# Parallel Structured Gaussian Elimination for the Number Field Sieve

Charles Bouillaguet, Paul Zimmermann

▶ **To cite this version:**

Charles Bouillaguet, Paul Zimmermann. Parallel Structured Gaussian Elimination for the Number Field Sieve. Mathematical Cryptology, In press. hal-02098114v1

**HAL Id: hal-02098114**
**https://inria.hal.science/hal-02098114v1**

Submitted on 12 Apr 2019 (v1), last revised 5 Jan 2021 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Parallel Structured Gaussian Elimination for the Number Field Sieve

Charles Bouillaguet
Université de Lille, France
charles.bouillaguet@univ-lille.fr

Paul Zimmermann
Université de Lorraine, CNRS, Inria, LORIA, F-54000
Nancy, France
paul.zimmermann@inria.fr

## ABSTRACT

This article describes a parallel algorithm for the Structured Gaussian Elimination step of the Number Field Sieve (NFS). NFS is the best known method for factoring large integers and computing discrete logarithms.

State-of-the-art algorithms for this kind of partial sparse elimination, as implemented in the CADO-NFS software tool, were unamenable to parallel implementations. We therefore designed a new algorithm from scratch with this objective and implemented it using OpenMP. The result is not only faster sequentially, but scales reasonably well: using 32 cores, the time needed to process two landmark instances went down from 38 minutes to 20 seconds and from 6.7 hours to 2.3 minutes, respectively.

## 1 INTRODUCTION

The Number Field Sieve (NFS) is the best known algorithm to factor large integers or to compute discrete logarithm over large prime fields. It was used to factor RSA-768 in 2009 [26], and to compute a 768-bit discrete logarithm in 2017 [27], which are the current world records. The NFS algorithm splits into four main phases: sieving, filtering, linear algebra, and post-processing [7]. The filtering phase decomposes itself into three steps: eliminating duplicates, eliminating singletons and excess — also called the "purge" step in the NFS community —, and a last Structured Gaussian Elimination (SGE) step — also called "merge". The SGE step takes as input a sparse matrix, and produces a smaller but denser matrix by combining rows of the input matrix. The SGE step drastically reduces the cost of the linear algebra step, which is usually performed with iterative methods like block Lanczos or block Wiedemann.

Unlike in numerical computations, where the coefficients of the matrix are floating-point numbers, here the coefficients are exact, either in GF(2) for integer factorization (IF), or in GF($\ell$) for discrete logarithms (DL), where GF($k$) is the finite field with $k$ elements. Thus there is no issue of numerical accuracy like partial or complete pivoting, condition number, etc.

The Structured Gaussian Elimination step takes as input a large and (nearly) square sparse matrix $M$ with integer coefficients. In the integer factorization case, one is looking for an element of the *left null space* of the matrix $M$ over GF(2), i.e., a linear combination of the rows that sums to the zero vector over GF(2). When such a linear combination is found, combining the corresponding rows yields an identity $x^2 = y^2 \bmod N$ where $N$ is the number to factor, then $\gcd(x - y, N)$ gives a non-trivial factor of $N$ with probability at least 1/2. In the discrete logarithm setting, one is looking for an element of the *right null space* of $M$ over GF($\ell$), where $\ell$ is a prime number of typically 160 or more bits. Once a solution is found, it gives the so-called *virtual logarithms*.

In both cases, the Structured Gaussian Elimination process transforms the initial matrix $M$ by performing a partial LU factorization: the columns are permuted for sparsity and some of them are eliminated. The SGE step factors $M$ into

$$PMQ = \begin{pmatrix} I & \\ A & I \end{pmatrix} \begin{pmatrix} I & \\ & M' \end{pmatrix} \begin{pmatrix} U & B \\ & I \end{pmatrix}, \quad (1)$$

where $P$ (resp. $Q$) is a permutation of the rows (resp. columns).

The matrix $U$ is triangular and invertible (it corresponds to the columns that have been eliminated), and $M'$ is the Schur complement, the new (smaller) matrix that will be given to an iterative algorithm in the subsequent linear algebra step. In the integer factorization case (where we want $xM = 0$), we first solve $yM' = 0$ and then, neglecting permutations, a solution of $xM = 0$ is $x = (-yA \mid y)$. In the discrete logarithm setting (where we want $Mx = 0$) once we have found $y$ such that $M'y = 0$, it remains to solve $Ux + By = 0$ to obtain $M(x \mid y)^t = 0$.

The goal of Structured Gaussian Elimination is to eliminate as many rows/columns as possible, i.e., make the dimensions of $M'$ as small as possible, while keeping its density below a given threshold.

*Related Problems.* In the context of sparse direct solvers, the problem is usually to find fill-reducing permutations in order to minimize the number of non-zero entries in the factors. Our setting is related but slightly different: in SGE the factors are essentially discarded and instead we seek to minimize fill-in in the remaining unfactored part $M'$. Of course, because non-zero entries of $M'$ will also be non-zero in the factors (neglecting numerical cancellation), full sparse factorizations also try to keep $M'$ as sparse as possible. This is precisely what Markowitz pivoting [33] does greedily.

Yannakakis [45] proved that it is NP-hard to compute the optimal permutation $P$ yielding the sparsest Cholesky factorization $PMP^t = LL^t$. Numerous heuristics have been developed to find good orderings, such as nested dissection [19] or (approximate) minimum degree [1].

In order to factor an unsymmetric matrix using a $PMQ = LU$ factorization, one possibility consists in obtaining a good *a priori* column ordering $Q$ and then choosing $P$ during the numerical factorization to account for pivoting and maintaining numerical stability. A possible way to find $Q$ consists in finding a good symmetric ordering for the Cholesky factorization of $M^tM$. Indeed, ignoring numerical cancellation, the non-zero pattern of $U$ is a subset the non-zero pattern of the Cholesky factor of $M^tM$, regardless of the choice of pivots. We refer to Davis' textbook [14] for more details.

All these techniques and machinery cannot be applied directly to our setting for several reasons: the randomness of our matrices trumps these heuristics; in particular, $M^tM$ will be significantly denser than $M$; we may safely ignore numerical stability issues;

numerical cancellation, which is ignored by these techniques, plays a significant role in SGE.

While numerical cancellations are often assumed not to occur with floating-point coefficients, they are very frequent with coefficients in a finite field, in particular modulo 2, where the only non-zero value is 1 and where $1 + 1 = 0$. Pivots therefore have to be chosen and eliminated in a way that minimizes fill-in, taking numerical cancellation into account. Thus, SGE is inherently "right-looking" (access to $M'$ is needed). In fact, two kinds of cancellation occur:

- when we add two rows to eliminate a non-zero coefficient in some column $j$, it might be that another column $j'$ also has non-zero coefficients in those two rows, and that these coefficients do cancel. In the IF case, it means that column $j$ has two "1" in the corresponding rows, and same for $j'$;
- a *catastrophic cancellation* can occur, when two rows of the current matrix are linear combinations of the *same row* of the initial matrix. In such a case, it might occur that *all* non-zero coefficients in the initial row do cancel.

In the same vein, Bouillaguet, Delaplace and Voge propose in [5] a greedy parallel algorithm to compute the rank of large sparse matrices over GF($p$), a problem which is very similar to SGE. Their main idea is to select a priori a set of "structural" pivots that cannot vanish because of numerical cancellation.

Structured Gaussian Elimination can be seen as a form of ILU (Incomplete LU factorization), often used as a preconditioner in numerical iterative methods. In ILU, the idea is to perform a *partial* sparse LU factorization of the input matrix $M$, by storing only certain non-zero entries in $L$ and $U$ (for instance, those already present in the input matrix, or those above a certain threshold). Then $LU \approx M$ and the system $Mx = y$ is equivalent to $\left(U^{-1}L^{-1}M\right) x = y'$ (with $LUy' = y$). The intention is that the new system is better conditioned.

*Previous Work.* In [29], Lamacchia and Odlyzko introduce Structured Gaussian Elimination in the context of both integer factorization and discrete logarithms. They propose to split the matrix into *heavy* and *light* columns. Rows which have only one non-zero element in the light part of the matrix are combined with other rows sharing this column. This process will clearly decrease the weight of the light part. However, this algorithm has the drawback that it requires some heuristic to define heavy columns (which grow throughout the process). In [38], Pomerance and Smith propose a very similar method, using different terms (*inactive* and *active* columns). As in [29], the number of heavy/inactive columns grows until the active/light part of the matrix collapses. In [8, 9], Cavallar introduces the term "merge" and discusses pivoting strategies, including the greedy heuristic known as Markowitz pivoting in numerical computations. She introduces the idea of finding an optimal sequence of operations to eliminate a column, using the computation of a minimum spanning tree. Markowitz pivoting was also used by Joux and Lercier [24], who discuss algorithmic details for pivot selection. Their algorithm requires to keep two copies of the matrix, one by rows, and one by columns. They also need to maintain a binary tree containing the best pivots.

