Parallel Structured Gaussian Elimination for the Number Field Sieve

Charles Bouillaguet[1,*], Paul Zimmermann[2]

[1]Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
[2]Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Received: 7th October 2020 | Accepted: 31st December 2020

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 21 seconds and from 6.7 hours to 2.3 minutes, respectively. This parallel algorithm was used for the factorization records of RSA-240 and RSA-250, and for the DLP-240 discrete logarithm record.*

Keywords: Number Field Sieve, Structured Gaussian Elimination, parallel algorithm

2010 Mathematics Subject Classification: 94A60, 11Y40

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-250 (a 829-bit number) and to compute a discrete logarithm over a 795-bit prime field [5], which are the current records. The NFS algorithm splits into five main phases: polynomial selection, sieving, filtering, linear algebra, and post-processing [8]. 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 merge step takes as input a sparse matrix, and produces a smaller but denser matrix by performing linear combinations of the 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.

The Structured Gaussian Elimination step takes as input a large and (nearly) square sparse matrix M with integer coefficients. In the integer factorization case (IF for short), 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 (DL for short), one is looking for an element of the *right null space* of M over GF(ℓ), where ℓ 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

$$PMQ = \begin{pmatrix} L & \\ A & I \end{pmatrix} \begin{pmatrix} I & \\ & M' \end{pmatrix} \begin{pmatrix} U & B \\ & I \end{pmatrix}, \tag{1}$$

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

The matrices L, U are triangular and invertible (they correspond to the rows or 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 $zM = 0$), we first solve $yM' = 0$ and then (neglecting permutations) it remains to solve $xL + yA = 0$ for x to obtain $(x \mid y)M = 0$.

In the discrete logarithm setting (where we want $Mz = 0$) once we have found y such that $M'y = 0$, it remains to solve $Ux + By = 0$ for x 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.

*Corresponding Author: charles.bouillaguet@univ-lille.fr

Besides speeding up all applications of NFS (factoring integers and computing discrete logarithms [5], computing class groups of number fields [3], ...), SGE can more generally be applied as a preprocessing step before virtually any application of usual iterative methods over an exact domain (integers, rationals, finite fields, ...). Possible applications include the resolution of arbitrary sparse linear systems, computation of the rank or determinant of a sparse matrix, exact computation of selected entries of the inverse matrix, etc.

Related Problems. In the context of sparse *direct* solvers (which compute a sparse factorization), 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: at each step, a new pivot is chosen that minimizes fill-in in M' during this step.

Yannakakis [44] 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 a non-symmetric 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 in Eq. (1) is a subset of the non-zero pattern of the Cholesky factor of M^tM, regardless of the choice of pivots. We refer to Davis' textbook [15] 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 these 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 *massive 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 [6] a greedy parallel algorithm to compute the rank of large sparse matrices over GF(p), a problem which is similar to SGE. Their main idea is to select a priori a set of "structural" pivots that cannot vanish because of numerical cancellation.

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 previously in the reference CADO-NFS implementation, both in terms of size of the output matrices, and in terms of computing time, even sequentially. CADO-NFS [41] 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.

The new algorithm has been used in the factorizations of RSA-240 and RSA-250, as well as in the computation of a discrete logarithm over a 795-bit prime field [5]. Interested readers can check its implementation by looking at `filter/merge.c` (for instance at commit `51989419`) in the CADO-NFS source repository located at:

`https://gitlab.inria.fr/cado-nfs/cado-nfs`

The whole algorithm uses less than 2000 lines of C code. Because an efficient implementation of the algorithm was needed to perform large computations, a lot of attention was given to low-level "details" such as choosing the most adapted data structures, careful memory management, etc. This article tries to explain how the high-level ideas translate to these low-level details.

We assume 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 sets the NFS context, 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 gives details on the data structures used, how the concurrency issues are solved, and how the memory is managed. Finally, Section 6 compares our implementation of the parallel algorithm with the previous single-thread implementation of CADO-NFS on several landmark examples.

2 THE NUMBER FIELD SIEVE, NOTATIONS AND BENCHMARKS

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.

2.1 THE NUMBER FIELD SIEVE

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 or compute discrete logarithms modulo large prime numbers. 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 [38] 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 [13], 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 dimension n, both algorithms solve $Mx = 0$ by performing $\Theta(n)$ matrix-vector products.

These iterative algorithms are Krylov subspace methods, yet they are quite different from their "numerical" counterparts such as the Biconjugate Gradient method [17] 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 [43], for instance, recovers the minimal polynomial $P(X) = X^r + a_{r-1}X^{r-1} + \cdots + a_1X$ 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 $0 = M\left(M^{r-1} + a_{r-1}M^{r-2} + \cdots + a_1I\right)$. The number of iterations of the block Lanczos/Wiedemann algorithm depends solely on the dimension 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 dimension of the current matrix.

