efficiency downfall compared to the previous algorithm over rows. First there is the overhead of writing the coefficients back to the matrix and then the fact that any block at index $i$ is read $i$ times in order to reduce subsequent blocks. Notice that we save the elements used in the inner block $axpy$ operations back to the blocks of $C$.

We will show in the experimental results that the block version of this algorithm is indeed less efficient than the non-block version. We solve this by applying a reduction over the rows of $C$, copy $C$ to a block matrix $C_{block}$, and then perform $Axpy$ over $C_{block}$ and $D$; this is actually much more efficient than an all-block or an all-non-block implementation.

#### 2.3.1.3 Parallelization of the non-pivot reduction step

Parallelizing the non-pivot reduction step is trivial also, whether the matric is a row matrix or a block matrix, there is no dependency between the rows of $C$. This makes parallelizing this step equivalent to assigning each row (or row of blocks) to a thread without any synchronization involved.

#### 2.3.1.4 Row echelon form reconstruction

The new pivots can now be located in the matrix $D$ with a Gaussian elimination as shown in 2.1.1. Reconstructing a row echelon form is very similar to the reconstruction step we have presented in 1.3.1.7 with a few differences:

- reconstructing the indices' maps of the non-pivot columns is now more intuitive since we have only to locate the new pivot rows in $D$ and their corresponding columns (no $B2$ or $D2$ indexes are involved);
- the coefficients of $A$ along with those of $B$ and $D$ are now included in the final matrix.

#### 2.3.1.5 Reduced row echelon form with the new method

The reduced row echelon form can also be computed using this new method. A second iteration is then required as in the original algorithm. The matrix we have just reconstructed –that we call $M_{0\_echelon}$– during the step 2.3.1.4 is now decomposed into 2 sub-matrices $A$' and $B$'.In fact, the matrix is decomposed exactly as in the original algorithm to the 4 sub-matrices $A$', $B$', $C$' and $D$'; however, since $M_{0\_echelon}$ is already in a row echelon form, the sub matrices $C$' and $D$' have whether 0 rows if $M_{0\_echelon}$ is a full rank matrix, or are matrices with only empty rows otherwise.

Now, one has simply to reduce the pivot rows of $M_{0\_echelon}$, the rows in sub-matrices $A'$ and $B'$ after decomposition, by each other: this can be achieved using the $Trsm$ algorithm of the original Faugère-Lachartre to compute $A'^{-1} \times B'$. The reconstruction of the final reduced row echelon form is the same as in the original algorithm.

Figure 2.3.1 shows the different steps of this new method.

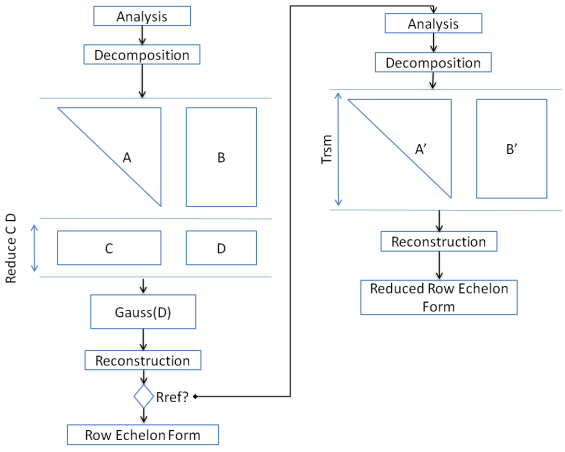


Figure 2.3.1: Sketch of the new ordering of operations of Faugère-Lachartre

#### 2.3.1.6 Experimental results

We show, in a nutshell, results of the efficiency of this method compared to the original Faugère-Lachartre algorithm; we measure the running time (in seconds) and the memory consumption (the Resident Set Size) in Megabytes. We compute an echelon form of the matrices issued from Katsura

13 and Minrank minors 996. The experiments were carried out on an Intel(R) Xeon(R) X5677 CPU @ 3.47GHz, 12MB of L3 cache and 144GB of RAM.

Table 2.3: New vs old method - row echelon form running time - Katsura 13

| Matrix | New method (second) | Standard Faugère-Lachartre (seconds) | Memory usage new method (MB) | mem usage standard Faugère-Lachartre (MB) |
|---|---|---|---|---|
| mat1 | **0.02** | 0.04 | **3.76** | 7.10 |
| mat2 | **0.16** | 0.39 | **14.89** | 42.52 |
| mat3 | **0.94** | 2.13 | **57.24** | 173.00 |
| mat4 | **3.54** | 7.70 | **154.80** | 454.90 |
| mat5 | **8.41** | 17.13 | **291.70** | 743.90 |
| mat6 | **12.65** | 31.56 | **362.90** | 1132 |
| mat7 | **11.67** | 34.59 | **441.60** | 1334 |
| mat8 | **7.20** | 34.96 | **469.80** | 1384 |
| mat9 | **3.15** | 33.66 | **476.80** | 1365 |
| mat10 | **1.74** | 33.62 | **477.60** | 1352 |
| mat11 | **1.45** | 38.01 | **478.60** | 1351 |

Table 2.4: New vs old method - row echelon form running time - Minrank minors 996

| Matrix | New method (second) | Standard Faugère-Lachartre (seconds) | Memory usage new method (MB) | mem usage standard Faugère-Lachartre (MB) |
|---|---|---|---|---|
| mat1 | **6.17** | 6.2 | **115.6** | 116.5 |
| mat2 | **19.06** | 22.71 | **570.3** | 587.1 |
| mat3 | **66.69** | 101 | **1380** | 1642 |
| mat4 | **166.6** | 280.4 | **2746** | 3071 |
| mat5 | **317.1** | 570.4 | **4440** | 5479 |
| mat6 | **530.8** | 1112 | **6273** | 7918 |
| mat7 | **548.1** | 1464 | **7812** | 9758 |
| mat8 | **295.1** | 1397 | **8595** | 10590 |

The new method is clearly more efficient on both the running time and the memory utilization. Notice that these are full rank matrices (Katsura 13 and Minrank minors 9_9_6). On matrices issued from F4, this new method does not lead the same efficiency. We provide more insights about these results on the chapter [experimental results ??].

# Chapter 3

# Implementation, Data Structures and Performance Considerations

In this chapter we report on the data structures, the different performance enhancements along with efficiency comparisons with other implementations of the Faugère-Lachartre algorithm. We show the various versions we have implemented and the improvements we introduce with each version. In 3.3.1 we discuss LELA's implementation of Faugère-Lachartre and its efficiency; we also mention briefly code optimization techniques in 3.1. In the following sections we discuss the versions of our implementation and their characteristics: the $SparseVector$, the *multiline* and the block versions. We then elaborate about the new method consisting in the new ordering of operations for Faugère-Lachartre in 3.7. Then, the parallel implementation and the scalability difficulties we have faced are discussed in 3.8. Finally, in section 3.9, we address the memory utilization of our method and compare it to the original Lachartre's version; we also briefly discuss the possibility of a distributed version of the Faugère-Lachartre algorithm in the same section.

## 3.1 Code Optimization

Software optimization is a well-documented subject. Most of the techniques when applied as they are lead to performance enhancements, while others need more fine tuning and profiling. An excellent reference about the subject is: "Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms" by Agner Fog [10]. We mention briefly the optimizations we have used in our implementation without elaborating in details the theory behind them.

### Compiler built-in optimization

Compilers can generate very optimized binary code by default. However, one can specify the level of optimization needed with specific options: for example, automatic loop unrolling, function inlining etc. In *gcc* for example (GNU Compiler Collection) using the $-O2$ or $-O3$ options can lead to great performance enhancements without any actual code tuning. Other options are available for more specific optimization.

### Loop unrolling

Loop unrolling is a technique that aims to minimize the loop controlling conditions. In a standard $for$ loop, an end loop test is usually necessary at each iteration, however, on an unrolled loop, these tests can be reduced subsequently by duplicating the functionality of the loop body. This is also beneficial

when accessing contiguous data in an array because the compiler can define ahead the offsets for accessing the array. an example of loop unrolling:

```
1  for(int i=0; i<100; ++i)
2      handle(i);
```

A corresponding unrolled loop can be:

```
1  for(int i=0; i<100; i+=4) {
2      handle(i);
3      handle(i+1);
4      handle(i+2);
5      handle(i+3);
6  }
```

In the above loop, only 20% of the branching tests are performed, leading to less loop management overhead. If the operations in the loop body are on contiguous array elements, the compiler can take advantage of that also and pre-compute the offsets for faster access. If supported, the compiler can even execute the statements inside the loop body in parallel if they were independent.

We make heavy use of loop unrolling in our program. We have noticed that the best loop unrolling degree corresponds to the quantity of data that has a size multiple of a cache line. For example in our experiments, with arrays of 64 bit elements (8 bytes), and a cache line size of 64 bytes, the best unrolling results are achieved with an unrolling degree of 16. Profiling is clearly needed in order to identify the most adequate unrolling degree.

## Prefetching

Data prefetching makes it possible to bring data from the main memory into the cache before it is accessed by the program. If the prefetch is done early enough, the data would eventually be available in the cache before the program can access it. This is aimed to minimize cache misses in cases where the compiler cannot predict the data to fetch next (case of sparse linear algebra!). The prefetching instructions are not blocking instructions, which means that the program issues a prefetch instruction and continues immediately the execution.

On *gcc*, one can use the built-in function *__builtin_prefetch* (*const void *addr*, ...) where *addr* is the address of the data to prefetch. On architectures that do not support prefetching, *gcc* simply drops the prefetching instructions (but evaluates the passed arguments for side effects nonetheless).

We have made usage of prefetch instructions at some point in our program. We could notice a 10% to 20% performance gain on cases where the access pattern is easy to predict. However, prefetching is not used in the final versions of our code.

## Data alignment

On modern architectures, the CPU reads and writes data from/to memory in word size chunks. Data alignment consists in putting the data at a memory address that is multiple of the word size (e.g. 4 bytes on 32 bit architectures). Aligning data could require adding padding to the end of data structures and/or using specialized memory allocation constructs when dynamic memory is needed.

Throughout our implementation of Faugère-Lachartre, we have made usage of aligned data when storing elements back to the matrices. We use basically *std* :: *vectors* as our underlying storage type. Handling aligned dynamic memory allocations with *std* :: *vectors* requires the use of custom allocators which are passed as a template parameter to *std* :: *vector*. An allocator manages all the underlying memory of the *std* :: *vector* and disposes of several functions for that. The most important one is *allocate* which is responsible of allocating dynamic memory for the *std* :: *vector*.

To allocate aligned memory, we use the *posix_memealign* function from *stdlib.h*. Our allocate function is the following:

```
1  pointer allocate (size_type size, const_pointer *hint = 0) {
2       pointer p;
3       posix_memalign((void**)&p, 16, size * sizeof (T));
4       return p;
5  }
```

### Inligning

Inlining is the process of replacing the function call with the body of the called function itself. This is very practical since it eliminates the time overhead of the function call/return instructions, and also improves space usage: code is packed together in the instruction cache. On small functions (i.e. with few lines of code), inlining can be very beneficial, however, it can also introduce some overhead if the functions are big. Once again, profiling is a must to identify if inlinng would benefit the performance of some code or not.

The *inline* keyword can be used to hint to the compiler that the following function in the code should be inlined; however, this is only a hint, the compiler may decide otherwise and not inline the function. On *gcc*, one can also specify in the code that a function must be inlined using the attribute "*always_inline*":

```
inline void foo (const char) __attribute__((always_inline));
```

The attribute "*noinline*" can also be used to make sure the function will not be inlined if the optimization flags are set ($-O2$ for example).

### Cache optimization

The cache is a small intermediary memory between the CPU and the main memory; it is usually shared by several cores. There exist several levels of the cache, generally $L1$ to $L3$, the most used data resides in the most inner level: $L1$ is faster than $L2$ and $L2$ is faster than $L3$. Latency times to access the main memory can be hundreds of times slower than cache latency. Between the cache levels themselves, for example $L1$ and $L2$, the latency can vary by a factor of 10 or more.

For this reason, making good use of the cache is paramount to achieve efficiency. The block version of Faugère-Lachartre 1.4 was mainly designed to take advantage of caching.

A very detailed study on the cache and cpu-cache access optimization from a programming point of view can be found in [11].

### 3.1.1 Sparse Vector AXPY and In Place Computations

On sparse Linear Algebra problems, the matrices and vectors are saved in different formats that take the minimum amount of memory while still offer efficiency. We will show the benefit of using a dense temporary array to present sparse vectors. We have been using this technique all along the algorithms we have encountered so far.