*Contributions.* The main contribution of this article is a first parallel algorithm for the SGE step. In addition, we demonstrate on two record-size examples that this parallel algorithm scales well with the number of threads, and behaves better as the classical algorithm implemented in the reference CADO-NFS implementation [41], both in terms of size of the output matrices, and in terms of computing time, even sequentially.

This article assumes that the SGE algorithm is run on one computing node with several cores, and both the input and output matrices fit into the main memory of that node.

*Organization of the article.* After Section 2 which gives some background about the Number Field Sieve, and defines the notations and benchmarks used throughout the article, we recall in Section 3 the classical algorithm used for the SGE step. The parallel algorithm is detailed in Section 4. Section 5 compares our implementation of the parallel algorithm with the single-thread implementation of CADO-NFS on two record-size examples (RSA-512 and DL-1024).

## 2 THE NUMBER FIELD SIEVE

This section presents some background to make the article self-contained, and provides some "NFS glossary" so that the next sections can be read without any NFS-specific knowledge.

*Background.* The Number Field Sieve (NFS) was invented by John Pollard in 1988, and later improved several times. In short, NFS can either factor large integers into prime factors (the difficulty of that problem is the basis of the RSA cryptosystem, where a public key $n$ is the product of two large secret prime numbers $p$ and $q$: $n = pq$), or compute discrete logarithms modulo large prime numbers, which is the basis of the Diffie-Hellman key-exchange. In this latter case, one is given for example a large prime number $p$, a generator $g$ of the multiplicative group of integers modulo $p$, an element $h$ of this group, and one is looking for an integer $x$ such that $g^x = h$. We refer the reader to the nice article [37] for more historical details about NFS.

*Linear Algebra in NFS.* One of the main computational tasks in NFS and similar sieve algorithms consists in solving a large system of sparse linear equations over a finite field. An early integer factoring code [35] from 1975 was capable of factoring the seventh Fermat number $F_7 = 2^{2^7} + 1$ (a 39-digit number), using *dense* Gaussian elimination — the matrix occupied 1504K bytes in memory. However, factoring larger integers required larger and larger matrices, and soon switching to sparse representations became inevitable.

Sieve algorithms such as NFS produce random unstructured matrices on which sparse elimination fails badly after causing a catastrophic amount of fill-in. Thus, the linear algebra step of NFS relies on iterative methods. The block Lanczos algorithm [34] was used initially; it is based on, but quite different from the "original" algorithm of Lanczos [30] to find the eigenvalues and eigenvectors of a Hermitian matrix. It is now superseded by the block Wiedemann algorithm [12], because the former requires a single cluster of tightly interconnected machines while the latter allows the work to be distributed on a few such clusters. Given a square matrix $M$ of size $n$, both algorithms solve $Mx = 0$ by performing $\Theta(n)$ matrix-vector products.

| relation | matrix row (of original matrix) |
|---|---|
| relation-set | matrix row (of current matrix) |
| ideal | matrix column |
| column weight | number of non-zero elements in column |
| singleton | column of weight 1 |
| $k$-merge | elimination of a weight-$k$ column |
| excess | difference between number of rows and columns |
| "purge" step | elimination of singletons and excess |
| "merge" step | Structured Gaussian Elimination (SGE) |

**Figure 1: Number Field Sieve Glossary**

These iterative algorithms are Krylov subspace methods, yet they are quite different from their "numerical" counterparts such as the Biconjugate Gradient method [16] or GMRES [40]. Indeed, because the coefficients of the matrix live in finite fields, there can be no notion of numerical convergence. The algorithm due to Wiedemann [44], for instance, recovers the minimal polynomial $P(X) = X^r + a_{r-1}X^{r-1} + \cdots + a_1 X$ of the matrix — the constant coefficient $a_0$ is zero because the matrix is singular. Once the coefficients of $P(X)$ have been computed, a kernel vector can be obtained by evaluating $0 = M\left(M^{r-1} + a_{r-1}M^{r-2} + \cdots + a_1 I\right)$. The number of iterations of the block Lanczos/Wiedemann algorithm depends solely on the size of the matrix, and it is large.

In order to speed up the iterative solver, the matrix is preprocessed by doing some careful steps of Gaussian elimination. The effect is to reduce the size of the matrix while making it only moderately denser. As long as the density increase is moderate, performing $O(n)$ matrix-vector products using the modified matrix can be faster than with the original, given that the needed number of iterations $n$ decreases with the size of the current matrix.

## 2.1 Notations and NFS Glossary

In the whole article we denote by $n$ the number of rows of the current matrix $M$ (which is modified in-place, starting from the original matrix) by $m$ its number of columns and $W$ denotes the total weight of the matrix, i.e., the number of non-zero elements. A column of the matrix corresponds to what is called an *ideal* in the NFS community, and a row corresponds to a *relation*, or a *relation-set* when it results from a linear combination of several initial rows. The *weight* of a column is the number of non-zero elements in that column, see Fig. 1. The elements of the matrix are the exponents of the corresponding ideal, thus $r_1 = {j_1}^{e_1} \cdots {j_s}^{e_s}$, means that row $r_1$ has (possibly zero) elements at columns $j_1, \ldots, j_s$, and those elements are $e_1, \ldots, e_s$. In the integer factoring case, we have $e \in \{0, 1\}$, whereas in the discrete logarithm case, we have $0 \le e < \ell$, where $\ell$ is some integer related to the given discrete logarithm computation.

## 2.2 Benchmarks Matrices

We benchmark the efficiency of our algorithms using two relevant test cases: the factorization of RSA-512 [10], and the kilobit hidden SNFS discrete logarithm computation [17].

| Problem | Ref. | #rows $n$ | weight $W$ |
|---|---|---|---|
| RSA-512 | [10] | 17,117,966 | 277,554,404 |
| DL-1024 | [17] | 95,928,630 | 2,432,387,612 |

**Figure 2: Characteristics of benchmark matrices.**

The former is a number from the RSA factoring challenge[1], which was factored in August 1999. Filtering took one calendar month at that time. We used CADO-NFS [41] to generate the matrix $M$ for this number, using different parameters and state-of-the-art algorithms, namely [2] for the polynomial selection, and [6] for the "purge" step.

In 2016, Fried, Gaudry, Heninger and Thomé performed a discrete logarithm computation over a 1024-bit (hidden) prime field [17]. Sieving took one calendar month using up to 3000 CPU cores, and the linear algebra step took another calendar month on 200 cores. Inbetween, filtering was quick by comparison, but was a sequential process.

Figure 2 shows the characteristics of these two matrices. A common feature of the matrices produced by NFS is that rows have about the same density (around 16 for RSA-512, and 25 for DL-1024), and column $j$ has density roughly in $1/j$ if we neglect logarithmic factors (cf. Figures 3, 7 and 8); besides that, entries are distributed more or less randomly. The sudden jumps and drops in column density in Fig. 3 are an artifact of the sieving algorithm used to produce the matrix $M$ — namely lattice sieving — which outputs some columns (the "special-$q$" columns) more often that others.

The DL-1024 matrix has theoretically to be considered over GF($\ell$) for the SGE step, where $\ell$ is a parameter of the discrete logarithm computation (a prime number of 160 bits in this case). However, in practice its coefficients remain quite small during SGE. For example, the maximal coefficient of the DL-1024 input matrix is 33 (in absolute value), and the maximal coefficient of the output matrix is 47. This always holds in the DL case, so that we do not need to reduce coefficients modulo $\ell$, and we can consider them over $\mathbb{Z}$.

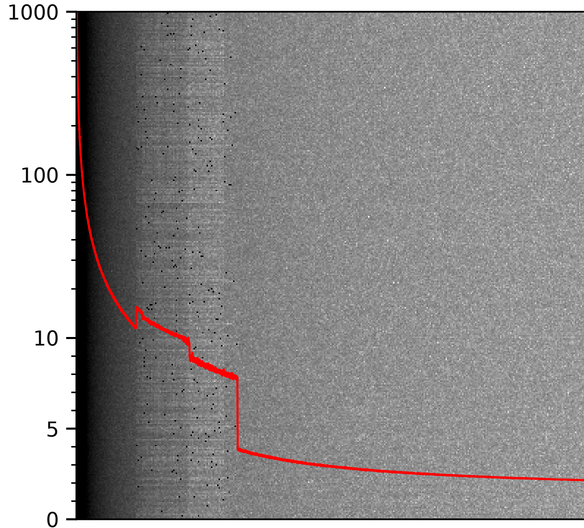
# 3 CLASSICAL STRUCTURED GAUSSIAN ELIMINATION

This section describes how the Structured Gaussian Elimination is classically performed, with a sequential program. We follow here [8] and Chapter 3 of [9].