This process was initially called "Structured Gaussian Elimination" by Lamacchia and Odlyzko in [29]. 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 [39], Pomerance and Smith propose a very similar method, using different terms (*inactive* and *active* columns instead of heavy and light). As in [29], the number of heavy/inactive columns grows until the active/light part of the matrix collapses.

In [9, 10], Cavallar denotes by "filtering" the combination of structured Gaussian elimination and removing extra rows from the underdetermined matrix (until it becomes nearly determined). Filtering is divided in two parts performed in sequence: "pruning" and "merging". The pruning step eliminates duplicate rows, columns with a single entry, and well-chosen groups of columns with low weight, along with the adjacent rows. The merging step performs some steps of Gaussian elimination.

2.2 NOTATIONS

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,

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)

Table 1: Number Field Sieve Glossary

Matrix	Case	Ref.	Input			Output			
			# rows (n)	Weight (W)	GB	# rows	weight	GB	d
RSA-155	IF	[11]	17M	278M	1.2	3.9M	662M	2.7	170
DLP-1024	DL	[18]	96M	2.4G	20	28M	5.7G	46	200
DLP-240	DL	[5]	150M	4.3G	35	36M	9G	73	250
RSA-240	IF		1.2G	23G	191*	282M	56G	454*	200
RSA-250	IF		1.8G	36G	302*	405M	102G	819*	250

Table 2: Characteristics of benchmark matrices. The space occupied by the matrices in memory (in GB) is computed assuming that the matrices are stored in Compressed Sparse Row representation (cf. Section 5.1) using 32-bit integers, except for the last two (starred) for which 64-bit integers have been used. This format uses one integer per row plus one (resp. two) integer per non-zero entry in the IF case (resp. DL case). The target density d is an input parameter of the algorithm.

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 Tab. 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 $j_1, ..., j_s$, and these elements are $e_1, ..., e_s$. In the integer factoring case, we have $e_k \in \{0, 1\}$, whereas in the discrete logarithm case, we have $0 \leq e_k < \ell$, where ℓ is some integer related to the given discrete logarithm computation.

2.3 BENCHMARK MATRICES

We benchmark the efficiency of our algorithms using two relevant test cases: the factorization of RSA-155 [11], and the kilobit hidden SNFS discrete logarithm computation [18]. We also report on the recent record factorizations of RSA-240, RSA-250 and computation of a discrete logarithm modulo a 795-bit prime [5].

RSA-155 is a number from the RSA factoring challenge. It 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 [7] for the "purge" step.

In 2016, Fried, Gaudry, Heninger and Thomé performed a discrete logarithm computation over a "trapdoored" 1024-bit prime field [18], carefully chosen to look normal while covertly allowing the use of the Special Number Field Sieve (SNFS) to compute discrete logarithms in practice. We denote the input of the merge step by "DLP-1024".

In 2020, both RSA-240 and RSA-250 (two more RSA challenge numbers) have been factored using CADO-NFS. At the same time, a discrete logarithm computation (DLP-240) has been carried out modulo a 240-digit prime.

Table 2 shows the characteristics of these five matrices. A common feature of the input matrices produced by NFS is that rows have about the same density (between 16 and 22.5 in the IF case; between 25 and 30 in the DL case), and column j has density roughly $1/j$ if we neglect logarithmic factors (cf. Figures 1, 4); besides that, entries are distributed more or less randomly. The sudden jumps and drops in column density in Fig. 1 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 discrete logarithm matrices have theoretically to be considered over GF(ℓ) for the SGE step, where ℓ is a parameter of the discrete logarithm computation (usually a prime number of at least 160 bits). However, in practice its coefficients remain quite small during SGE. For example, the maximal coefficient of the DLP-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,