#### Sparse-Sparse vector reduction: axpy

Consider the two sparse vectors $x$ and $y$. We would like to perform an *axpy* operations on both, in other words compute $y \leftarrow a \times x + y$ for some scalar $a$. The following struct represents a sparse vector which is composed of an array of positions and another array of the values corresponding to the elements of the vector.

```
1  typedef strcut sparse_vector {
2          std::vector<int> positions;
3          std::vector<int> values;
4  } sparse_vector;
```

The pseudo-code of the *axpy* function over two sparse vectors is shown in the following code. It is inspired directly from *LELA*'s *BLAS*1 :: *axpy*() function.

```
1  void axpy_sparse_sparse(int a, sparse_vector x, sparse_vector y) {
2      strcut sparse_vector tmp;
3      int i=0, j=0;
4      long int c;
5
6      for(i=0, j=0; i<x.positions.size (); ++i) {
7          while(j<y.positions.size () && y.positions[j] < x.positions[j]) {
8              tmp.positions.push_back (y.positions[j]);
9              tmp.values.push_back (y.values[j]);
10             ++j;
11         }
12
13         if (j < y.positions.size () && x.positions[i] == y.positions[j]) {
14             c = a * x.values[i] + y.values[j];
15             ++j;
16         }
17         else
18                 c = a * x.values[i];
19
20         if(c != 0) {
21             tmp.positions.push_back (x.positions[i]);
22             tmp.values.push_back (c % RING_MODULUS);
23         }
24     }
25     while (j < y.positions.size ()) {
26         tmp.positions.push_back (y.positions[j]);
27         tmp.values.push_back (y.values[j]);
28         ++j;
29     }
30
31     copy_tmp_to_sparse_vector(tmp, y);
32 }
```

Notice how much overhead is entailed by the above *axpy* function, indeed, before any addition/-multiplication a test is made to check what index the elements are originated from. This is clearly impractical knowing that the most used function in the structured Gaussian elimination and the Faugère-Lachartre algorithm is the *axpy* function.

We show now how this *axpy* operation is done if the $y$ vector is represented as a dense temporary array in the following code:

```
1  void axpy_sparse_dense_temp(int a, strcut sparse_vector x, int y[]) {
2      for(i=0, j=0; i<x.positions.size (); ++i) {
3          y[x.positions[i]] += ((long int)a * x.values[i]) % RING_MODULUS;
4      }
5  }
```

The complexity of the above code is exactly of the size of the non-zero elements in the vector $x$. This method is of great use in the case of the structured Gaussian elimination and the $Trsm$, $Axpy$ and $Echelonize$ parts of the Faugère-Lachartre algorithm. Indeed, in those methods, a vector is reduced by all other pivots before it, which means that if we represent this vector as a dense array and used the function *axpy_sparse_dense_temp*, the number of arithmetic operations is exactly the number of non-zero elements of these pivots. Had the *axpy_sparse_sparse* function been used, most of the time would be spent on branching tests rather than actual computations. This method increases the performance of the code by hundreds of times.

One can also use a big data type for the temporary dense array in order to accumulate operations and avoid the modulo reductions after each addition. For example, over sparse vectors of elements represented with 16 bits, and accumulator of size 64 bits can be used to accumulate over $2^{31}$ additions before any need to apply the modulo reduction. This allows a gain that can reach 50% in performance, refer to section 3.4.2 for detailed results.

## 3.2 Generalities about the implementation

### 3.2.1 Main structure of the program

During the implementation of the different versions of the Faugère-Lachartre algorithm, a pattern has emerged about the architecture and organization of the components of our C++ code. These components are:

- **The indexer**: this is the class responsible for the analysis, splicing (decomposition) and reconstruction of the final matrix. The indexer is adapted according to the different data structures used in the different versions we have developed. There is also an analogous "parallel indexer" which have some parallel traits in the splicing/reconstruction steps.
- **Matrix-ops** (matrix operations): this class is responsible for everything from vector level operations (*axpy*, *head*, *normalize* etc.) to matrix level operations: $Trsm$, $Axpy$, *echlonize* etc. Decomposing this class further was very helpful in obtaining good modularity for the whole code in the case of block data structures; we have 3 levels for matrix-ops:
  - **Level1-ops**: low level vector and memory operations like *copy_vector_to_dense_array*, *head*, *normalize* etc.
  - **Level2-ops**: vector-level and block-level operations: $axpy_{sparse}$, $axpy_{hybrid}$, $Axpy_{block}$, $Trsm_{block}$ etc.
  - **Level3-ops**: matrix level operations: $Axpy$, $Trsm$, $Echlonize$ (Gauss).
- **Matrix-utils**: this class has different utilities used throughout the code like reading/writing matrices from/to disc; equality comparison between matrices of different data structures; conversion of matrices to other representations: from sparse rows representation to a multiline or block representation and vice versa; dumping an image representation of the matrix etc.
- **Main program**: implements the actual Faugère-Lachartre algorithm using the different methods of the aforementioned components.

### 3.2.2 Notes on the Validation of Results

Due to the number of steps of the Faugère-Lachartre algorithm and the multiple versions/data structures we have implemented, it was paramount to validate each step of the algorithm separately on the go. Furthermore, some computation errors might not appear unless relatively large matrices are used, which makes it difficult to rely on small dummy data. To this end we have used a progressive method by which we validate results as we use different data structures and progress in the algorithm steps; the main indicators of correctness used are the following:

- **The rank**: the rank of the matrix obtained is a strong indicator about whether the outcome is correct or not. For $F5$ matrices, the rank is equal to the row size; and for $F4$, matrices might undergo a rank loss. We compare the rank from our program with other implementations (e.g. the original implementation of Sylvain Lachartre). However, the rank does not guarantee that the result is correct.

- **The shape of the resulting matrix**: we usually dump the matrix as an image file (c.f. 3.2.3) and visualize it compared to the image file of a matrix issues from a reliable implementation, this is very helpful since it allows spotting the errors quickly.
- **Rref**: the ultimate indicator is computing the *reduced row echelon form* of our matrices and then comparing them with the *rref* of the corresponding matrices issued from a reliable implementation. Indeed, each matrix has a unique echelon form, hence, if this test fails, then it is guaranteed that the matrices are no equivalent.

Obviously, several steps in this process are critical. For example, the equal operation between two matrices must be very reliable or the overall validation could fail. We use *LELA*'s $BLAS3 :: equal$ as much as possible when we can convert our matrices to *LELA*'s native types ($SparseMatrix$); these methods are considered reliable enough. For the *Rref* validation step, we have developed the structured Gauss method which we validated against *LELA*'s naïve Gaussian elimination.

While developing a version $i$, the outcome of each step is validated against that of the corresponding step in version $i-1$. For example, the matrices issued from the analyze step of the block version are validated against the matrices issued from the analyze step of the normal row version. Only after we make sure that this analyze step is reliable enough, we advance to developing the following steps. Our very first version was validated against *LELA*'s implementation [REF]. Using the most recent stable version is useful since it is usually the most efficient too; this means we can validate larger matrices in less time as we progress.

### 3.2.3 Visualization of the Structure of Matrices

As we have mentioned in the validation step, visualizing the structure of the matrices is very helpful to spot computation errors. To this end, we have developed small routines to represent our matrices as image files that can be visualized with different imaging tools. We use the ***pbm*** (*Portable BitMap*) format which is a simple black and white image format. In this format, an image can be considered as a matrix $n \times m$ of bits where $0s$ represent a black pixel and $1s$ a white pixel. The header contains information about the file type, the size of each row, and the number of rows in the bitmap.

**Example:**

```
P1          #magic header, comment after # is ignored
5 3         #the size of each row followed by the number of rows
0 1 0 1 1
1 1 0 0 1
1 1 0 0 0
```

This is perfectly adapted to the nature of our matrices: they are distinctively sparse, and usually when there are computation errors, some elements would zero out and become clearly visible on a *pbm* format. Following is an example of *katsura12/matrix1*:

However, the *pbm* format is not suitable for big matrices; indeed, for a matrix of size $n \times m$, the size of the pbm file is $(n \times m)/8$ bytes. Using other image formats that support compression is necessary. For instance, *LELA* supports reading/writing binary matrices from/to *png* files. Unfortunately there is no current support for matrices over small size fields yet.

### 3.2.4 Format of the matrices on disc

When reading and saving matrices over $F_{65521}$ we use the following format to store the matrices on disc, all the bytes are written contiguously without any separators:

***Header***:

| number of rows | number of columns | Field characteristic | number of non-zero elements |
|---|---|---|---|
| 4 bytes (uint32_t) | 4 bytes | 4 bytes | 8 bytes (uint64_t) |

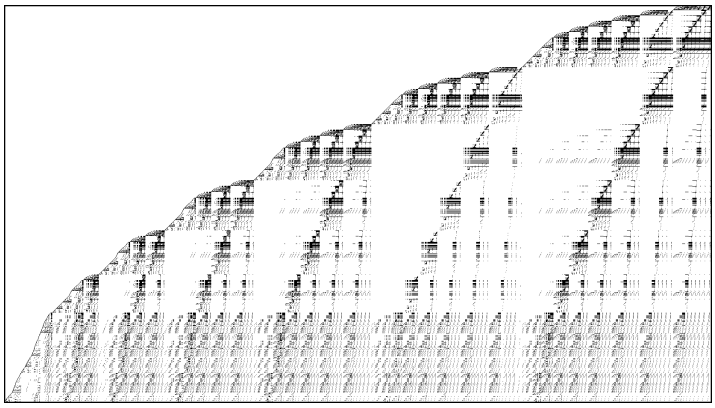

Figure 3.2.1: katsura12/matrix1 represented as a pbm image file

***Data***: If the number of non-zero elements in the matrix is $Nz$, then the data is saved as following:

| Nz non-zero values | Nz positions of non-zero values | number of non-zero elements in each row |
|---|---|---|
| 2 bytes each element | 4 bytes per element (uint32_t) | 4 bytes |

## 3.3 LELA Library

Library for Exact Linear Algebra (*LELA*) is a C++ template library for computations in exact Linear Algebra. We have used *LELA*'s data types all over the development process (in representing fields with its built in Modular class for example). *LELA* offers a great unified interface and flexibility when dealing with different types of matrices and vectors: dense, sparse, hybrid etc. Furthermore, *LELA* takes advantage of highly tuned specialized libraries when detected on the target systems. These libraries are:

- *BLAS* (Basic Linear Algebra Subprograms): can improve greatly the computations over dense matrices on $\mathbb{Z}/n\mathbb{Z}$, several options are avaiable: goto blas, ATLAS etc.
- *M4RI* [7]: for high performance calculations with dense matrices over $GF(2)$
- *Libpng*: for reading writing matrices over $GF(2)$ in png format.

The basic types of LELA are: Ring and Ring-elements for ring arithmetic; and vectors along with matrices over a Ring. The *BLAS* interface is a very intuitive and a unified way to perform different operations over matrices and vectors. The $BLAS1$ namespace contains various functions handling vector operations, the vectors can be of any types: sparse, dense, hybrid, hybrid over $FG(2)$ etc.; by use of C++ templates, LELA assures the most suitable method is used for any of the above types, this can be defined at compile time. $BLAS2$ contains matrix-vector routines while $BLAS3$ handles matrix-matrix operations.

### 3.3.1 LELA's Faugère-Lachartre implementation

LELA has a basic implementation of the Faugère-Lachartre algorithm. The analyze, decomposition and reconstruction steps are performed using a class called *Splicer*. LELA's implementation has a large memory footprint: indeed, the sub-matrix $A$ is a sparse matrix, whereas $B$, $C$ and $D$ are dense matrices. This is indeed very limiting in the case of very large matrices, a lot of memory can be wasted presenting $0s$ and a lot of overhead could be introduced with useless computations.

The $Trsm$ step (c.f. 1.3.1.3) can be performed using only one function call in LELA:

```
BLAS3::trsm (ctx, ctx.F.one (), A, B, UpperTriangular, false);
```

Likewise, the *Axpy* step is equivalent to the *gemm* routine in *BLAS*, using LELA's *BLAS*3 the *Axpy* step is performed by a call to:

```
BLAS3::gemm (ctx, ctx.F.minusOne (), C, B, ctx.F.one (), D);
```

The *ctx* parameter is a *Context* object which holds several information about the ring over which the computations are performed. *ctx.F* is the actual ring, and the *ctx.F.one*() and *ctx.F.minusOne*() are elements in the ring *ctx.F*.

**Experimental results**

We have performed running time experiments of LELA's Faugère-Lachartre implementation on sparse matrices over the field $F_{65521}$. We use 4 configurations: with *CBLAS* enabled or disabled, and over matrices' elements represented over 16 bit with the *uint*16_*t* types, or over 64 bits with the *double* type. One advantage of using the *double* data type is that the *CBLAS* routines can achieve better performance over these elements. One downfall is that the memory required is now 4 times larger compared to when using 16 bit elements.