We assume that we are given a target density $d$, and that SGE will stop when the average number of non-zero elements per row reaches $d$ in the Schur complement. In NFS, it is common to use $100 \le d \le 200$, for example the factorization of RSA-768 started with a matrix of about 2.5G rows, and ended with a matrix of size about 193M, with average row density 144 [26]. The goal of SGE is to get a matrix with average density $d$, and dimension as small as possible.

*Elimination.* Eliminating a column of weight $k \ge 1$ results in discarding this column and one row. It thus reduces the Schur complement by one row and one column. More precisely, assuming

[1] This number is named RSA-155 in the RSA factoring challenge, where 155 denotes its number of digits, but since newer numbers from the RSA challenge are numbered in bits, like RSA-768, we prefer to name it RSA-512.



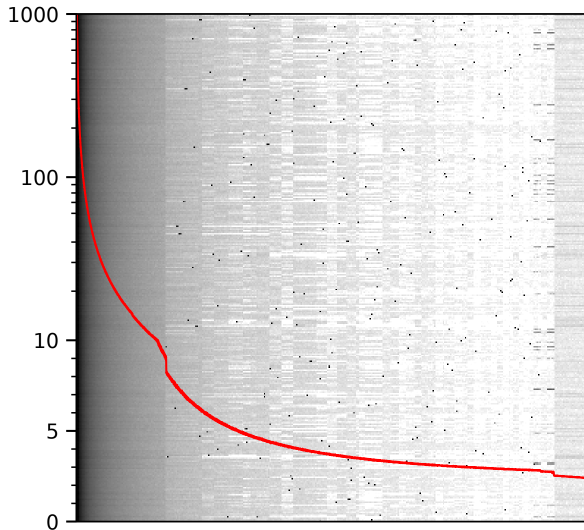




**Figure 3: Benchmark matrices (top: RSA-512, bottom: DL-1024). The intensity of each pixel is proportional to the logarithm of the density in the corresponding zone of the matrix. The presence of vertical bands is an artifact of lattice sieving. The red line shows the number of entries on each column. Note that the vertical scale is linear from 0 to 10 and logarithmic above 10.**

| | $j_1$ | $j_2$ | $j_3$ | $j_4$ | $j_5$ | $j_6$ | $j_7$ | $j_8$ |
|---|---|---|---|---|---|---|---|---|
| $r_1$ | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| $r_2$ | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| $r_3$ | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| $r_4$ | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| $r_5$ | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| $r_6$ | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| $r_7$ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| $r_8$ | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |

**Figure 4: Example matrix**

column $j$ has weight $k$, we replace the $k$ rows involving $j$ with $k-1$ combinations of rows where $j$ does not appear any more. The *excess* (difference between the number of rows and the number of columns) remains unchanged by this process. Eliminating a column $j$ of weight 2 replaces one of the rows $r_1$ and $r_2$ containing $j$ by $r_1 + r_2$ (in the integer factoring case), and discards the other row. Up from $k = 3$, we need to choose a pivot, and up from $k = 4$, we have even more choice, as we need not use a single "pivot" to eliminate a column.

*Elimination Cost and Fill-in.* When we eliminate columns, the number of non-zero elements of the current matrix usually changes. In sparse factorizations, the non-zero pattern of the input matrix is a subset of that of the factors; entries in the factors that do not appear in the input are called "fill-in".

In the case of SGE, where only the Schur complement is kept at the end of the process, eliminating a column may create fill-in, but it can also *reduce* the number of remaining non-zero entries for two reasons: 1) a column and a row are discarded from the Schur complement (this reduces its weight) and 2) numerical cancellation may kill non-zero entries. To avoid confusion, we refer to the variation of the number of non-zero entries in the Schur complement when a column is eliminated as the *cost* of this elimination. It can be negative.

Let us consider the elimination of column $j_7$ in Figure 4 (in the integer factoring case, thus modulo 2): row $r_2$ has initially weight 6, row $r_8$ has initially weight 6, and $r_2 + r_8 = j_4 + j_5$ has weight 2. The cost for that elimination is thus $2 - (6 + 6) = -10$. We have reduced the number of non-zero elements by 10. In general the cost for a column of weight 2 is $-2$, but it might be smaller if other columns than the one being eliminated do cancel between the two rows: here columns $j_1$, $j_3$, $j_6$ and $j_8$ also cancel.

Still with the matrix from Figure 4, column $j_3$ appears in rows $r_2$, $r_3$ and $r_8$, thus has weight 3. There are three different possible ways to eliminate column $j_3$: either (a) replace $r_2$, $r_3$ and $r_8$ by $r_2 + r_3$ and $r_2 + r_8$, or (b) replace them by $r_3 + r_2$, $r_3 + r_8$, or (c) replace them by $r_8 + r_2$, $r_8 + r_3$. The corresponding weights are 2 for $r_2 + r_8$ as seen above, 5 for $r_2 + r_3$, and 3 for $r_3 + r_8$, while the three rows have total weight $6 + 3 + 6 = 15$. The cost will thus be $5 + 2 - 15 = -8$ for case (a), $5 + 3 - 15 = -7$ for (b), and $2 + 3 - 15 = -10$ for (c). The optimal way to eliminate $j_3$ is thus via $r_8 + r_2$ and $r_8 + r_3$.

*Eliminating a Column.* Assume the $k$ rows with a non-zero coefficient in column $j$ are $i_0, \ldots, i_{k-1}$. Choose the row, say $i_0$, with the

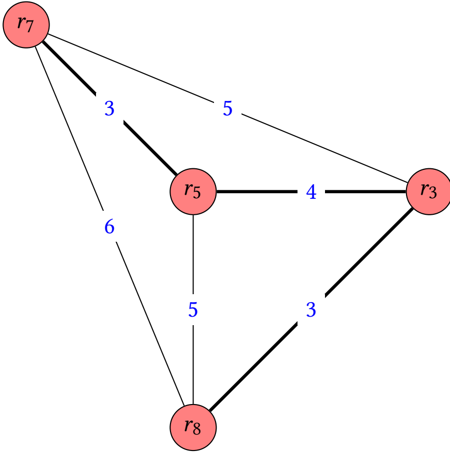



**Figure 5: The cost graph corresponding to the elimination of column $j_5$ of the matrix from Fig. 4. Using the sparsest row ($r_3$) as pivot yields a total weight of 12 for the 3 combined relations, while the minimal spanning tree algorithm performs $r_3 + r_5$, $r_5 + r_7$, and $r_3 + r_8$, with total weight 10.**

smallest weight, and use it to eliminate column $j$ in the other $k-1$ rows. If no other cancellation occurs, the cost of this strategy is:

$$c = (k-2)w(i_0) - 2(k-1), \tag{2}$$

where $w(i_0)$ denotes the weight of row $i_0$. Indeed, row $i_0$ is added to the $k-1$ other rows, and discarded afterwards, which amounts to $(k-2)w(i_0)$; and in each of the $k-1$ combinations of two rows, column $j$ is discarded twice.

Greedily using the entry on the sparsest row as "pivot" to eliminate a column is suboptimal, as illustrated by Figure 5. Instead, we use a strategy introduced by Cavallar [9]: we compute a minimal spanning tree of the complete graph whose vertices are the $k$ corresponding rows, and the weight of each edge is the weight one would obtain when combining the two adjacent rows, taking into account all cancellations (not only those of the eliminated column). We then use this minimal spanning tree to identify the additions we will perform for this column elimination.

Using this procedure is crucial because, as already noticed by Denny and Müller [15], then by Cavallar [9, p. 55] and later by Papadopoulos [36, slides 23-25], there are huge cancellations where two rows are combinations of the same row of the original matrix.

For example, on the RSA-512 experiment[2] from §2.2, for 36,679,976 additions of two rows, we have 174,440,850 columns that do cancel, i.e., an average of 4.76 per row addition; this means that on average, we have 3.76 extra cancellations, in addition to the one for the eliminated column. For the DL-1024 experiment, for 226,745,942 combinations[3] of two rows, we have 1,897,349,863 cancellations, i.e., 7.37 extra cancellations per row addition on average. Failing to identify those cancellations, with a minimal spanning tree or any other technique, will not only produce a larger elimination cost, but also propagate original rows all over the matrix, which will make much more expensive further eliminations.