We compare all these with our implementation that we will present in 3.6. The experiments were performed on an Intel(R) Core(TM) i7 CPU @ 2.93GHz, 8MB of L3 cache and 8GB of RAM. All the timings are in seconds. When no timing is given (marked with a "-") it means that the program takes abnormally long time that the experiment was aborted. Notice that we compute just a row echelon form and not a reduced row echelon form.

Table 3.1: LELA's Faugère-Lachartre implementation timings

| matrix | size | LELA with uint16_t | | LELA with double | | new |
|---|---|---|---|---|---|---|
| | | no BLAS | with BLAS | no BLAS | With BLAS | implementaion |
| kat13/mat1 | 1042 x 2135 | 2.38 | 2.3 | 0.79 | 0.39 | **0.04** |
| kat13/mat2 | 3827 x 6207 | 52.22 | 53.39 | 13.69 | 6.50 | **0.42** |
| kat13/mat3 | 10014 x 14110 | 2478 | 2463.8 | 422.16 | 41.45 | **2.42** |
| kat13/mat4 | 19331 x 25143 | - | - | 5574.56 | 183.74 | **8.98** |
| kat13/mat5 | 28447 x 35546 | - | - | - | 435.56 | **20.06** |
| Minrank minors 9 9 6/mat2 | 5380 x 22400 | - | - | - | 307.66 | **27.34** |

We can already notice that using the LELA Faugère-Lachartre implementation with matrices' elements represented with 16 bits is not efficient whether high tuned *BLAS* routines are used or not, both experiments take almost the same time. However, using elements represented with the *double* type we can see a clear performance benefit compared to the 16 bits version. Indeed, without *BLAS* enabled the double version is almost 6 times faster than the one with 16 bits. On the other hand, using *BLAS*, we can notice another performance shift for the *double* version which is about 10 times faster than the corresponding non *BLAS* version. Although the double version is faster, it requires 4 times the memory of the 16 bits version as we have already mentioned.

Despite all the benefit provided by *BLAS* and the use of the *double* data type, LELA's Faugère-Lachartre is not as efficient as our implementaion which can be almost 20 times faster.

LELA's Faugère-Lachartre implementation is, however, very efficient on matrix over $GF(2)$. Indeed, Martin Albrecht have performed a comparison of the efficacity of LELA's Faugère-Lachartre implementation with several other libraries including *M4RI*. It was shown [1] that LELA's implementation over $GF(2)$ outperforms indeed the other libraries, especially when the matrices get bigger.

[1] http://martinralbrecht.wordpress.com/2012/06/22/linear-algebra-for-grobner-bases-over-gf2-lela/

## 3.4 First version using the SparseMatrix type

As in the Structured Gaussian Elimination 1.2, we have used LELA's *SparseMatrix* type to represent our matrices. The rows are represented with the type *SparseVector* which is composed internally of two vectors: one for the values and the other for the positions of the elements. Both types are parameterized with the use of to C++ templates. Over all our implementation we will consider specifically matrices over $F_{65521}$: the elements of these matrix can be represented with 16 bits data types.

The *Trsm* algorithm from 1.6 can be translated using LELA's data structures like following:

```
1  void Trsm(const SparseMatrix<uint16_t> A, SparseMatrix<uint16_t> B)
2  uint64_t temp[B.coldim ()];
3  for(int i = A.rowdim ()-1; i>=0; --i){
4        copy_sparse_row_to_dense_array(B[i], temp);
5        for(j=1; j<A[i].size (); ++j){
6              register uint16_t Av = A[i][j];
7              register uint32_t Ap = j;
8              /* axpy: temp = temp - A[i][j] * B[j] */
9              for(int k=0; k<B.coldim (); ++k){
10                   temp[B[Ap][k].first] += (uin32_t)Av * B[Ap][k].second;
11             }
12       }
13       copy_dense_array_to_sparse_row(temp, B[i]);
14 }
```

*temp* is an array of size equal to the column size of the matrix $B$; its elements are of size 64 bits. We use *temp* as an accumulator in the *axpy* operations to avoid modulo reductions each time; we can perform up to $2^{31}$ additions in this accumulator without overflow. The *axpy* operation in the above code corresponds to the very inner loop over $k$; clearly the number of multiplications/additions corresponds exactly to the number of non-zero elements of the rows.

The value $A[i][j]$ by which we reduce is constant throughout the *axpy* operation, we can hence hint to the compiler to keep it in a *register* for fast access, the same is true for the position of the row, *Ap*, by which we reduce. LELA's *SparseVector* abstractly exposes elements from the two underlying vectors (values and positions) as a tuple: the tuple's member *first* is used for the position and *second* for the value.

### 3.4.1 Case of generic matrices

In case of generic fields, we can directly use LELA's built-in operations in the Ring module $Modular < Ring\_Element >$. For example, doing a multiplication-reduction operations can be performed with $Modular < Element > .axpyin()$ method. While this ensures that the program functions over any field, it can undergo severe performance degradation due to the modular reductions performed on each element during the *axpy* step.

### 3.4.2 Experimental results

We have implemented both methods (using an accumulator and using in-place computations) of the first implementation of Faugère-Lachartre with LELA's *SparseMatrix* with rows represented with LELA's *SparseVector*. When the accumulator is used, it is used throughout all the steps of the algorithm: *Trsm*, *Axpy* and *Gauss* (or *Echelonize*). We compute the reduced row echelon form in each experiment.

The experiment was performed on an Intel(R) Core(TM) i7 CPU with a clock speed of 2.93GHz, 8MB of L3 cache and 8GB of RAM. All the timings are in seconds.

Table 3.2: Running time of the first implementation of Faugère-Lachartre

| matrix | size | density | with accumulator | without accumulator |
|---|---|---|---|---|
| Kat13/mat1 | 1042 x 2135 | 5.02% | **0.076** | 0.122 |
| Kat13/mat2 | 3827 x 6207 | 3.69% | **0.915** | 1.658 |
| Kat13/mat3 | 10014 x 14110 | 2,97% | **6.819** | 13.26 |
| Kat13/mat4 | 19331 x 25143 | 2,69% | **28.77** | 58.21 |
| Minrank 996/mat1 | 1296 x 11440 | 99.99% | **31.14** | 67.99 |
| Minrank 996/mat2 | 5380 x 22400 | 45.29% | **110.9** | 235.3 |

The above table shows that using an accumulator is very advantageous; the running time of the version with an accumulator is generally two times faster than the version without an accumulator. Although the non-accumulator version is generic and can work over any generic field, we will focus in the following versions only on matrices over a field with a characteristic that can be represented over 16 bits (a prime less than $2^{16}$).

## 3.5 Second version: using the "multiline" data structure

The matrices issued from Gröbner basis computations have some very special properties as we have discussed in 1.3. Other than these properties, we can notice two patterns of the elements in the matrices: *horizontal* and *vertical*. In the vertical pattern, when $row_i[j] \neq 0$, $row_{i+1}[j]$ tends to be different from 0 too; On the other hand, a horizontal pattern can be spotted also: when $row_i[j] \neq 0$ then $row_i[j+1]$ tends to be different from 0.

Such grouping patterns can be easily spotted in the figure 3.5.1 of the matrix kat12/mat2 for example:

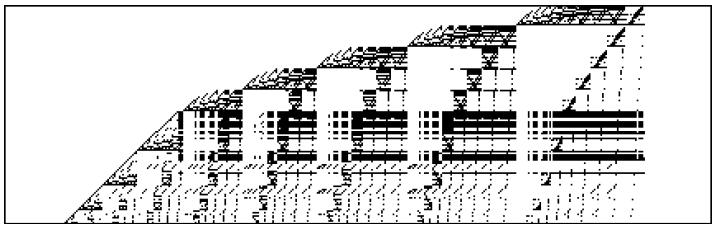

Figure 3.5.1: Patterns of elements' grouping in the matrix kat12/mat2

We can exploit this based on the flow of operations in the $Trsm$ and $Axpy$ steps. Consider, for example, the row $i$ and the column $j$; in the $Trsm$ step we reduce the row $B[i]$ by the row $B[j]$ using the scalar $A[i][j]$ (c.f. 3.4). Based on the previous patterns, the element at column $j+1$ tends to be non-zero and hence, the row $B[i]$ is likely to be reduced by the row $B[j+1]$ too.

Now considering the elements themselves inside the row $B[i]$; we can notice that when the element at column $t$ in $B[i]$ is reduced by the element $B[j][t]$, it is likely to be reduced by $B[j+1][t]$. This is due to the horizontal pattern in the matrix $A$ which is projected as vertical pattern in $B$. We shall mention that $B$ is generally relatively dense, and hence the elements $B[i][t]$ and $B[i+1][t]$ are likely to be non-zeros.

The schema in figure 3.5.2 shows the relation of these patterns in the $Trsm$ operation:

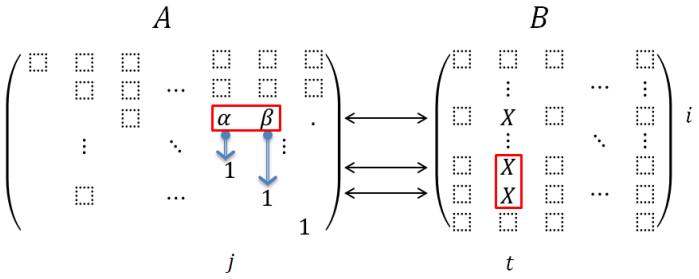


Figure 3.5.2: Flow of operations exploiting grouping pattern in the $Trsm$ step

In the figure 3.5.2, on line $i$, if the elements at columns $j$ and $j+1$ in $A$ are not 0 (horizontal contiguous pattern), the element $B[i][t]$ will be reduced by the both $B[j][t]$ and $B[j+t][t]$ (vertical contiguous pattern).

**Definition:**

We call a multiline vector the data structure in which we present the elements of several sparse vectors with a unique positions' vector and a unique values vector adding zeros when necessary. The elements are stored contiguously following a column order: the elements of all the rows at column $i$ are stored contiguously in memory before those of the column $i+1$.The following class, represents the multiline data structure:

```
1  template <typename Element, typename Index>
2  class multiline {
3      int NB_ROWS;  /* the number of the vectors represented in this multiline */
4      std::vector<Index> pos;  /* the column index of the elements */
5      std::vector<Element> values;  /* the values, this vector has always a size
           NB_ROWS * pos.size() */
6  }
```

Internally, the multiline is stored as two different $std :: vectors$. The first, called $pos$, holds the positions, and the second, called $values$, holds the values of the elements stored by column major order. An element from row $x$ at position $i$ in the multiline has a column index $pos[i]$ and a value $values[i \times NB\_ROWS + x]$ where $NB\_ROWS$ is the number of rows packed in a single multiline.

***Example***: Consider the two rows:

$$row_1 = [\ 1 \quad 0 \quad 0 \quad 2 \quad 5 \quad 3 \quad 0 \quad 1 \quad 3\ ]$$
$$row_2 = [\ 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 3 \quad 0 \quad 2 \quad 0\ ]$$

A sparse representation of each row separated would be equivalent to (first column is at index 1):

$$row_1 = [\ (1,1) \quad (2,4) \quad (5,5) \quad (3,6) \quad (1,8) \quad (3,9)\ ]$$
$$row_2 = [\ (1,3) \quad (1,4) \quad (3,6) \quad (2,8)\ ]$$

Where the notation $(x, y)$ means the element of value $x$ is at column index $y$.

Representing this with a multiline data structure (of 2 rows) would be equivalent to merging the indexes of the two rows, and interleaving the values according to their original positions:

$$multiline_{(row_1,row_2)} = [\ (\{1,0\},1) \quad (\{0,1\},3) \quad (\{2,1\},4) \quad (\{5,0\},5) \quad (\{3,3\},6) \quad (\{1,2\},8) \quad (\{3,0\},9)\ ]$$

The notation $(\{x_i, x_j\}, y)$ means that the two elements $x_i$ and $x_j$ are at index $y$. Notice that $x_i$ and $x_j$ are from two successive rows. This multiline in represented in figure WWW.

| *positions* | 1 | 3 | 4 | 5 | 6 | 8 | 9 |
|---|---|---|---|---|---|---|---|
| *values* | 1 | 0 | 2 | 5 | 3 | 1 | 3 |
| | 0 | 1 | 1 | 0 | 3 | 2 | 0 |

Figure 3.5.3: the multiline data structure

### 3.5.1 Multiline row reduction

We now focus on the row reduction using this new data structure. For the sake of simplification we will only consider multiline rows that represent two sparse vectors. There are 2 cases for rows reduction now: when there are two successive (column-wise) non-zero horizontal elements in $A$ and when there are not (e.g. $\alpha$, $\beta$ are successive column-wise in the figure 3.5.2). Notice that the vertical elements in a multiline are always successive.

#### 3.5.1.1 Case of non-successive elements

In this case, the reduction is made only over the elements originated from *one vector* in the multiline. This may cause a lot of jumps and the fetching of unused data into the cache (since the elements from the 2 original vectors are now interleaved). What is important is that we reduce two rows at the same time. The following function performs the *axpy* operation between a multiline row represented by 2 dense arrays $temp_1$ and $temp_2$ on one hand, and another multiline vector on the other hand. Using the two scalars $a_1$ and $a_2$.

```
1 void axpy(uint16_t a1, uint16_t a2, multiline<uin16_t> v, uint64_t temp1[],
      uint64_t temp2[], int line) {
2       register uint32_t idx, val;
3       for(int i=0; i<v.size(); ++i)      {
4             idx = v.pos[i];
5             val = v.values[line + i*2];
6             temp1[idx] += (uint32_t) a1 * val;
7             temp2[idx] += (uint32_t) a1 * val;
8       }
9 }
```

*idx* represents the column index of the element at position $i$ in the multiline, and *val* holds the corresponding value of that element. The *line* variable indicates which vectors we are reducing by among those in the multiline. Again, we hint to the compiler that these variables should be used as ***registers*** for fast access. Notice that while we access contiguously the *positions* vector, we skip every other element in the *values* vector. This causes more cache misses and more overhead while the data is transferred from main memory to the cache.

The main benefit of this data structure is now visible: we are reducing 2 rows at the same time, notice how the variables *val* and *idx* are fetched only once from memory but are used twice.

#### 3.5.1.2 Case of successive elements

It gets more interesting when the horizontal pattern is observed on the rows of $A$: the case column-wise of successive elements. Indeed, in this case, we can perform 4 rows reductions at once as shown in the following code:

```
1  void axpy2(uint16_t a1_1, uint16_t a1_2, uint16_t a1_1, uint16_t a1_2, multiline<
       uin16_t> v, uint64_t temp1[], uint64_t temp2[]) {
2      register uint32_t idx;
3      register uint16 val1, val2;
4      for (int i = 0; i < v.size(); ++i) {
5          idx = v.IndexData[i];
6          val1 = v.values[i*2];
7          val2 = v.values[i*2+1];
8          arr1[idx] += (uint64_t)(((uint32_t) a1_1 * val1) + (uint64_t)((uint32_t) a1_2
               * val2));
9          arr2[idx] += (uint64_t)(((uint32_t) a2_1 * val1) + (uint64_t)((uint32_t) a2_2
               * val2));
10     }
11 }
```

---

As in the previous case, *idx* represents the column index of the element in position $i$; $val1$ represents the value at position $i$ of the first vector and $val2$ the value of the second vector at the positions $i$. $a1\_1$ and $a1\_2$ are the scalars from a given column $j$ ($j$ is the variable in the loop of the $Trsm$ algorithm), whereas $a2\_1$ and $a2\_2$ are the scalars from the column $j+1$.

Notice how we perform 4 reductions at one pass (each dense array is reduced by two rows).

#### 3.5.1.3 Introduction of useless operations with the multiline data structure

When using a multiline of size 2, we can make sure no operations are performed on null elements. For example, in the *axpy* and *axpy*2 functions above, we can perform tests before the actual loop to make sure to include in the actual computations only the non-zero scalars ($a1$, $a2$ for *axpy* and $a1\_1$, $a1\_2$, $a2\_1$, $a2\_2$ for *axpy*2). This becomes very complicated when we deal with multilines of variable size: we cannot predict every possible configuration and eliminate useless multiplications by 0. Indeed, for a multiline of size $n$, there are $n^2$ scalars to reduce by at each step (e.g. for size 2, we have used $2^2$ scalars in the *axpy*2 algorithm).

Furthermore, on variable size multiline rows, the memory overhead is very important too. Indeed, for a multiline of size 2, on a given column index, we can lose at worst 4 bytes in case one element is null and the other is not. On a multiline of size $n$, we can waste $n-1 \times 4$ bytes of memory in case only one element is not zero in the corresponding column at the $n$ successive rows. This entails a great overhead of useless computations as we have mentioned previously.

### 3.5.2 Experimental results

We have implemented a version of Faugère-Lachartre using the multiline data structure with values represented over 16 bits and positions over 32 bits. We use 64 bit dense array accumulators throughout the $Trsm$, $Axpy$ and $Echelonize$ to eliminate modulo reductions overhead. A comparison between the new version using the multiline data structure with the last version which uses the $SparseVector$ to represent the rows of the matrix (c.f. 3.4) is shown below. We compute the ***reduced row echelon form*** in each experiment.

The experiment was performed on an Intel(R) Core(TM) i7 CPU @ 2.93GHz, 8MB of L3 cache and 8GB of RAM. All the timings are in seconds.

Table 3.3: Running time of the multiline implementation of Faugère-Lachartre

| matrix | size | density | using LELA *SparseVector* | using *multiline* |
|---|---|---|---|---|
| Kat13/mat1 | 1042 x 2135 | 5.02% | 0.076 | 0.078 |
| Kat13/mat2 | 3827 x 6207 | 3.69% | 0.915 | **0.78** |
| Kat13/mat3 | 10014 x 14110 | 2,97% | 6.819 | **4.92** |
| Kat13/mat4 | 19331 x 25143 | 2,69% | 28.77 | **19.29** |
| Minrank 996/mat1 | 1296 x 11440 | 99.99% | 31.14 | **16.94** |
| Minrank 996/mat2 | 5380 x 22400 | 45.29% | 110.9 | **64.14** |

We can notice a clear shift in performance when the multiline data structure is used. On slightly dense matrices (the case of the last two ones in the table 3.3) there is a very great efficiency shift from 30% to 60%: this is mainly because the elements are usually contiguous inside the multiline and most of the reductions are *axpy*2 reductions which reduce 4 rows at a time.

#### 3.5.2.1 Disadvantages of the multiline data structure

As mentioned in 3.5.1.3, the multiline data structure can introduce some loss in memory and efficiency in case of general multiline sizes. However, this can be minimized with multilines of size 2.

The main disadvantage of using this data structure is the complexity of the code introduced, for instance, when reducing two multilines by each other in the *Gauss* step, the coefficients $a1\_1$, and $a1\_2$ are used directly, but not the coefficients $a2\_1$, $a2\_2$. Indeed $a2\_1$ and $a2\_2$ are to the left of $a1\_1$ and $a1\_2$, to this end, their image is computed as if we were only reducing by $a1\_1$ and $a1\_2$ before calling the actual *axpy*2 step.

This data structure is inefficient with algorithms that require sorting also. It can be easily seen that sorting individual rows would entails a complete rewrite of whole multiline rows. For example, exchanging the data of the first vector in a multiline with another vector in another multiline would entail the whole two multilines to be rewritten: all the data of all the vectors in both multilines. Notice that in case of simple vectors like *C++ STL std::vectors* or *LELA*'s $SparseVector$, this operation is an $O(1)$ operation. This is the main reason why we have chosen to use the standard structured Gaussian elimination in the *Gauss* step of the FGL algorithm, rather than the method described in 1.3.1.5. Refer to 2.1.1 for more details about this step.

## 3.6 Block Representation

A block representation of the matrices as described in 1.4 offers the Faugère-Lachartre algorithms a twofold efficiency gain: first, it allows more data to be packed in small blocks that can fit into the cache, thus, taking advantage of spatial and temporal locality. Second, separating data in block columns introduces the possibility to perform parallel processing over the columns of the matrices $B$ and $D$ since there is no depend between these columns.

We have seen in the previous sections that changing the data structure play an important role in achieving good performance. To this end, we have implemented several data structures to represent the blocks and compared the different running times in order to identify the most suitable data structure. We introduced a new type to represent block matrices: $SparseBlockMatrix$ which is a template class parameterized by the block type. This class is a list of rows of blocks where each row is a list of blocks. The different block representations are:

#### Blocks of SparseVectors

In this data structure, a block is simply a list of LELA's $SparseVector$. This block doesn't support hybrid rows.

### Contiguous Sparse Blocks

A contiguous sparse block is a block where all the elements of the block are stored in the same array *val*; all the corresponding positions are also stored contiguously in another array *pos*.A third list, *sz*, is used to hold the size of each vector in the original block. This is mainly the same block structure explained in [2], very similar to the CSR format [2].

The arrangement of the elements themselves in *pos* and *val* is different following the matrix the block is originated from. For instance, for the matrices $A$ and $C$, the elements are listed contiguously from right to left and then from down to top; whereas for the blocks of the matrices $B$ and $D$, the elements are listed from left to right then from down to top inside each block. Notice that the indexes of the elements change following the directions they are listed from: on $A$ and $C$, the index 1 is the right most column and on $B$ and $D$, the left most element is at index 1.

There is no support for hybrid rows in this data structure: all the rows are in a sparse format which means, every element in the *val* array has a corresponding element in the *pos* array implying that both the arrays have the same size all the time.

***Example***: consider the following bloc:

$$\begin{bmatrix} 1 & 2 & 0 & 7 \\ 0 & 3 & 9 & 0 \\ 0 & 1 & 4 & 0 \end{bmatrix}$$

If it was originated from $A$ or $C$ the corresponding lists are:

$val = \begin{bmatrix} 4 & 1 & 9 & 3 & 7 & 2 & 1 \end{bmatrix}$
$pos = \begin{bmatrix} 2 & 3 & 2 & 3 & 1 & 3 & 4 \end{bmatrix}$
$sz = \begin{bmatrix} 2 & 2 & 3 \end{bmatrix}$

Now the same block considered in $B$ or $D$:

$val = \begin{bmatrix} 1 & 4 & 3 & 9 & 1 & 2 & 7 \end{bmatrix}$
$pos = \begin{bmatrix} 2 & 3 & 2 & 3 & 1 & 2 & 4 \end{bmatrix}$
$sz = \begin{bmatrix} 2 & 2 & 3 \end{bmatrix}$

### Hybrid Contiguous Blocks

A contiguous hybrid block differs from the sparse block only by the fact that it takes into account hybrid rows. If a row has more elements than a given ***threshold***, then it is represented as a dense array introducing zeros when needed; we keep no elements in the *pos* array in this case. This is very practical because it ensures the memory used is always optimal: indeed, if the threshold is 50%, a sparse vector can at worst have half the elements represented as value-position pairs, which does not exceed the size of a dense array. If the number of the elements exceeds the threshold, then a dense representation would allow actually more memory savings compared to if only a sparse representation was used.

***Example***: on the following block, with a 60% threshold

$$\begin{bmatrix} 1 & 2 & 0 & 7 \\ 0 & 3 & 9 & 0 \\ 0 & 1 & 4 & 0 \end{bmatrix}$$

If it was originated from A or C the corresponding lists are:

$val = \begin{bmatrix} 4 & 1 & 9 & 3 & 7 & 0 & 2 & 1 \end{bmatrix}$
$pos = \begin{bmatrix} 2 & 3 & 2 & 3 \end{bmatrix}$
$sz = \begin{bmatrix} 2 & 2 & 4 \end{bmatrix}$

[2]CSR: Compressed Sparse Row

### Sparse MultiLine Blocks

Like a sparse block, a sparse mutliline block is a list of sparse *multiline*s rather than LELA's *SparseVector*s.

### Hybrid MultiLine Blocks

A hybrid multiline, analogous to a hybrid vector, is a mutliline where the elements are stored in a dense array in the case the number of the elements inside the multiline exceeds some *threshold*, introducing zeros when needed. If the number of elements in the multiline exceeds a given threshold then the multiline becomes simply a pack of $NB\_ROWS$ contiguous arrays, no positions' data is stored in the *pos* vector. A hybrid multiline block is then simply a list of hybrid multilines.

## 3.6.1 Experimental results

We have performed timing experiments over the 5 block data structures taking into account the $Trsm$ and $Axpy$ operations. The change of block type requires a complete rewrite of the underlying code of the actual $Axpy$ and $Trsm$. The blocks are of size $256x256$ and each multiline is holding 2 vectors at once.

The experiments were performed on an Intel(R) Core(TM) i7 CPU @ 2.93GHz, 8MB of L3 cache and 8GB of RAM. The timings of the $Trsm$ step using different data structures are shown; all the timings are in seconds shown in figure 3.6.1.