[2]Exact figures might differ between runs because of the inherent non-determinism of the parallel algorithm.

[3]In the DL case, if the two rows have exponents $e$ and $e'$ respectively in the column to eliminate, and $g = \gcd(e, e')$, we multiply the first row by $e'/g$ and subtract the second one multiplied by $e/g$.

*Sequential SGE.* In its simplest form, the classical sequential algorithm works as follows: choose a column with minimum elimination cost; eliminate it; repeat while the density of the Schur complement stays below the given threshold. A well-known greedy pivot-selection strategy was proposed by Markowitz: choose the pivot whose elimination yields the lowest cost, assuming no numerical cancellation. In the context of SGE, equation (2) should be used as "Markowitz cost".

To make this fast, one possibility is to use a priority queue of columns sorted by increasing elimination cost, and to update the pending costs whenever a column elimination is performed, as proposed in [24]. This is essentially what the single-thread algorithm from CADO-NFS 2.3.0 does. However, the bookkeeping of these complex data-structures makes an efficient parallel implementation almost impossible.

# 4 THE PARALLEL ALGORITHM

To overcome this situation, we propose a new algorithm which keeps only some minimal data-structures, namely the current matrix. It turns out that this is also faster sequentially.

The new algorithm works by doing passes over the matrix. In each pass (steps 4-15 of Algorithm 1), candidate columns with low elimination cost are selected and some of those column eliminations are performed. To select these candidates, we could compute the elimination cost of all columns, but this would be very costly. We use two ideas to speed it up. First, since the elimination cost of a column is correlated with its weight, we pre-select columns of small weight, and only compute the elimination cost for these columns. Second, we use an upper-bound on the actual elimination cost using the "Markowitz cost" from Eq. (2). In such a way, we select the columns whose elimination cost stays under a given bound.

**Algorithm 1** Parallel Structured Gaussian Elimination

**Input:** matrix $M$ over GF($p$) with $n$ rows and $m$ columns, target $d$
**Output:** Schur complement (in place of $M$)

1: compute the total weight $W$ (# of non-zero elements) of $M$
2: $w_{\max} \leftarrow 2$, $c_{\max} \leftarrow 0$
3: **while** $W/n < d$ **do**
4:     extract submatrix $S$ of columns $j$ with $0 < w[j] \le w_{\max}$
5:     transpose $S$ and store the result in $R$ [§4.3]
6:     $L \leftarrow \emptyset$
7:     **for** each row of $R$ **do**
8:         let $j$ be the corresponding column
9:         using Eq. (2), obtain a bound $c$ for eliminating $j$ [§4.4]
10:         if $c \le c_{\max}$, append $j$ to $L$
11:     deduce from $L$ a maximal independent set $J$ ... [§4.5]
12:     ... and eliminate in-place all columns of $J$ [§4.5]
13:     update number of rows $n$ and total weight $W$
14:     **if** all columns of weight 2 have been eliminated **then**
15:         $w_{\max} \leftarrow \min(32, w_{\max}+1)$, $c_{\max} \leftarrow c_{\max} + c_{\text{incr}}$

In Algorithm 1, the bound $c_{\max}$ for the elimination cost is increased by $c_{\mathrm{incr}}$ at each iteration, once all columns of weight 2 have been eliminated (these columns benefit from a preferential treatment, because their elimination cost is always negative). We found experimentally that the value $c_{\mathrm{incr}} = 13$ is close to optimal for integer factorization, and $c_{\mathrm{incr}} = 31$ for discrete logarithm computations. We use a hardcoded upper-bound of 32 on the weight of selected columns $w_{\max}$.

## 4.1 Sparse Matrix Formats

An $n \times m$ sparse matrix $A$ with $nnz$ non-zero entries can be seen as a set of $nnz$ triples $(i, j, x)$ such that $A_{ij} = x$. We use three representations of sparse matrices in memory. For a more in-depth presentation, we refer the reader to [14].

The *coordinate* format uses three arrays `Ai`, `Aj` and `Ax` of size $nnz$ to store each component of the triplets. For $0 \le k < nnz$, the coefficient $\texttt{Ax}[k]$ is in row $\texttt{Ai}[k]$ and column $\texttt{Aj}[k]$. It is commonly used to store sparse matrices in files, for instance in the MatrixMarket format.

The *compressed sparse rows* (CSR) format groups the triples by row indices. It uses an array of "row pointers" `Ap` of size $n + 1$ and two arrays `Aj` and `Ax` of size $nnz$. It allows direct access to the rows: the column index and value of entries on row $i$ are found in `Aj` and `Ax` at locations $\texttt{Ap}[i], \ldots, \texttt{Ap}[i + 1] - 1$. This representation, or its "by column" twin, is commonly used in sparse matrix software tools. As such, they usually include a "coordinate to CSR" format conversion routine.

The *list of list* (LIL) format uses an array of actual pointers in memory to access the rows: the entries of row $i$ are found at memory address $\texttt{Ap}[i]$. Each row is described by its length and an array of pairs $(j, x)$. This format is more rarely used, but it allows in-place modification of the rows, at the expense of a more complex memory management. We use the LIL format in Algorithm 1 for the current matrix (below, $M$ will denote the current matrix of Algorithm 1).

## 4.2 Computing and Updating the Weights

A full computation of the column weights is done only once, the first time the `while` loop of Algorithm 1 is performed. This initial computation is multi-threaded, but since it takes a very small proportion of the total time, we do not discuss it further.

In the subsequent passes, the weights are updated incrementally, whenever a new column is eliminated (see §4.5). In such a way, the cost of updating the column weights is essentially proportional to the number of column eliminations performed.

## 4.3 Computing the Rows for Each Column Elimination

For those columns with weight less or equal to $w_{\max}$, we need to access the rows where these columns appear (steps 4-5), to estimate the cost of each potential column elimination (step 9) and to actually eliminate these columns (step 12).

This involves two tasks: 1) extract the submatrix $S$ (of the current matrix) composed of columns of weight at most $w_{\max}$, and 2) compute its transpose $R = S^t$. These two tasks are somewhat entangled. Because $R$ will only be read and must be accessed by rows, it is stored in CSR format. However, $S$ is stored in coordinate format for simplicity.

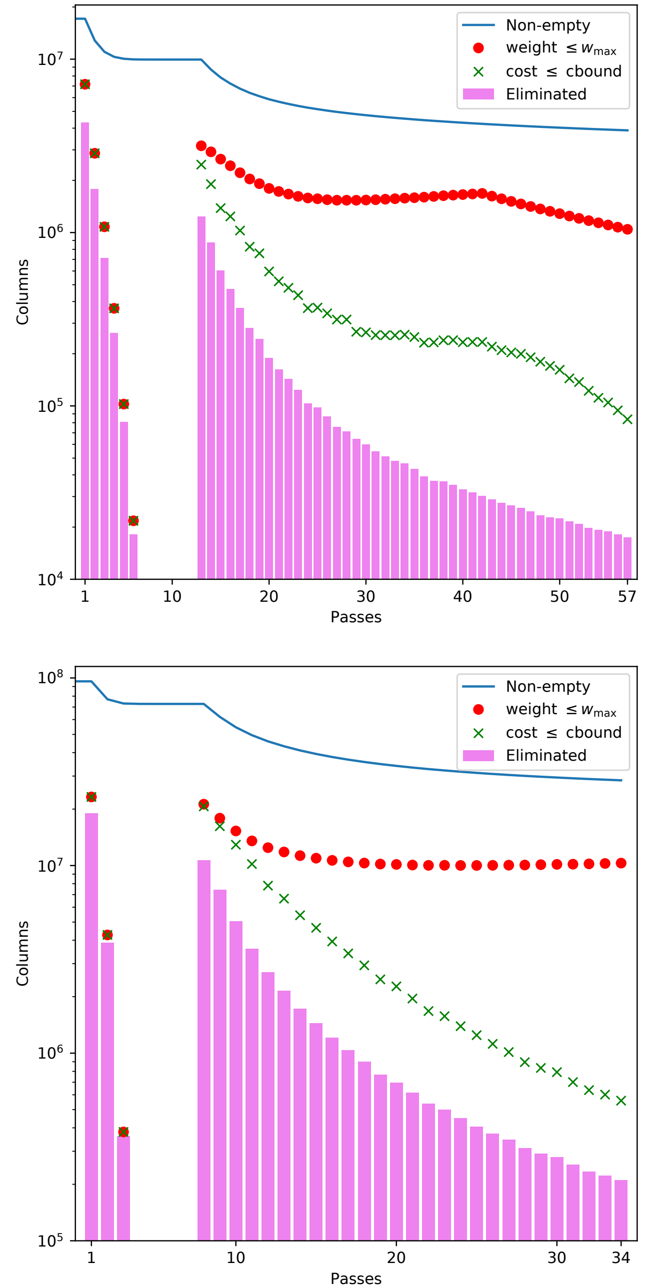



**Figure 6: Evolution of column types during the progress of the algorithm on RSA-512 (top) and DL-1024 (bottom).**

Fig. 6 shows that 10–20% of the columns of $M$ belong to the extracted submatrix (curve weight $\le w_{\max}$). To avoid the wasteful situation where the transpose contains 80–90% of empty rows, we renumber the extracted columns on-the-fly: column $j$ in $M$ corresponds to row $q[j]$ in $R$, so that $R$ has no empty rows. Once

this is done, a potential column elimination is represented internally by a row index $k$ in $R$. The corresponding column can be found by looking up the reverse renumbering $j = qinv[k]$.

*Submatrix Extraction.* To extract the "low column-weight" submatrix (step 4), we first iterate over the column weight array $w[]$ in order to accumulate the number of selected columns and the number of entries in these columns. We also compute the row pointers for $R$ (this is a prefix-sum operation). For each column $j$, if $0 < w[j] < w_{\max}$, then do: $\texttt{Rq}[j] \leftarrow Rn$, $\texttt{Rqinv}[Rn] \leftarrow j$, $\texttt{Rp}[Rn] \leftarrow Rnz$, $Rnz \leftarrow Rnz + w[j]$, $Rn \leftarrow Rn + 1$.

This is easily parallelized. The parallelization of the prefix-sum operation needs to read the $w[]$ array twice, and performs twice more arithmetic operations than the sequential version.

Given these informations, storage to hold $S$ is allocated. To actually extract it, we iterate in parallel over the rows of $M$. Because the row lengths are not uniformly distributed (see Figures 7 and 8), dynamic load-balancing has to be used. For each row, we copy selected entries to $S$. Concurrent access to $S$ is handled as follows: each thread copies chosen entries to a thread-private buffer that fits in L1 cache. When this buffer is full, it is flushed to $S$. This involves an OpenMP `atomic capture` directive to update a pointer to the end of $S$. The copy from the buffer to $S$ is done using non-temporal stores (using `vmovntdq` from the AVX instruction set — conveniently accessible with the `_mm256_stream_si256` intrinsic).

*Transposition.* Once $S$ has been extracted, we need to obtain its transpose in CSR format, which we denote by $R$ (step 5).

Transposing a sparse matrix stored in coordinate format is trivial: it suffices to swap the two arrays `Ai` and `Aj`. Thus, converting a matrix from coordinate to CSR format and producing its transpose (in CSR format) are essentially the same operation.

It was observed in 1979 by Gustavson [20] that sparse transposition is related to sorting: in a sparse matrix $A$ in CSR format, the entries are stored in memory in increasing row number, while in $A^t$ they are stored in increasing column number. Because the column indices are known to live in a small interval, Gustavson observed they can be sorted in linear time using a bucket sort with one bucket per row of the output (this sorting algorithm is called *distribution counting* in [28, §5.2]). This yields a sparse transposition algorithm that runs in $O(n + m + nnz)$ operations.

This algorithm is almost always implemented sequentially. It is for instance the case in several widely-used software packages (UMFPACK [13], CHOLMOD [11], SuperLU [31], Octave [22], SciPy [23], PaStiX [21], ...). Indeed, transposition (or conversion to CSR) is often done only once and is almost never a bottleneck.

Wang, Liu, Hou and Feng [42] recently discussed scenarios where sparse matrix transposition is somewhat critical, such as Simultaneous Localization and Mapping (SLAM) problems, which repeatedly require the computation of $A^t A$ for sparse $A$. Wang *et al.* also propose two parallel sparse transposition algorithms, which are in fact a parallel bucket sort and a parallel merge sort.

In fact, doing a bucket sort with a very large number of buckets is a terrible idea on current hardware: accessing each bucket is likely to cause both a cache miss and a TLB miss. This is well-known in the relational database community: Balkesen, Teubner, Alonso and Özsu [3] compared a *hardware oblivious* direct bucket-sort with a *hardware conscious* multi-pass radix sort and found the latter to be much more efficient for large in-memory join computations.

We thus implemented sparse transposition using a radix sort. Let $b$ denote $\lceil \log_2 Rn \rceil$, where $Rn$ is the number of rows of $R$, thus of columns of $S$. We perform $k$ sorting passes where the $i$-th pass does a "distribution counting" on $a_i$ bits of the column indices (with $a_1 + \cdots + a_k = b$). The efficiency of a pass decreases when the "radix" $a_i$ is too high; the best value depends on characteristics of the memory subsystem (number of L1 cache lines, number of TLB entries, ...). We use an upper-bound of 11 for all $a_i$s. We included several improvements described in Rahman and Raman [39]: performing all counting steps at once (a technique that they credit to the work of Friend in 1956 [18]) and using of a *software write-combining buffer*. This means that, instead of writing entries directly to their output bucket (which may cause a cache and a TLB miss), they are written to a small per-bucket intermediate buffer which has the size of an L1 cache line. When such as buffer is full, it is flushed to the actual bucket with non-temporal stores. This enables the use of higher radices and provides better scalability.

Our implementation of radix sort is parallel. This is not original: Wassenberg and Sanders [43] describe such a parallel radix sort. The first pass operates (in parallel) on the most significant bits; this splits the input into independent sub-lists which can then be processed in parallel without synchronization. These sub-lists are sorted by applying the remaining passes sequentially, starting with the least significant bits.

## 4.4 Estimating the Elimination Cost of Each Column

We have computed — or updated — the weight $w[j]$ of each column, and for those columns with $w[j] \leq w_{\max}$, we have extracted the rows where $j$ appears, say rows $R_{q[j],1}, R_{q[j],2}, \ldots, R_{q[j],w[j]}$ (see §4.3). We now want to estimate the cost of eliminating column $j$. The rationale is that we want to greedily eliminate columns with smaller cost, in order to control the global fill-in of the Schur complement.

Since this step is almost read-only, it parallelizes well. Nevertheless, we use dynamic load-balancing because the columns of larger index have smaller weight (cf Fig. 3). The cost of eliminating a column is estimated using Eq. (2).

We could also compute the exact elimination cost using a minimal spanning tree computation as described in §3; however, we found experimentally that this is much more expensive and does not save much in the final matrix. This can be explained as follows. The cost estimate we get is an upper-bound of the exact cost: if the estimate is so small that the corresponding column is selected in step 10, a fortiori it would be with the real cost. What can occur is that we miss a column with a small (exact) cost, because the estimate is larger than $c_{\max}$, but usually it will be selected in a further pass.

## 4.5 Eliminating the Columns

This step is the most critical one, since it modifies the matrix itself. We want to eliminate the columns whose cost is below some threshold $c_{\max}$ (see Algorithm 1). However, if two columns to eliminate have a non-zero element in the same row, we might have problems. Consider for example the elimination of columns $j_3$ and $j_7$

in Figure 4, implying rows $r_2, r_3, r_8$, and $r_2, r_8$ respectively. Assume the elimination of column $j_3$ is performed by thread 0, and the elimination of column $j_7$ is performed by thread 1. Thread 0 might decide to add row $r_3$ to $r_2$ and $r_8$, and remove row $r_3$ afterwards. Thread 1 might decide to add row $r_8$ to $r_2$, and remove row $r_8$. We see that row $r_8$ might be already discarded when thread 0 tries to access it. Even using a lock on each row will not solve that problem, since once $r_8$ is added to $r_2$, and $r_8$ discarded, the weight of column $j_3$ drops from 3 to 1, and thus the pre-computed rows for column $j_3$ become invalid. (Note that this issue also exists in sequential code.)

The solution we propose to overcome this issue is to compute a maximal subset $J$ of *independent columns* of the set $L$ of selected columns. We say that two columns are *independent* if they do not share any row, i.e., if no row contains a non-zero coefficient in both columns. Still with the matrix from Figure 4, the maximal independent sets of columns are $\{j_1\}$, $\{j_2, j_3\}$, $\{j_2, j_7\}$, $\{j_4\}$, $\{j_5\}$, $\{j_6\}$. What we need is in fact a maximal independent set in the column intersection graph of $M$: the graph whose vertices are columns of $M$ where $j \leftrightarrow j'$ when the columns $j$ and $j'$ have an entry on the same row. The adjacency matrix of this graph is precisely $M^t M$.