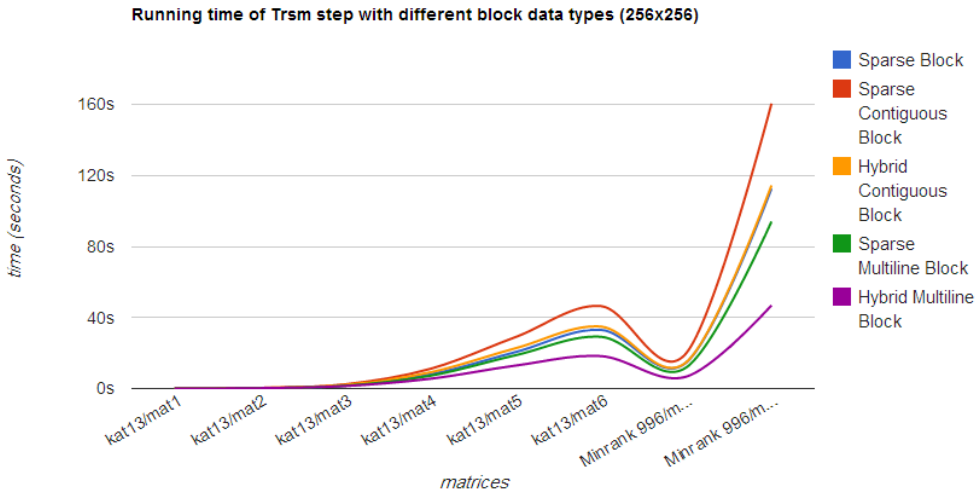


Figure 3.6.1: Running time of $Trsm$ with different block data types

Clearly both the sparse and hybrid multiline blocks outperform the other types. We notice however that the sparse contiguous is the most inefficient. The hybrid contiguous block has a running time almost equivalent to the sparse block. The sparse multiline block version outperforms the non-multiline ones, which is predictable given the performance of the multiline data structure shown in 3.3.

The hybrid multiline version is the most efficient being 3 to 4 times faster compared to the other data structures. This can be explained, in addition to the performance of the sparse multiline data structure itself, by the type of *axpy* operations performed on dense multiline rows. Indeed, one can take advantage of loop unrolling to a great level when the size of the block is known; also, in the case of dense multiline, the elements are stored and treated contiguously which allows the compiler to perform further optimization like prefetching and loop prediction.

In the case of hybrid multiline blocks, the simple *axpy* operation we have presented in 3.1.1 is modified to handle the multilines according to their nature: sparse/dense, block size/relative size, reduction by one rows/two rows. These functions are:

- SparseScalMulSub__one_row__vect_array
- DenseScalMulSub__one_row__vect_array
- DenseScalMulSub__two_rows__vect_array
- SparseScalMulSub__two_rows__vect_array
- DenseScalMulSub__one_row__vect_array__variable_size
- DenseScalMulSub__two_rows__vect_array__variable_size

In the $Trsm_{block}$ we also reduce a block by itself, leading to two more *axpy* operations on dense simple arrays:

- DenseScalMulSub__one_row__array_array
- DenseScalMulSub__two_rows__array_array

The compiler succeeds in the case of the latter two functions to generate SSE (Streaming SIMD Extensions) code using *mmx* registers.

### 3.6.2 Block version vs multiline version performance

We present in the table 3.4 the comparison between the running time of the block version and the multiline (matrix of multiline rows) version when computing the reduced row echelon form of several matrices. The multiline is of size 2 always and the blocks are $256x256$. The experiments were performed on an Intel(R) Xeon(R) X5677 CPU @ 3.47GHz, 12MB of L3 cache and 144GB of RAM.

Table 3.4: Running time of the multiline implementation of Faugère-Lachartre

| matrix | size | density | multiline version | block version |
|---|---|---|---|---|
| Kat13/mat5 | 28447 x 35546 | 2.65% | 38.07 | **27.75** |
| Kat13/mat6 | 34501 x 42315 | 2.46% | 59.57 | **41.38** |
| Kat13/mat7 | 38165 x 46265 | 2.55% | 63 | **44.73** |
| Kat13/mat8 | 39590 x 47768 | 2.65% | 58.23 | **42.13** |
| Minrank 996/mat3 | 12224 x 36784 | 32.84% | 214.9 | **131.5** |
| Minrank 996/mat4 | 21066 x 52502 | 27.43% | 583.6 | **346.8** |
| Minrank 996/mat5 | 30519 x 67094 | 24.07% | 1338 | **760.2** |
| Minrank 996/mat10 | 46830 x 88400 | 24.45% | 2593 | **1302** |
| Minrank 996/mat11 | 46956 x 88535 | 24.47% | 2622 | **1280** |

The block version is clearly advantageous compared to a simple multiline version. On dense matrices, the block version can even be more than 50% faster than the simple multiline version.

### 3.6.3 Notes on the matrix decomposition using blocks

The block decomposing and reconstruction of the matrices become relatively complex and introduced some overhead. First, the blocks are represented from left to right on $C$ and $D$ and from right to left to right on $A$ and $C$, this requires two passes on each row before its elements can be put in their

final positions. Using the multiline data structure entails some overhead when the rows are accessed individually during the matrix reconstruction also.

Generally the decomposition and reconstruction steps are very negligible compared to the overall execution time of the algorithm; however, on the last matrices, where only several non-pivot rows are present, the time in the decomposition and the reconstruction becomes very visible. More optimizations are possible in our implementation of the *Indexer* class responsible for the decomposition and the reconstruction of the matrices.

## 3.7 Implementation of the new ordering of operations for Faugère-Lachartre

We now consider the efficiency of our main contribution which consists in the new ordering of operations in Faugère-Lachartre presented in 2.3. We have implemented this new method using the multiline data structure; this allows us to perform the reduction of the rows of $C$ and those of $D$ at the same time without having to write back to $C$. Performing these two operations separately does not introduce a lot of overhead neither: in this case, the coefficients by which $D$ is to be reduced must be saved to $C$ and then used to reduce $D$ in a second time. Algorithm 2.3 shows how we can reduce $C$ and $D$ at the same time; performing separate reduction is simply applying the loop of this algorithm twice, first on the rows of $C$ and saving them back, then on the rows of $D$.

Up to this point, we have seen that the block version is well fast than the multiline version, however, in the case of the new method, the multiline version allows us nonetheless to take advantage of performing one pass over $C$ to reduce $D$. As explained in 2.3.1.2, when blocks are used to reduce $C$ in the new method, a block $i$ (starting from the left) is read at least $i-1$ times to reduce blocks of lower index in $C$. Not only using blocks adds the overhead of writing results back to $C$, but also the blocks of $C$ are read multiple times in order to reduce $C$ completely.

We show in the table 3.5 the running time of reduction of $C$ in the two cases: when $C$ is represented by multiline rows and when it is represented by blocks. The experiment was performed on an Intel(R) Core(TM) i7 CPU @ 2.93GHz, 8MB of L3 cache and 8GB of RAM. All the timings are in seconds.

Table 3.5: Reduce C, block vs multiline performance

| matrix | reduce C multiline | Copy $C_{multiline}$ to $C_{block}$ | reduce C block |
|---|---|---|---|
| Kat13/mat4 | 0.71 | 0.08 | 0.98 |
| Kat13/mat5 | 1.82 | 0.17 | 2.63 |
| Kat13/mat6 | 2.61 | 0.20 | 4.30 |
| Kat13/mat7 | 2.55 | 0.17 | 4.13 |
| Minrank 996/mat2 | 0.50 | 0.05 | 0.55 |
| Minrank 996/mat3 | 3.95 | 0.18 | 4.09 |
| Minrank 996/mat14 | 17.25 | 0.42 | 17.33 |

While the multiline is slightly more efficient than the block version in the case of dense matrices as in the case of the last matrices in the table, it is mostly 2 times faster on sparser matrices like those of the Katsura problem. To this end, we chose to use explicitly the multiline version while reducing $C$, then writing $C$ back to a block matrix, $C_{block}$, while releasing $C$'s memory on the go, since it is more efficient to perform a block reduction of $D$ which is more efficient than the multiline one. Another advantage to doing this is that the decomposition time is improved also (c.f. 3.6.3).

### 3.7.1 Experimental results

We can see in the results of table 3.6 that the new method does not suffer from the case when the matrices have only several non-evident pivot rows as in the case of the standard Faugère-Lachartre algorithm, the time spent on the $Trsm$ step is very penalizing when the reduced row echelon form is not required.

The table 3.6 shows the running time in seconds of the new method when computing a row echelon form.

Table 3.6: New method vs old method performance

| matrix | New FGL method | Old FGL method | Old/New |
|---|---|---|---|
| Kat13/mat4 | 3.54 | 7.697 | 2.17 |
| Kat13/mat5 | 8.409 | 17.13 | 2.04 |
| Kat13/mat6 | 12.65 | 31.56 | 2.49 |
| Kat13/mat7 | 11.67 | 34.59 | 2.96 |
| Minrank 996/mat2 | 19.06 | 22.81 | 1.20 |
| Minrank 996/mat3 | 66.69 | 101.3 | 1.52 |
| Minrank 996/mat4 | 166.6 | 280.9 | 1.69 |
| Minrank 996/mat12 | 24.85 | 1341.77 | 49.42 |
| Minrank 996/mat13 | 22.26 | 1390.76 | 55.03 |
| Mr10/mat2 | 104.7 | 131.9 | 1.26 |
| Mr10/mat3 | 318.7 | 535.9 | 1.68 |

On these matrices, the new method is always more efficient than the old one, we can see a speedup of 50 times faster on the last matrices, for instance the matrix 12 of Minrank Minors 996 problem.

However, the new method is less efficient on some occasions on non-full rank matrices as shown in the table 3.7. The matrices in the table are issued from the F4 algorithm and not full rank.

Table 3.7: New method vs old method performance on F4 matrices

| matrix | New FGL method | Old FGL method | Old/New |
|---|---|---|---|
| Kat11/mat5 | 2.208 | 2.276 | 1.03 |
| Kat11/mat6 | 2.752 | 3.218 | 1.17 |
| Kat11/mat7 | 4.026 | 3.802 | 0.94 |
| Kat12/mat4 | 9.992 | 8.306 | 0.83 |
| Kat13/mat5 | 20.63 | 18.06 | 0.88 |

More testing must be made on non-full rank matrices in order to deduce if the new method is not efficient for these cases. The Katsura 11 and 12 problems are relatively small and cannot be considered as definitive benchmarks of the efficacity of the new method.

We show in the figure 3.7.1 the speedup that we achieve using the new ordering of operations compared to our implementation of the standard Faugère-Lachartre algorithm. We can see that the new ordering of operations is always more efficient that the standard algorithm, with a scale from 1 to 5.

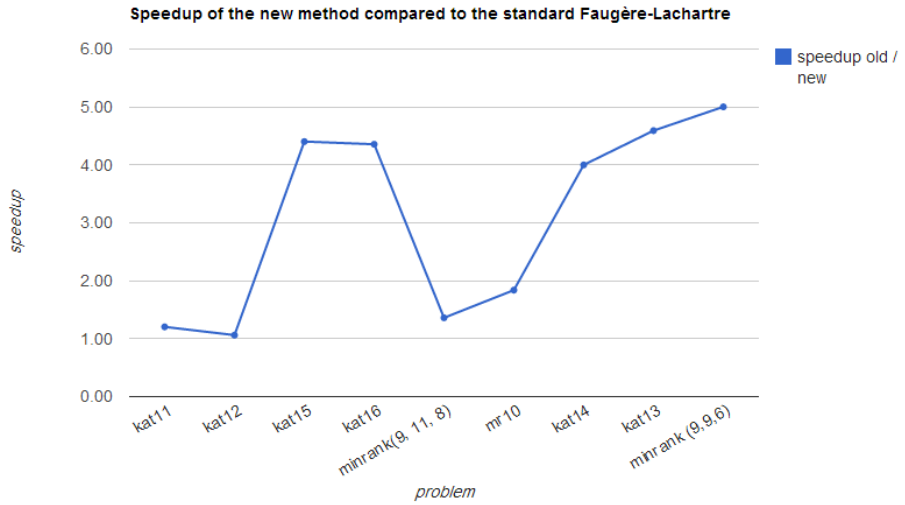


Figure 3.7.1: Speedup of the new ordering of operations compared to the standard Faugère-Lachartre

## 3.8 Parallel Implementation and Scalability Issues

Our parallel implementation differs from the original one in the way the parallel operations are carried out. In the original Faugère-Lachartre, the operations $Trsm_i$ and $Axpy_i$ are performed at the same time, reducing a block as soon as all the needed blocks to reduce this one are ready. In out implementation, the $Trsm$, $Axpy$ and $Gauss$ operations are applied in parallel but separately; this is motivated by the fact that our $Gauss$ step is now parallel and operates on the lines of the matrix rather than columns of blocks.

### 3.8.1 Generalities on $Trsm$ and $Axpy$ parallelization

The $Trsm$ and $Axpy$ steps of the Faugère-Lachartre algorithm are basically very easy to parallelize since there is no dependency between the columns of the matrices $B$ and $D$. However, achieving scalability of the algorithm is a very complicated task even without data dependency. Indeed, problems like cache inconsistency and false sharing are the most challenging parts on any parallel problem. In the table 3.8 we show the speedup achieved on the $Trsm$ step using a different number of threads (we omit some measures sometimes, marked with "-" in the table).