The *column elimination tree* [32] (the elimination tree of $M^t M$) is widely used in sparse factorizations; it captures dependencies between columns of $M$ in the presence of partial pivoting when all columns are eliminated in order. The leaves of the column elimination tree are indeed independent columns, but the converse is not true (this is because the column elimination tree encodes *transitive* dependencies between all columns, assuming they are all eliminated). For instance, the column elimination tree of the example matrix of Figure 4 is a string $j_1 \to \cdots \to j_8$, therefore $j_2$ is a descendant of $j_7$, yet $j_2$ and $j_7$ are independent.

It is well-known that finding a maximum independent set is NP-complete. Therefore we settle for a greedy parallel algorithm that only yields a large (not maximum) independent set. This problem was already studied in the literature for graphs, and some parallel algorithms were designed (see [4] and its references). Once we have found a set $J$ of independent columns, we can eliminate them in parallel, since they will read/write separate rows.

The maximal subset $J$ of independent columns is computed using a greedy multi-thread algorithm, using a byte-array to identify the rows already involved in a previously selected column. When a thread considers a new potential column to eliminate, it first checks if all rows where that column appears have their corresponding byte set to "free". If this is not the case, that potential column elimination is discarded; otherwise, the rows where that column appears are checked again, this time by setting the corresponding bytes to "busy", using the OpenMP `atomic capture` directive. If that second check succeeds (it might fail if some rows have been set "busy" by other threads in the meantime), the thread proceeds with the corresponding column elimination, using the minimum spanning tree algorithm described in §3. Steps 11 and 12 of Algorithm 1 are therefore interleaved.

We found that this greedy two-pass procedure to find independent columns (with a first lock-free pass) is much more efficient than either a one-pass procedure using locks, or using hardware transactional memory, and in practice the number of columns that are discarded in the second pass is very small (at most a few dozens in all our experiments, while the set $L$ contains several million columns). When that happens, a few rows are spuriously left in a "busy" state until the end of the pass.

When we eliminate a column of weight $k$, we have $k-1$ row combinations and one row deletion. This entails $k-1$ calls to `malloc` to allocate the new rows, and $k$ calls to `free` to discard the old rows. Now if we have many concurrent threads eliminating columns at the same time, the `malloc` and `free` routines will be under high pressure, thus the global efficiency of the algorithm will also depend on how those routines scale with many threads.

*Updating the Column Weights Incrementally.* When we add say row $i'$ to row $i$, for each column index $j$ present in $i'$, we increase $w[j]$ by 1 if $j$ is not present in $i$, otherwise we decrease $w[j]$ by 1, since $j$ cancels between $i$ and $i'$. (This is in the integer factoring case, in the DL case $w[j]$ might stay unchanged.) When we remove a row, we simply decrease the weight of all its column indices. Those operations can be performed efficiently in parallel using the OpenMP `atomic update` directive.

# 5 EXPERIMENTS

We have implemented Algorithm 1 in CADO-NFS [41], which is an open-source implementation of the Number Field Sieve, capable of dealing with both integer factorization and discrete logarithm computations. We used OpenMP to deal with parallelism. We compare in this section our parallel implementation with that of CADO-NFS 2.3.0, which implements a single-thread algorithm.

Experimental Setup: We used a machine equipped with two Intel Xeon Gold 6130 CPUs each with 16 cores running at 2.1Ghz, 192GB RAM, GCC 8.3.0, under Debian GNU/Linux 9.8, with hyper-threading and turbo-boost disabled. For better performance, we used Google's `tcmalloc` library (thread-caching malloc) for the `malloc` and `free` calls.

Within CADO-NFS, the output of the SGE program (called `merge`) is a `history` file, which contains the sequence of row combinations and/or eliminations performed — essentially, matrix $A$ from Eq. (1). Another program (called `replay`) then actually computes $M'$ from $M$ and the `history` file. Each column elimination corresponds to a few lines in this `history` file, which are written by a single `fprintf` system call. Since `fprintf` is thread-safe, this guarantees that the lines written by two different threads are not interleaved. However, it is very important to bufferize those lines per thread, otherwise the lock set by `fprintf` will make the other threads wait for their turn, and decrease the overall speedup.

Figures 7 and 8 show the distribution of row sizes for both benchmark matrices before and after the SGE step. We see that in both cases, the row sizes follow somehow a normal distribution at input (with mean around 16 for RSA-512 and 25 for DL-1024), while the output distributions are far from normal, and can contain quite dense rows (up to 2000 non-zero elements for RSA-512, up to 1000 for DL-1024). This explains why some steps of the algorithm have to be implemented using dynamic scheduling.

Figure 9 demonstrates the efficiency of our parallel algorithm on the RSA-512 and DL-1024 benchmarks. The "output rows" values are interpolated, since Algorithm 1 might not stop *exactly* when the target density is attained. The time is wall-clock time, not including the time to read the matrix, and the time to free all memory. We observe from this figure that:

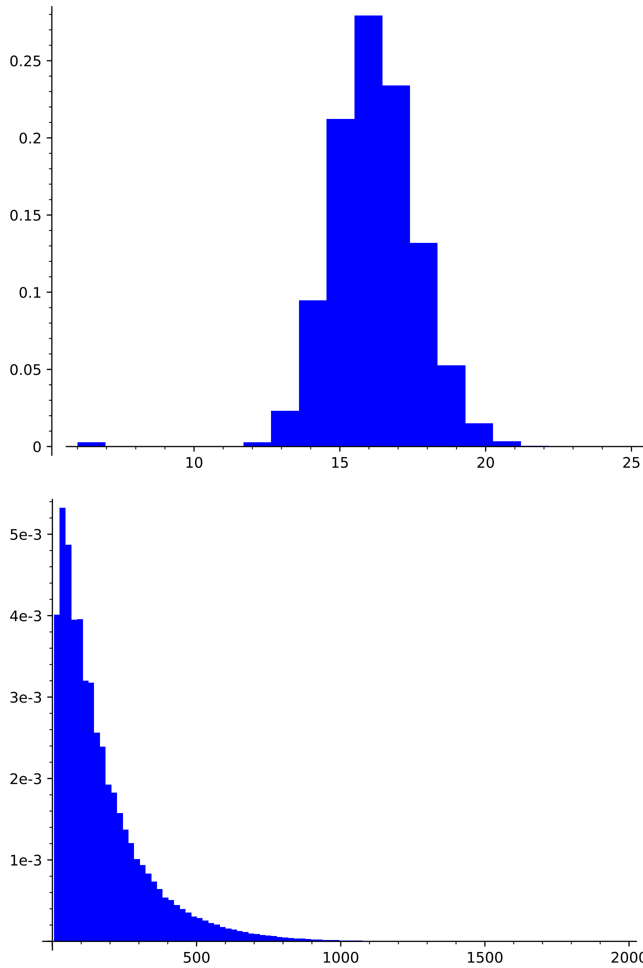



**Figure 7: Histogram of row sizes for RSA-512 before (up) and after (down) the SGE step.**

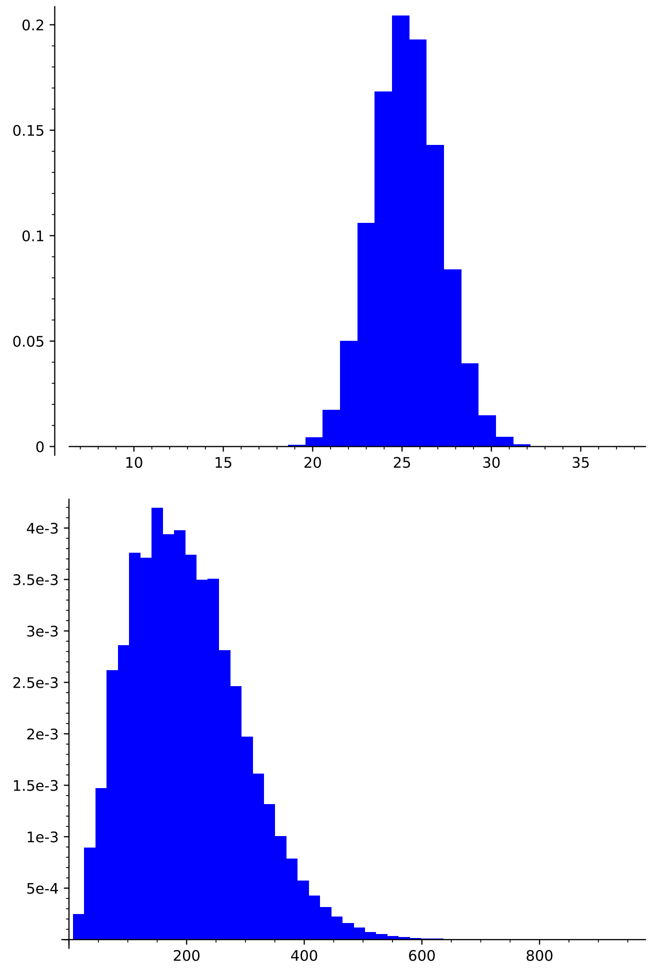



**Figure 8: Histogram of row sizes for DL-1024 before (up) and after (down) the SGE step.**

- the new algorithm always produces a smaller final matrix (by almost 10% for RSA-512, and by 2% for DL-1024);
- the new algorithm uses less memory than CADO-NFS 2.3.0, even with 32 threads;
- the new algorithm is faster even with one thread (by a factor of 5.2 for RSA-512, 8.2 for DL-1024);
- the new algorithm scales reasonably well with the number of threads.