Table 3.8: Speedup of parallel $Trsm$ with different number of threads

| matrix | 2 threads | 4 threads | 8 threads | 12 threads | 16 threads |
|---|---|---|---|---|---|
| Minrank 996/mat3 | 1.98 | 3.67 | 7.12 | 7.25 | 7.48 |
| Minrank 996/mat4 | 1.84 | 3.85 | 7.14 | 7.39 | 7.44 |
| Minrank 996/mat5 | 1.71 | 3.75 | 7.34 | 7.04 | 7.40 |
| Minrank 996/mat6 | 1.79 | 3.71 | 7.35 | 7.45 | 7.36 |
| Minrank 996/mat7 | 1.83 | 3.67 | 7.29 | 7.45 | 7.32 |
| Mr10/mat2 | 1.99 | 3.83 | 7.43 | - | 7.63 |
| Mr10/mat3 | 2.01 | 3.94 | 7.47 | - | 7.50 |
| Mr10/mat4 | 1.98 | 3.87 | 7.51 | - | 7.46 |
| Kat16/mat7 | - | 3.55 | 6.81 | - | - |
| Kat16/mat8 | - | 3.64 | 6.96 | - | - |
| Kat16/mat9 | - | 3.94 | 7.66 | - | - |

We can see that the best speedup is achieved by 8 threads; using more threads does not bring any more speedup benefit.

We suspect that the cache is the main raison behind this; while blocks of $A$ are read only and can be shared by the threads, the columns of $B$ are not. Indeed, each thread uses the blocks of a column block $B_i$ fetching data into the cache as needed. When the block $B[i][j]$ is reduced, a row $r$ from the block $B[i][j-1]$ is fetched into the cache, it is used only while reducing this block and not reused till all the other blocks in the same column below $j$ are used. In the meantime, the other threads are also fetching rows in the same way which causes the row $r$ to be overridden in the cache. When a thread needs the row $r$ for the next block, it has to fetch it from the main memory again. We have tried to minimize this phenomenon using nested parallelism.

### 3.8.2 Nested parallelism

We have tried limiting the problem of poor reutilization of the cache using nested parallelism. The idea is that while threads reduce the columns $B_i$ in the $Trsm$ step for example, the $Axpy_{block}$ operation itself is now performed by several threads rather than only one. Indeed, in the $Axpy_{block}$ algorithm (c.f. 1.12), if the rows $i1$ and $i2$ of the $block_A$ have a non-zero entry at the same column index $j$, then the rows $B[i1]$ and $B[i2]$ are reduced by the same row $B[j]$. To that end, if the rows $i1$ and $i2$ were reduced by two separate threads, then if the first threads fetched the row $j$ to the cache, the second can directly use it from the cache eliminating latencies.

We implemented this with *OpenMP* built-in nested parallelism, and using a *thread pool*.

#### OpenMp nested parallelism

OpenMP supports nested parallelism when two loops are nested and are both enclosed by a *#pragma parallel* clause. By default nested parallelism is disabled with OpenMP and some implementation do not even support it. To enable nested parallelism, the *OMP_NESTED* environment variable must be set or else a call to *omp_set_nested*() has to be made from the code.

Following is an example of nested parallelism with OpenMP:

```
1 omp_set_dynamic(0);
2 #pragma omp parallel num_threads(2) {
3     #pragma omp parallel num_threads(2)      {
4         do_stuff();
5     }
6 }
```

In the inner *#pragma omp parallel* section, the function would be called 4 times, 2 times by a pair of threads belonging to the same team.

In our code, the outer loops of the $Axpy_{block}$ and $Trsm_{block}$ algorithms were marked as parallel using *#omp pragmas* (the code of the functions was inlined in the outer $Trsm$, $Axpy$ functions). Unfortunately, using OpenMP's nested parallelism didn't bring more efficiency to our parallel implementation; it even did slow it up on 8 threads where the speedup was at its peak level. After inspection with the intel vtune profiler [8] we have discovered that OpenMP creates and destroys threads as the inner loops finishes execution, this introduced great overhead where most of the time was spent on thread creation and destruction.

#### Using a thread pool

To avoid OpenMP's thread creation and destruction when nested parallelism is used, we decided to try a thread pool where threads are created at the beginning of the computation and then they keep waiting for tasks to perform. The tasks are put on a shared message queue. Threads keep probing the queue for available tasks, once a task available, a thread fetches it and executes the given task; it then checks the queue again and so on.

In our parallel $Trsm$ algorithm for example, master threads handle columns of $B$, and in the inner loop (the call to $Axpy_{block}$ and $Trsm_{block}$) they simply put a task on the queue for the threads in the thread pool to perform. This can be described in the following pseudo-code:

```
for(int i=0; i<nb_block_columns_B) do in parallel
  for(j=0; j<nb_block_rows_B)
    queue.add(reduce task of block B[i][j], start from 0 to block_heigt/2)
    queue.add(reduce task of block B[i][j], start from block_heigt/2 to block_heigt)
```

In the above code, the master threads assign the reduction of each block to 2 threads by pushing the corresponding tasks. This should indeed eliminate the overhead introduced by thread creation and destruction as in the OpenMP case. As with nested parallelism with OpenMP, this method didn't provide much efficiency neither. Under more inspection with intel vtune profiler, it was clear that threads synchronization was very important: the threads would endure a context switch each time they can't access to the queue or when the queue is empty. A thread pool synchronization over the queue with *spinlocks* could possibly eliminate the overhead of context switching.

We have used Ronald Kriemann's C++ thread-pool implementation [9] based on posix threads.

## 3.9 Memory utilization

One of the major goals of our implementation is to reduce the memory footprint of the program as much as possible while assuring the best possible efficiency.

As we have noted in 3.3.1, LELA's implementation is not efficient when it comes to memory utilization. In fact, to achieve better performance with LELA, one has to represent matrices' elements over the *double* data type consuming 4 times more memory than if the elements were represented over 16 bits. Considering memory utilization always, another drawback of LELA's implementation is that the temporary sub-matrices used ($A$, $B$, $C$, $D$, $D1$, $D2$, $B1$ and $B2$) are not freed unless the computations are done. Moreover, all the aforementioned sub-matrices are represented in dense format except for $A$ which is represented as sparse matrix.

On the other hand, the original implementation of Sylvain Lachartre [1] uses a technique close to an arena allocator: at the start of the program, a big memory chunk is reserved and all the computations are performed inside the allocated memory. This has the benefit of eliminating dynamic allocations/deallocations overhead during the program's execution. While this technique can lead to

more performance, it has the drawback that the needed memory must be set before the program starts, implying one has to know the amount of memory required or giving the program more memory than what it needs. On the other hand, the memory reserved by the program is not released unless the computations are completed, even if a big part of it is not used at some point, which can prevent other programs from running correctly.

In our implementation we have considered releasing memory as soon as it is not useful anymore. For instance, during the decomposition step of the sub-matrices $A$, $B$, $C$ and $D$, the original matrix $M_0$ is of no use anymore since all the data resides in the sub-matrices now; to this end, out implementation releases the memory of $M_0$ as it is copied to sub-matrices. Another case is when the $Trsm$ step is completed, the sub-matrix $A$ can be released immediately since it has no use after that. In the same way, after the $Axpy$ step, the sub-matrix $C$ can be freed since it is equivalent to 0. In our hybrid implementation involving block matrices and multiline matrices, this becomes very practical since when copying a matrix from one representation to another, the original one is released on the go.

All our allocations/deallocation are done through the standard $malloc/free$ functions, or using $posix\_memealign$ when aligned memory is needed (c.f. 3.1). One difficulty in achieving a perfect smooth memory management is memory fragmentation: for example, if the rows of a matrix $A$ are interleaved in memory with those of $B$, then freeing $A$ would not necessarily free it's underlying memory since $B$ is still occupying the memory pages where $A$ was mapped. Notice that the matrix decomposition process causes this fragmentation unless the memory of sub-matrices is allocated separately which would require two passes: one for analysis and allocation and the second for the actual decomposition.

Besides releasing memory on the go, our new method that uses the new ordering of operations on echelon forms computations has a relatively less memory footprint than the standard Faugère-Lachartre algorithm in addition to being more efficient. This is due to the fact that the sub-matrix $B$ becomes denser after the $Trsm$ step (generally doubles its density); this step is never performed with the new method which means that the memory of $B$ does not increase throughout the program lifetime.

### 3.9.1 Distributed parallel processing of big matrices

When the size of the matrices becomes very important (from dozens of *gigabytes* to even several *terabytes*), performing the Gaussian elimination on a single machine becomes infeasible. One way to achieve this is through distributed computing. Each node would be assigned the matrix $A$ which is generally sparse and hence have a small size (with a density from less than 1% to ~5%) and a column of blocks $B_i$ to perform the $Trsm$ step and a blocks' column $D_i$ for the $Axpy$ step. A master node can perform the decomposition process ahead, or else all the nodes can perform the decomposition process locally and keep only the data they are assigned to use. Generally the decomposition process's running time is minimal compared to the actual $Axpy$, $Trsm$ operations.

The table 3.9 shows the proportion of the running time of each step of the Faugère-Lachartre algorithm compared to the total running tume.

The decomposition process in our implementation as shown in the table 3.9 takes generally from 1% to 4% of the time, which means that each node can perform this step separately in a distributed environment. The reconstruction in our implementation takes more time on sparse matrices as shown in the case of the katsura 16 matrices. Our *Indexer* is not very optimized which means that these times can be reduced even further.

### Memory utilization of the new implementation vs the original implementation

We show in the table 3.10 the amount of memory used by our implementation and the original implementation of Sylvain Lachartre. Considering our implementation, we show the memory usage of the standard Faugère-Lachartre method and the memory usage of the new method with the modified

Table 3.9: Running time of the different steps of Faugère-Lachartre

| matrix | density | decomposition | *Trsm* | *Axpy* | *Gauss* | reconstruction |
|---|---|---|---|---|---|---|
| Minrank 996/mat3 | 32.84% | 1.06% | 77.13% | 19.51% | 0.62% | 1.47% |
| Minrank 996/mat4 | 27.43% | 1.13% | 88.06% | 9.01% | 0.13% | 1.51% |
| Minrank 996/mat5 | 24.07% | 1.18% | 93.54% | 3.43% | 0.02% | 1.60% |
| Mr10/mat2 | 48.68% | 2.70% | 25.91% | 52.77% | 16.10% | 2.09% |
| Mr10/mat3 | 35.54% | 1.62% | 37.25% | 50.40% | 8.54% | 1.92% |
| Kat16/mat5 | 1.40% | 3.29% | 71.89% | 6.59% | 2.64% | 14.21% |
| Kat16/mat6 | 1.24% | 2.55% | 73.99% | 7.79% | 3.40% | 11.10% |

ordering of operations. For Lachartre's original implementation, the minimum required amount of memory allocated is shown along with the actual memory used during the computations.

Table 3.10: Memory utilization in MB of the new sequential implementation vs the original Lachartre's version

| matrix | new implementation | | Lachartre's version | |
|---|---|---|---|---|
| | new method | standard method | reserved memory | actual used memory |
| Minrank 996/mat3 | **1469** | 1683 | 2048 | 2289 |
| Minrank 996/mat4 | **2793** | 3005 | 4096 | 4701 |
| Minrank 996/mat5 | **4565** | 5568 | 8192 | 7637 |
| kat13/mat7 | **466** | 1402 | 1024 | 806 |
| kat13/mat8 | **498** | 1417 | 1024 | 883 |
| Kat16/mat10 | **8433** | 33130 | 16384 | 15012 |

From table 3.10, we can see that our new method implementation has a less memory footprint than the original implementation of Sylvain Lachartre on dense matrices ($Minrak996$ and $Mr10$). However, on sparse matrices ($Katsura$ problems) our implementation of the standard method has a larger memory footprint; this is due to the fact that when decomposing the matrices, the rows of blocks of $A$ and $B$ are interleaved, the same happens with those of $C$ and $D$ (no prior separate allocation is made) which makes deallocations useless since the memory pages are still occupied all along the program's lifetime. We can notice however that the new ordering of operations uses less memory than both our implementation of the standard method and the original Lachartre's version.

**Note**: the memory footprint of our implementation can be reduced to the half. Indeed, at the reconstruction step, our program doubles its memory usage which means that it doesn't succeed in releasing the memory on the go during this step. This must be investigated more, since it can reduce the memory usage of our implementation easily by 50%. The figure 3.9.1 shows the evolution of the memory usage during the program's lifetime.