With 32 threads, the use of the `tcmalloc` library yields an overall speedup of a factor 1.7 (resp. 2.1) for the RSA-512 (resp. DL-1024) benchmark over the GNU libc `malloc`/`free` implementation. This shows that with a large number of threads, we not only exercise the efficiency of our algorithm, but also of the memory management implementation.

Figure 10 shows the detailed timings and speedup of the different steps of Algorithm 1, relative to the corresponding wall-clock time with one thread. The two most expensive steps are computing the transposed matrix $R$ and performing the column eliminations.

| **RSA-512** | 2.3.0 | This article | | | | | |
|---|---|---|---|---|---|---|---|
| Threads | 1 | 1 | 2 | 4 | 8 | 16 | 32 |
| output rows (M) | 4.27 | 3.86 | 3.86 | 3.86 | 3.86 | 3.86 | 3.86 |
| time (s) | 2273 | 435 | 227 | 117 | 61 | 33 | 20 |
| speedup | - | 1 | 1.92 | 3.72 | 7.13 | 13.2 | 21.8 |
| memory (GB) | 6.5 | 5.8 | 5.8 | 5.8 | 5.8 | 5.9 | 6.1 |

| **DL-1024** | 2.3.0 | This article | | | | | |
|---|---|---|---|---|---|---|---|
| Threads | 1 | 1 | 2 | 4 | 8 | 16 | 32 |
| output rows (M) | 28.8 | 28.2 | 28.2 | 28.2 | 28.2 | 28.2 | 28.2 |
| time (s) | 24294 | 2964 | 1563 | 797 | 420 | 226 | 140 |
| speedup | - | 1 | 1.90 | 3.72 | 7.06 | 13.1 | 21.2 |
| memory (GB) | 92.4 | 85.1 | 85.1 | 86.6 | 86.8 | 86.8 | 84.7 |

**Figure 9: Experimental results for RSA-512 (target density: 170 entries/row) and DL-1024 (target density: 200 entries/row) compared to CADO-NFS 2.3.0.**

| **RSA-512** | Time (s) | Speedup | | | | |
|---|---|---|---|---|---|---|
| Threads | 1 | 2 | 4 | 8 | 16 | 32 |
| Computing $R$ [§4.3] | 147.5 | 2.0 | 3.8 | 7.4 | 14.0 | 24.2 |
| Computing $L$ [§4.4] | 25.1 | 1.8 | 3.4 | 6.6 | 12.6 | 19.3 |
| Eliminations [§4.5] | 250.7 | 1.9 | 3.7 | 7.1 | 13.6 | 23.0 |
| Total | 435 | 1.9 | 3.7 | 7.1 | 13.2 | 21.8 |

| **DL-1024** | Time (s) | Speedup | | | | |
|---|---|---|---|---|---|---|
| Threads | 1 | 2 | 4 | 8 | 16 | 32 |
| Computing $R$ [§4.3] | 802.8 | 1.8 | 3.5 | 6.8 | 12.9 | 22.3 |
| Computing $L$ [§4.4] | 221.1 | 2.1 | 4.0 | 7.9 | 14.9 | 23.8 |
| Eliminations [§4.5] | 1816.8 | 1.9 | 3.8 | 7.3 | 13.7 | 22.5 |
| Total | 2964 | 1.9 | 3.7 | 7.1 | 13.1 | 21.2 |

**Figure 10: Detailed timings and speedup of the different steps of SGE.**

*Parallel Sparse Transpose.* SGE is one of the few algorithms whose performance and scalability heavily depends on that of sparse transposition (to compute $R$). It turns out that extracting $S$ is much longer than transposing it, and fortunately scales much better. We benchmarked several sparse transpose codes against a naive, direct implementation of the Gustavson algorithm. For this, we stored to disk all matrices $S$ extracted while performing SGE on our larger DL-1024 matrix. Then, for each of the 35 resulting matrices, we loaded it in coordinate representation, transposed it (trivially) and converted the result to CSR representation using the various algorithms.

A direct implementation of the Gustavson algorithm takes 447s to convert all matrices. Using the radix sort described in §4.3 on a single core takes 60s. Using all 32 cores, this drops down to 15s (a disappointing ×4 acceleration). Both versions move each non-zero entry several times in memory: we measured a memory bandwidth of 25-30GB/s using 32 threads. For the sake of comparison, the Intel MKL library (version 2019 update 3) takes 39s.

Wang *et al.* proposed two parallel sparse transpose algorithms but did not report actual speedups. However, their code is available, and we could find that the proposed techniques do not scale very well: we observed a median speedup of 2 compared to a naive sequential implementation, using 32 cores, on their own benchmark matrices. In each case, the best speedup is not obtained using all available physical cores, and performance ultimately degrades when the number of cores actually used increases.

## 6 CONCLUSION

In this paper, we have proposed a parallel algorithm for the SGE step of the Number Field Sieve. We have shown that our implementation of this algorithm compares very well to the state-of-the-art CADO-NFS program, even in single-thread mode, and scales reasonably well (with a large number of threads, we hit the limits of the memory bandwidth and of the malloc/free implementation).

Some authors (for example [29]) claim that *there should be considerably more equations than unknowns* for SGE to work best. We have demonstrated in this article that there is no need of a huge excess for the SGE step. On the contrary, in CADO-NFS, almost all the excess is removed in the "purge" step, for which state-of-the-art algorithms have greatly improved [6].

The goal of SGE is to make a subsequent iterative method faster by applying it to $M'$ instead of $M$ in Equation (1). It follows from (1) that:

$$M' = \begin{pmatrix} -A & I \end{pmatrix} \begin{pmatrix} M_{01} \\ M_{11} \end{pmatrix}, \qquad \text{where } PMQ = \begin{pmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{pmatrix}.$$

The iterative method (e.g., the block Wiedemann algorithm) then computes a sequence $(x_n)$ defined by $x_{n+1} = x_n M'$. Kleinjung's "double matrix" idea [25] consists in computing instead $y = x_n(-A \mid I)$, followed by $x_{n+1} = y(M_{01} \mid M_{11})^t$. This is quite similar to the use of an incomplete LU factorization as a preconditioner in numerical iterative methods. If computing both vector-matrix products is faster than the original operation, then the iterative solver will be faster. In this case, the objective of SGE will no longer be to minimize fill-in inside $M'$, but instead to minimize fill-in inside $A$. Algorithm 1 can easily be adapted to the "double matrix" SGE.

This work does not address the case of a *distributed implementation* of SGE, where the input/output matrices are split among several nodes. New algorithms are needed for that case. A possible strategy is to split the matrix column-wise: each thread stores a subset of the columns. In such a way, when two rows are combined, each thread only has to combine its subset of the two rows. However, some communication will be required, for example to gather the information needed to compute the minimum spanning trees.

*Acknowledgements.* Experiments presented in this article were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). This work was supported by the French "Ministère de l'Enseignement Supérieur et de la Recherche", by the "Conseil Régional de Lorraine", and by the European Union, through the "Cyber-Entreprises" project. We also thank the authors of [17] who kindly made the data of their computation available, and Jens Gustedt for fruitful discussions about atomic operations.

## REFERENCES

[1] Amestoy, P. R., Davis, T. A., and Duff, I. S. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis Applications 17*, 4 (1996), 886–905.

[2] Bai, S., Bouvier, C., Kruppa, A., and Zimmermann, P. Better polynomials for GNFS. *Mathematics of Computation 85* (2016), 861–873.