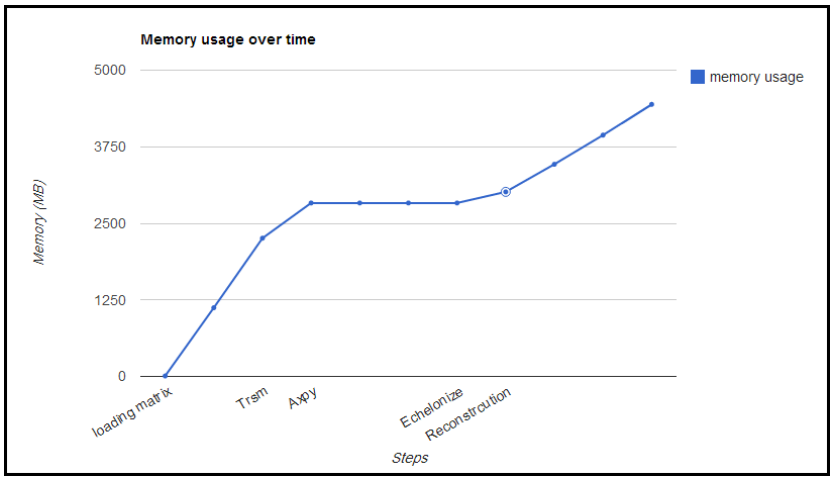


Figure 3.9.1: Memory utilization with time

We can see clearly how the memory grows starting with the reconstruction step. To avoid this, a more rigorous memory allocation strategy must be considered in our *Indexer* in order to avoid fragmentation when decomposing/reconstructing matrices.

# Conclusion and future work

In this report we have presented the Faugère-Lachartre method for computing row echelon forms and reduced row echelon forms for matrices issued from Gröbner bases computations. After presenting the standard Gaussian and Gauss-Jordan elimination algorithms, we have shown how the Structured Gaussian Elimination method can be very efficient compared to the naïve methods by taking advantage of the reduced writing memory area. A detailed description of the Faugère-Lachartre method then followed, where we discussed the block version of the algorithm and the parallelization of its overall steps.

The modifications to the standard Faugère-Lachartre to fit with our multiline data structure were then discussed in the contributions part. We introduced a new parallel algorithm for the structured Gaussian elimination method and proposed a new ordering of operations for the Faugère-Lachartre algorithm that outperforms the standard implementation in both CPU performance and memory usage.

The performance of the implementation and the efficient use of memory were the main goals of this internship. We demonstrated how we achieved the best performance in our implementation by making use of the multiline data structure and the blocks decomposition of matrices which allowed an intuitive parallelization of most of the parts of the algorithm. The new ordering of operations is then detailed where we showed that this new method is indeed more efficient compared to the standard one. The memory footprint of our implementation is then investigated by comparing it with the Lachartre's original version; we showed that the new method is always more efficient on memory usage compared to both Lachartre's version and our implementation of the standard method. We also discussed briefly the possibility of distributing the computations in case of very large matrices.

Although our implementation of Faugère-Lachartre can be 300 times faster than a naïve Gaussian elimination, more enhancements are required in order to achieve the best performance and memory efficiency; we list in what follows the basic points to address in a future improvement of our implementation:

- Memory management: as we have mentioned in 3.9.1, during the reconstruction step, the memory is almost doubled which means that our *Indexer* is not able to release memory at this step, fixing this would require that the allocation of the sub-matrices be separated in different memory areas.
- Along the previous point, assuring a better memory allocation for the standard implementation which is not very efficient concerning memory usage especially on relatively sparse matrices.
- Investigating scalability issues for the parallel implementation.
- Implementing a more fine-grained version of our parallel structured Gaussian elimination algorithm 2.2.
- Considering developing a distributed version which should not require a lot of code refactoring compared to our parallel implementation.
- Our implementation seems to have a strange behavior on very sparse matrices (c.f. Appendix A) (of density less than 5% as in some matrices of the *Katsura n* problem); we notice a clear drop in performance over these matrices. This should be investigated further to identify the possible reasons.

- The decomposing/reconstruction steps can be greatly improved in the sequential version. These steps are also partially parallelized which means they can be optimized further by more parallelism.
- Considering the use of more efficient memory management libraries that are shown to have better performance than *malloc*, especially on multithreading programs, like *TC malloc* from Google or *Threading Building Blocks* from intel.

# Appendix A

# Experimental Results and comparisons with existing Faugère-Lachartre implementations

### Row echelon form computation

We show in the following tables the running time (in seconds) of computing a **row echelon form** [1] of matrices from different types of problems. Our implementation is marked as "*New program.*" We list the timings of the new method we have proposed and the timings of the standard Faugère-Lachartre along with their corresponding parallel versions when the running time relatively long. We compare these timings with Lachartre's original implementation (sequential and parallel).

For running time comparisions with LELA's implementation, refer to 3.3.1.

The figure A.0.1 shows the speedup achieved by our implementation using the new ordering of operations compared to Lachartre's original implementation. The measures are made on the total running time required to solve each problem. We notice that our implementation outperforms the original version on most of the problems, except *Katsura 11* and *Katsura 12* which are very small compared to the rest of the problems.

[1] Notice that this is not a ***reduced*** row echelon form

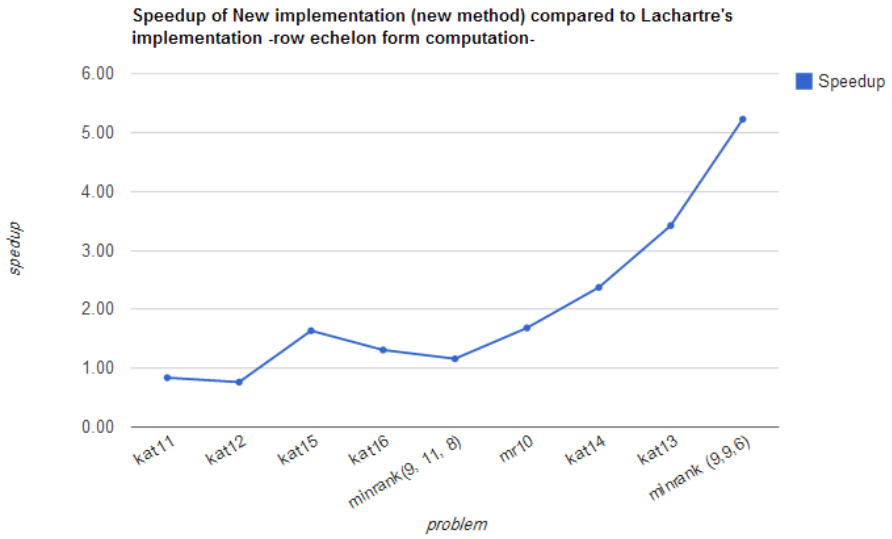


Figure A.0.1: Speedup of the new implementation of Faugère-Lachartre (using the new ordering of operations) compared to Lachartre's original implementation

| matrix | New program | | Lachartre's version |
|---|---|---|---|
| | New method | Old method | |
| mat1 | 0.01 | 0.02 | 0.01 |
| mat2 | 0.14 | 0.18 | 0.11 |
| mat3 | 0.76 | 0.76 | 0.52 |
| mat4 | 2.21 | 2.28 | 1.33 |
| mat5 | 2.75 | 3.22 | 1.96 |
| mat6 | 4.03 | 3.80 | 2.79 |
| mat7 | 2.91 | 3.13 | 2.15 |
| mat8 | 1.24 | 1.86 | 1.33 |
| mat9 | 0.37 | 1.28 | 1.09 |
| mat10 | 0.14 | 1.00 | 0.95 |
| Total | 14.56 | 17.53 | ***12.24*** |

Table A.1: Katsura 11

| matrix | New program | | | | | | Lachartre's version | | |
|---|---|---|---|---|---|---|---|---|---|
| | New method | | | Old method | | | | | |
| | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel |
| mat1 | 0.00 | 0.00 | 0.73 | 0.00 | 0.01 | 0.44 | 0.01 | 0.01 | 1 |
| mat2 | 0.03 | 0.02 | 1.46 | 0.07 | 0.04 | 1.91 | 0.04 | 0.01 | 4.00 |
| mat3 | 0.36 | 0.13 | 2.74 | 0.85 | 0.30 | 2.80 | 0.45 | 0.10 | 4.50 |
| mat4 | 0.47 | 0.25 | 1.87 | 2.56 | 0.95 | 2.70 | 1.39 | 0.31 | 4.48 |
| mat5 | 3.27 | 0.95 | 3.45 | 9.92 | 2.55 | 3.89 | 5.72 | 1.14 | 5.02 |
| mat6 | 36.26 | 6.28 | 5.77 | 53.49 | 11.90 | 4.49 | 33.14 | 6.32 | 5.24 |
| mat7 | 109.80 | 18.98 | 5.79 | 167.30 | 33.67 | 4.97 | 117.04 | 21.69 | 5.40 |
| mat8 | 1.97 | 3.12 | 0.63 | 31.56 | 8.04 | 3.93 | 19.97 | 4.15 | 4.81 |
| mat9 | 66.51 | 13.58 | 4.90 | 148.60 | 31.29 | 4.75 | 96.42 | 18.65 | 5.17 |
| mat10 | 97.22 | 18.32 | 5.31 | 230.80 | 45.17 | 5.11 | 144.28 | 26.87 | 5.37 |
| mat11 | 75.83 | 16.16 | 4.69 | 250.50 | 50.28 | 4.98 | 142.95 | 26.10 | 5.48 |
| mat12 | 37.21 | 9.20 | 4.04 | 245.60 | 49.75 | 4.94 | 135.13 | 24.66 | 5.48 |
| mat13 | 14.31 | 6.96 | 2.06 | 232.80 | 50.52 | 4.61 | 129.93 | 23.79 | 5.46 |
| mat14 | 7.45 | 5.27 | 1.41 | 229.30 | 47.57 | 4.82 | 129.25 | 23.51 | 5.50 |
| mat15 | 6.17 | 4.51 | 1.37 | 224.00 | 47.11 | 4.75 | 128.59 | 23.53 | 5.46 |
| **Total** | ***456.86*** | ***103.74*** | 4.40 | 1827.36 | 379.15 | 4.82 | 1084.31 | 200.83 | 5.40 |

Table A.2: Katsura 14

| matrix | New program | | | | | | Lachartre's version | | |
|---|---|---|---|---|---|---|---|---|---|
| | New method | | | Old method | | | | | |
| | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel |
| mat1 | 0.00 | 0.00 | 0.73 | 0.01 | 0.01 | 0.94 | 0.01 | 0.01 | 1 |
| mat2 | 0.05 | 0.03 | 1.63 | 0.12 | 0.06 | 1.86 | 0.07 | 0.01 | 7.00 |
| mat3 | 0.85 | 0.23 | 3.69 | 1.62 | 0.70 | 2.31 | 0.84 | 0.18 | 4.67 |
| mat4 | 1.33 | 0.54 | 2.44 | 6.22 | 1.78 | 3.49 | 3.21 | 0.72 | 4.46 |
| mat5 | 5.34 | 1.53 | 3.50 | 22.67 | 5.54 | 4.09 | 12.14 | 2.51 | 4.84 |
| mat6 | 120.10 | 21.41 | 5.61 | 171.20 | 34.32 | 4.99 | 105.89 | 19.48 | 5.44 |
| mat7 | 306.10 | 49.05 | 6.24 | 550 | 105.50 | 5.21 | 312.10 | 52.68 | 5.92 |
| mat8 | 506.10 | 85.65 | 5.91 | 851.90 | 159.40 | 5.34 | 373.07 | 74.73 | 4.99 |
| mat9 | 12.06 | 8.06 | 1.50 | 376 | 82.86 | 4.54 | 188.03 | 34.60 | 5.43 |
| mat10 | 690.90 | 114.30 | 6.04 | 1336 | 245.50 | 5.44 | 539.47 | 104.71 | 5.15 |
| mat11 | 644.80 | 107.10 | 6.02 | 1734 | 301.60 | 5.75 | 586.23 | 115.46 | 5.08 |
| mat12 | 425.80 | 79.68 | 5.34 | 1778 | 322.70 | 5.51 | 616.35 | 111.36 | 5.53 |
| mat13 | 174.80 | 42.99 | 4.07 | 1647 | 311.30 | 5.29 | 551.98 | 106.14 | 5.20 |
| mat14 | 59.86 | 23.76 | 2.52 | 1605 | 304.20 | 5.28 | 536.08 | 103.53 | 5.18 |
| mat15 | 29.14 | 23.82 | 1.22 | 1568 | 291.30 | 5.38 | 558.17 | 99.86 | 5.59 |
| mat16 | 23.44 | 19.71 | 1.19 | 1565 | 299.10 | 5.23 | 532.53 | 101.44 | 5.25 |
| Total | ***3000.67*** | ***577.86*** | 5.19 | 13212.73 | 2465.87 | 5.36 | 4916.16 | 927.41 | 5.30 |