[3] Balkesen, C., Teubner, J., Alonso, G., and Özsu, M. T. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (April 2013), pp. 362–373.

[4] Blelloch, G. E., Fineman, J. T., and Shun, J. Greedy sequential maximal independent set and matching are parallel on average. In *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012* (2012), G. E. Blelloch and M. Herlihy, Eds., ACM, pp. 308–317.

[5] Bouillaguet, C., Delaplace, C., and Voge, M. Parallel sparse PLUQ factorization modulo $p$. In *Proceedings of the International Workshop on Parallel Symbolic Computation, PASCO@ISSAC 2017, Kaiserslautern, Germany, July 23-24, 2017* (2017), J. Faugère, M. B. Monagan, and H. Loidl, Eds., ACM, pp. 8:1–8:10.

[6] Bouvier, C. The filtering step of discrete logarithm and integer factorization algorithms. http://hal.inria.fr/hal-00734654, 2013. Preprint, 22 pages.

[7] Buhler, J. P., Lenstra, H. W., and Pomerance, C. Factoring integers with the number field sieve. In *The development of the number field sieve* (Berlin, Heidelberg, 1993), A. K. Lenstra and H. W. Lenstra, Eds., Springer Berlin Heidelberg, pp. 50–94.

[8] Cavallar, S. Strategies in filtering in the number field sieve. In *Algorithmic Number Theory, 4th International Symposium, ANTS-IV, Leiden, The Netherlands, July 2-7, 2000, Proceedings* (2000), W. Bosma, Ed., vol. 1838 of *Lecture Notes in Computer Science*, Springer, pp. 209–232.

[9] Cavallar, S. *On the Number Field Sieve Integer Factorisation Algorithm*. PhD thesis, University of Leiden, 2002. 108 pages.

[10] Cavallar, S., Dodson, B., Lenstra, A. K., Lioen, W., Montgomery, P. L., Murphy, B., te Riele, H., Aardal, K., Gilchrist, J., Guillerm, G., Leyland, P., Marchand, J., Morain, F., Muffett, A., Putnam, C., Putnam, C., and Zimmermann, P. Factorization of a 512–bit RSA modulus. In *Proceedings of Eurocrypt'2000* (Bruges, Belgium, 2000), B. Preneel, Ed., vol. 1807 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–18.

[11] Chen, Y., Davis, T. A., Hager, W. W., and Rajamanickam, S. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Trans. Math. Softw. 35*, 3 (Oct. 2008), 22:1–22:14.

[12] Coppersmith, D. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Mathematics of Computation 62*, 205 (1994), 333–350.

[13] Davis, T. A. Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. 30*, 2 (June 2004), 196–199.

[14] Davis, T. A. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[15] Denny, T. F., and Müller, V. On the reduction of composes relations from the number field sieve. In *Algorithmic Number Theory, Second International Symposium, ANTS-II, Talence, France, May 18-23, 1996, Proceedings* (1996), H. Cohen, Ed., vol. 1122 of *Lecture Notes in Computer Science*, Springer, pp. 75–90.

[16] Fletcher, R. Conjugate gradient methods for indefinite systems. In *Numerical Analysis* (Berlin, Heidelberg, 1976), G. A. Watson, Ed., Springer Berlin Heidelberg, pp. 73–89.

[17] Fried, J., Gaudry, P., Heninger, N., and Thomé, E. A kilobit hidden SNFS discrete logarithm computation. In *36th Annual International Conference on the Theory and Applications of Cryptographic Techniques - Eurocrypt 2017* (Paris, France, Apr. 2017), J.-S. Coron and J. B. Nielsen, Eds., vol. 10210 of *Advances in Cryptology – EUROCRYPT 2017*, Springer.

[18] Friend, E. H. Sorting on electronic computer systems. *J. ACM 3*, 3 (July 1956), 134–168.

[19] George, A. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis 10*, 2 (Apr. 1973), 345–363.

[20] Gustavson, F. G. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw. 4*, 3 (Sept. 1978), 250–269.

[21] Hénon, P., Ramet, P., and Roman, J. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing 28*, 2 (2002), 301–321.

[22] John W. Eaton, David Bateman, S. H., and Wehbring, R. *GNU Octave version 5.1.0 manual: a high-level interactive language for numerical computations.* 2018.

[23] Jones, E., Oliphant, T., Peterson, P., et al. SciPy: Open source scientific tools for Python, 2001–.

[24] Joux, A., and Lercier, R. Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the Gaussian integer method. *Math. Comput. 72*, 242 (2003), 953–967.

[25] Kleinjung, T. Filtering and the matrix step in NFS. Slides presented at the Workshop on Computational Number Theory on the occasion of Herman te Riele's retirement from CWI Amsterdam, 2011. https://event.cwi.nl/wcnt2011/slides/kleinjung.pdf.

[26] Kleinjung, T., Aoki, K., Franke, J., Lenstra, A. K., Thomé, E., Bos, J. W., Gaudry, P., Kruppa, A., Montgomery, P. L., Osvik, D. A., te Riele, H., Timofeev, A., and Zimmermann, P. Factorization of a 768-bit RSA modulus. In *CRYPTO 2010 Advances in Cryptology - CRYPTO 2010* (Santa Barbara, USA, 2010), T. Rabin, Ed., vol. 6223 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 333–350.

[27] Kleinjung, T., Diem, C., Lenstra, A. K., Priplata, C., and Stahlke, C. Computation of a 768-bit prime field discrete logarithm. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I* (2017), J. Coron and J. B. Nielsen, Eds., vol. 10210 of *Lecture Notes in Computer Science*, pp. 185–201.

[28] Knuth, D. E. *Searching and sorting*, second ed., vol. 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 10 Jan. 1998.

[29] Lamacchia, B. A., and Odlyzko, A. M. Solving large sparse linear systems over finite fields. In *Advances in Cryptology (CRYPTO'90)* (1991), A. J. Menezes and S. A. Vanstone, Eds., vol. 537 of *Lecture Notes in Computer Science*, Springer, pp. 109–133.

[30] Lanczos, C. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Natl. Bur. Stand. B 45* (1950), 255–282.

[31] Li, X. S. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw. 31*, 3 (September 2005), 302–325.

[32] Liu, J. W. H. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl. 11*, 1 (Jan. 1990), 134–172.

[33] Markowitz, H. M. The elimination form of the inverse and its application to linear programing. *Management Sci. 3* (1957), 255–269.

[34] Montgomery, P. L. A block Lanczos algorithm for finding dependencies over GF(2). In *Advances in Cryptology — EUROCRYPT '95* (Berlin, Heidelberg, 1995), L. C. Guillou and J.-J. Quisquater, Eds., Springer Berlin Heidelberg, pp. 106–120.

[35] Morrison, M. A., and Brillhart, J. A method of factoring and the factorization of $F_7$. *Mathematics of Computation 29*, 129 (1975), 183–205.

[36] Papadopoulos, J. A self-tuning filtering implementation for the number field sieve. Slides presented at CADO workshop on integer factorization, 2008. http://cado.gforge.inria.fr/workshop/abstracts.html.

[37] Pomerance, C. A tale of two sieves. *Notices of the AMS 43*, 12 (Dec. 1996), 1473–1485.

[38] Pomerance, C., and Smith, J. W. Reduction of huge, sparse matrices over finite fields via created catastrophes. *Experimental Mathematics 1*, 2 (1992), 89–94.

[39] Rahman, N., and Raman, R. Adapting radix sort to the memory hierarchy. *J. Exp. Algorithmics 6* (Dec. 2001).

[40] Saad, Y., and Schultz, M. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing 7*, 3 (1986), 856–869.

[41] The CADO-NFS Development Team. CADO-NFS, an implementation of the number field sieve algorithm. http://cado-nfs.gforge.inria.fr/, 2017. Release 2.3.0.

[42] Wang, H., Liu, W., Hou, K., and Feng, W.-c. Parallel transposition of sparse data structures. In *Proceedings of the 2016 International Conference on Supercomputing* (New York, NY, USA, 2016), ICS '16, ACM, pp. 33:1–33:13.

[43] Wassenberg, J., and Sanders, P. Engineering a multi-core radix sort. In *European Conference on Parallel Processing* (2011), Springer, pp. 160–169.

[44] Wiedemann, D. H. Solving sparse linear equations over finite fields. *IEEE Trans. Information Theory 32*, 1 (1986), 54–62.

[45] Yannakakis, M. Computing the Minimum Fill-In is NP-Complete. *SIAM Journal on Algebraic Discrete Methods 2*, 1 (1981), 77–79.