Table A.3: Katsura 15

| matrix | New program | | | | | | Lachartre's version | | |
|---|---|---|---|---|---|---|---|---|---|
| | New method | | | Old method | | | | | |
| | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel |
| mat1 | 0.00 | 0.01 | 0.38 | 0.01 | 0.02 | 0.49 | 0.00 | 0.00 | #DIV/0! |
| mat2 | 0.06 | 0.06 | 1.08 | 0.18 | 0.08 | 2.29 | 0.09 | 0.02 | 4.50 |
| mat3 | 1.19 | 0.49 | 2.44 | 3.67 | 0.87 | 4.23 | 1.64 | 0.33 | 4.97 |
| mat4 | 2.18 | 0.78 | 2.79 | 12.85 | 2.72 | 4.73 | 6.13 | 1.34 | 4.57 |
| mat5 | 9.80 | 2.55 | 3.84 | 54.98 | 10.88 | 5.05 | 27.74 | 5.43 | 5.11 |
| mat6 | 47.56 | 9.24 | 5.15 | 180.90 | 34.54 | 5.24 | 114.42 | 20.88 | 5.48 |
| mat7 | 1164 | 185.40 | 6.28 | 1533 | 257.70 | 5.95 | 774.07 | 150.05 | 5.16 |
| mat8 | 1158 | 186.50 | 6.21 | 1778 | 287.70 | 6.18 | 813.49 | 163.15 | 4.99 |
| mat9 | 3,512 | 633 | 5.55 | 7573 | 1078 | 7.03 | 3028.35 | 616.88 | 4.91 |
| mat10 | 30.54 | 22.92 | 1.33 | 1378 | 245.80 | 5.61 | 659.93 | 130.44 | 5.06 |
| mat11 | 4350 | 685.20 | 6.35 | 9795 | 1460 | 6.71 | 3649.61 | 713.34 | 5.12 |
| mat12 | 4920 | 848.70 | 5.80 | 10290 | 1528 | 6.73 | 3212.97 | 625.13 | 5.14 |
| mat13 | 3823 | 624.20 | 6.12 | 11600 | 1689 | 6.87 | 3228.08 | 624.19 | 5.17 |
| mat14 | 2384 | 410.90 | 5.80 | 11470 | 1695 | 6.77 | 3008.39 | 581.88 | 5.17 |
| mat15 | 788.30 | 170.20 | 4.63 | 10970 | 1608 | 6.82 | 2859.91 | 548.89 | 5.21 |
| mat16 | 241.70 | 87.59 | 2.76 | 10680 | 1594 | 6.70 | 2768.60 | 527.52 | 5.25 |
| mat17 | 104.70 | 71.94 | 1.46 | 10620 | 1569 | 6.77 | 2758.06 | 526.58 | 5.24 |
| mat18 | 83.14 | 69.03 | 1.20 | 10600 | 1581 | 6.70 | 2745.39 | 517.85 | 5.30 |
| Total | ***22620*** | ***4008.71*** | 5.64 | 98539.59 | 14642.3 | 6.73 | 29656.87 | 5753.90 | 5.15 |

Table A.4: Katsura 16

| matrix | New program | | | | | | Lachartre's version | | |
|---|---|---|---|---|---|---|---|---|---|
| | New method | | | Old method | | | | | |
| | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel |
| mat1 | 6.18 | 1.08 | 5.70 | 6.24 | 1.25 | 4.98 | 6.40 | 6.32 | 1.01 |
| mat2 | 19.06 | 3.51 | 5.43 | 22.81 | 4.01 | 5.68 | 18.41 | 3.65 | 5.04 |
| mat3 | 66.69 | 11.10 | 6.01 | 101.30 | 16.38 | 6.18 | 84.38 | 12.97 | 6.51 |
| mat4 | 166.60 | 28.20 | 5.91 | 280.90 | 43.84 | 6.41 | 250.25 | 39.02 | 6.41 |
| mat5 | 317.10 | 52.22 | 6.07 | 571.60 | 85.40 | 6.69 | 496.52 | 78.06 | 6.36 |
| mat6 | 530.80 | 82.27 | 6.45 | 1114 | 165.80 | 6.72 | 1010.39 | 168.36 | 6.00 |
| mat7 | 548.10 | 86.48 | 6.34 | 1466 | 214.20 | 6.84 | 1447.38 | 246.16 | 5.88 |
| mat8 | 295.10 | 53.91 | 5.47 | 1399 | 200.50 | 6.98 | 1475.26 | 223.58 | 6.60 |
| mat9 | 143 | 27.87 | 5.13 | 1307 | 190.70 | 6.85 | 1413.65 | 204.41 | 6.92 |
| mat10 | 65.78 | 17.28 | 3.81 | 1254 | 185.10 | 6.77 | 1376.96 | 199.09 | 6.92 |
| mat11 | 34.85 | 12.68 | 2.75 | 1232 | 182.40 | 6.75 | 1398.04 | 194.79 | 7.18 |
| mat12 | 24.85 | 14.08 | 1.76 | 1228 | 181.90 | 6.75 | 1341.77 | 192.75 | 6.96 |
| mat13 | 22.26 | 10.52 | 2.12 | 1225 | 181.30 | 6.76 | 1390.76 | 195.59 | 7.11 |
| **Total** | ***2240.37*** | ***401.21*** | 5.58 | 11207.85 | 1652.78 | 6.78 | 11710.17 | 1764.75 | 6.64 |

Table A.5: Minrank Minors (9, 9, 6)

| matrix | New program | | | | | | Lachartre's version | | |
|---|---|---|---|---|---|---|---|---|---|
| | New method | | | Old method | | | | | |
| | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel |
| mat1 | 67.01 | 12.37 | 5.42 | 72.64 | 11.63 | 6.25 | 52.62 | 51.54 | 1.02 |
| mat2 | 168.50 | 27.73 | 6.08 | 210.10 | 33.97 | 6.18 | 174.27 | 35.84 | 4.86 |
| mat3 | 384.40 | 59.54 | 6.46 | 658.70 | 99.45 | 6.62 | 552.58 | 94.66 | 5.84 |
| mat4 | 1018 | 154.90 | 6.57 | 1662 | 238.50 | 6.97 | 1445.49 | 241.06 | 6.00 |
| mat5 | 1647 | 246.60 | 6.68 | 2732 | 386.10 | 7.08 | 2404.22 | 388.06 | 6.20 |
| mat6 | 2228 | 333.10 | 6.69 | 3134 | 452.60 | 6.92 | 2656.08 | 497.38 | 5.34 |
| mat7 | 2679 | 387.50 | 6.91 | 3257 | 466.00 | 6.99 | 2733.10 | 689.12 | 3.97 |
| mat8 | 1735 | 259.50 | 6.69 | 1955 | 281.80 | 6.94 | 1667.11 | 580.84 | 2.87 |
| mat9 | 577.20 | 83.28 | 6.93 | 625.70 | 88.03 | 7.11 | 543.82 | 172.05 | 3.16 |
| mat10 | 72.16 | 11.35 | 6.36 | 80.21 | 12.57 | 6.38 | 63.57 | 21.73 | 2.93 |
| mat11 | 9.16 | 1.96 | 4.66 | 10.13 | 2.02 | 5.02 | 8.56 | 3.26 | 2.63 |
| mat12 | 0.97 | 0.24 | 3.97 | 1.08 | 0.29 | 3.67 | 0.86 | 0.40 | 2.15 |
| mat13 | 0.10 | 0.06 | 1.66 | 0.10 | 0.04 | 2.54 | 0.08 | 0.04 | 2.00 |
| mat14 | 0.01 | 0.01 | 1.05 | 0.01 | 0.01 | 1.02 | 0.001 | 0.01 | 0.3 |
| mat15 | 0.001 | 0.001 | 0.48 | 0.001 | 0.001 | 0.46 | 0.001 | 0.02 | 0.04 |
| mat16 | 0.001 | 0.001 | 0.32 | 0.001 | 0.001 | 0.36 | 0.001 | 0.001 | 1 |
| **Total** | ***10586*** | ***1578*** | 6.71 | 14398.67 | 2073 | 6.95 | 12302.36 | 2776.01 | 4.43 |

Table A.6: Minrank Minors (9, 11, 8) spec1

| matrix | New program | | | | | | Lachartre's version | | |
|---|---|---|---|---|---|---|---|---|---|
| | New method | | | Old method | | | | | |
| | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel | seq | parallel (8) | speedup seq / parallel |
| mat1 | 30.9 | 7.16 | 4.31 | 31.61 | 6.66 | 4.75 | 24.06 | 23.64 | 1.02 |
| mat2 | 104.7 | 18.32 | 5.72 | 131.9 | 22.57 | 5.84 | 102.91 | 24.32 | 4.23 |
| mat3 | 318.7 | 50.24 | 6.34 | 535.9 | 82.15 | 6.52 | 489.95 | 73.61 | 6.66 |
| mat4 | 1007 | 157.70 | 6.39 | 1778 | 264.20 | 6.73 | 1722 | 278.95 | 6.17 |
| mat5 | 1898 | 284.90 | 6.66 | 3675 | 531.60 | 6.91 | 3387.1 | 678.60 | 4.99 |
| mat6 | 4466 | 654.60 | 6.82 | 8231 | 1162 | 7.08 | 7461.61 | 1381.08 | 5.40 |
| **Total** | ***7825.30*** | ***1172.92*** | 6.67 | 14383.41 | 2069.18 | 6.95 | 13187.63 | 2460.20 | 5.36 |

Table A.7: Mr10

| matrix | New program | | Lachartre's version |
|---|---|---|---|
| | New method | Old method | |
| mat1 | 0.02 | 0.04 | 0.02 |
| mat2 | 0.33 | 0.41 | 0.27 |
| mat3 | 2.44 | 2.70 | 1.62 |
| mat4 | 9.99 | 8.31 | 6.23 |
| mat5 | 20.63 | 18.06 | 13.14 |
| mat6 | 26.40 | 22.85 | 17.06 |
| mat7 | 27.59 | 24.50 | 17.90 |
| mat8 | 16.11 | 19.23 | 12.31 |
| mat9 | 6.05 | 9.44 | 7.33 |
| mat10 | 1.54 | 6.47 | 5.26 |
| mat11 | 0.51 | 6.47 | 4.36 |
| Total | 111.61 | 118.48 | ***85.50*** |

(a) Katsura 12

| matrix | New program | | Lachartre's version |
|---|---|---|---|
| | New method | Old method | |
| mat1 | 0.02 | 0.04 | 0.02 |
| mat2 | 0.16 | 0.39 | 0.21 |
| mat3 | 0.94 | 2.13 | 1.27 |
| mat4 | 3.54 | 7.70 | 5.33 |
| mat5 | 8.41 | 17.13 | 12.55 |
| mat6 | 12.65 | 31.56 | 20.53 |
| mat7 | 11.67 | 34.59 | 26.81 |
| mat8 | 7.20 | 34.96 | 27.92 |
| mat9 | 3.15 | 33.66 | 26.83 |
| mat10 | 1.74 | 33.62 | 26.43 |
| mat11 | 1.45 | 38.01 | 26.38 |
| Total | ***50.92*** | 233.80 | 174.28 |

(b) Katsura 13

# Bibliography


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

[2] Jean-Charles Faugère and Sylvain Lachartre. Parallel Gaussian Elimination for Gröbner bases computations in finite fields. In M. Moreno-Maza and J.L. Roch, editors, Proceedings of the 4th International Workshop on Parallel and Symbolic Computation, PASCO '10, pages 89-97, New York, NY, USA, July 2010. ACM.

[3] J.-C. Faugère. A New Efficient Algorithm for Computing Groebner bases (F4). Journal of Pure and Applied Algebra , 139(1-3):61–88, 1999.

[4] J.-C. Faugère. A new efficient algorithm for computing Groebner bases without reduction to zero F5. In Proceedings of the ACM SIGSAM International Symposium on Symbolic and Algebraic Computation, 2002.

[5] Bradford Hovinen : LELA Library for Exact Linear Algebra http://www.singular.uni-kl.de/lela/

[6] FGb: A C library for the computation of Groebner bases (http://www-polsys.lip6.fr/~jcf/Software/FGb/index.html)

[7] M. Albrecht and G. Bard : M4RI a library for fast arithmetic with dense matrices over $F_2$ http://m4ri.sagemath.org/

[8] Intel® VTune™ Amplifier XE: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/

[9] Ronald Kriemann - Implementation and Usage of a Thread Pool based on POSIX Threads: http://www.hlnum.org/english/projects/tools/threadpool/

[10] Agner Fog - Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms: http://www.agner.org/optimize/

[11] Ulrich Drepper: What Every Programmer Should Know About Memory : http://www.akkadia.org/drepper/cpumemory.pdf

[12] W. Stein et al. Sage Mathematics Software. http://www.sagemath.org.

[13] 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.

[14] Automatically Tuned Linear Algebra Software (ATLAS) http://math-atlas.sourceforge.net/

[15] Severin Neumann: Parallel reduction of matrices in Grobner bases computations, Proceedings, 14th International Workshop, CASC 2012, Maribor, Slovenia, LNCS Volume 7442, p260