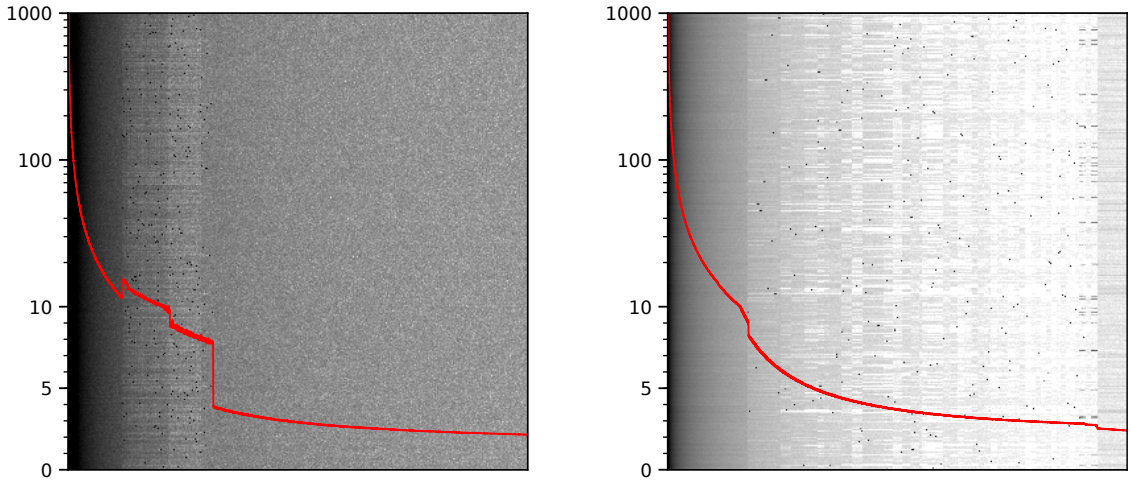

Figure 1: Benchmark matrices (left: RSA-155, right: DLP-1024). The intensity of each pixel is proportional to the logarithm of the density in the corresponding zone of the matrix. The presence of vertical bands is an artifact of lattice sieving. The solid red line shows the number of entries on each column. Note that the vertical scale is linear from 0 to 10 and logarithmic above 10 ; it is truncated at 1000 for better readability (the first columns are fully dense).

so that we do not need to reduce coefficients modulo ℓ, and we can consider them over $\mathbb{Z}$.

3 CLASSICAL STRUCTURED GAUSSIAN ELIMINATION

This section describes how the Structured Gaussian Elimination is classically performed. The only known algorithms in the literature are sequential, and the two open-source tools we are aware of (Msieve [37] and CADO-NFS [41]) provide a sequential SGE step. We follow here [9] and Chapter 3 of [10]. Essentially the same algorithm is also described in Joux's book [24].

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 M' in Eq. (1). In NFS, it is common to use $100 \leq d \leq 250$, for example the "merge" step of RSA-250 started with a (nearly square) matrix of about 1.8G rows, and ended with a matrix of size about 405M, with average density 252 [5]. The goal of SGE is to get a matrix with average density d, and dimension as small as possible. It is typical that SGE reduces the matrix size by a factor 3 to 5.

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

More precisely, the admissible linear combinations of rows which eliminate a column of weight k correspond to the spanning trees of the complete graph with k labelled nodes, of which there are k^{k-2}.

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

[1] In exceptional cases it could increase by one, but this never happens in practice.

$$\begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \\ r_8 \end{array} \begin{array}{c} \begin{array}{cccccccc} j_1 & j_2 & j_3 & j_4 & j_5 & j_6 & j_7 & j_8 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

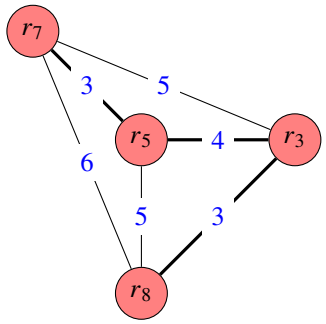

Figure 2: Left: Sample input matrix for SGE over GF(2). Right: The cost graph corresponding to the elimination of column j_5 of this matrix. Using the sparsest row (r_3) as pivot yields a total weight of 12 for the 3 combined relations, while the minimal spanning tree algorithm performs $r_3 + r_5$, $r_5 + r_7$, and $r_3 + r_8$, with total weight 10 (bold edges).

kill non-zero entries. 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 thus be negative.

Let us consider the elimination of column j_7 in Figure 2-left (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, in addition to j_7.

Still with the matrix from Figure 2-left, 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 $i_0, ..., i_{k-1}$. Choose the row, say i_0, with the 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 (called Markowitz cost) 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. Note that for $k = 2$, Eq. (2) gives $c = -2$, if no other cancellation occurs, as mentioned above.

Greedily using the sparsest row as "pivot" to eliminate a column is suboptimal, as illustrated by Figure 2-right. Instead, we use a strategy introduced by Cavallar [10]: 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 [16], then by Cavallar [10, 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, processing the RSA-155 matrix[2] from §2.3, 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 DLP-1024 matrix, 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 these 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 further eliminations much more expensive.

[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.

Algorithm 1 Classical Structured Gaussian Elimination

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

compute the total weight W (# of non-zero elements) of M
compute the transpose of M
for each column j, compute the cost c_j of eliminating it and push (j, c_j) in a priority queue Q
while $W/n < d$ **do**
    pop from Q the column j with a smallest elimination cost
    eliminate column j, updating the matrix M in-place
    update the transpose of M
    update the priority queue Q
    update the number of rows n and the total weight W

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 (Algorithm 1).

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 [25].

Computing the cost given by Eq. (2), as well as eliminating a column requires the ability to iterate over the coefficients in a given row *and* in a given column. The usual data structures to hold sparse matrices do not allow both simultaneously (this will be discussed in Section 5). Therefore, the usual solution is to maintain both M and its transpose: if both can be efficiently accessed by rows, then the columns of M are the rows of the transpose.

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 NEW PARALLEL ALGORITHM

To overcome this situation, we propose a new algorithm which keeps only some minimal data-structures, namely the current matrix. This simpler strategy also turns out to be faster sequentially.

Algorithm 2 Parallel Structured Gaussian Elimination

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

compute the total weight W (# of non-zero elements) of M
for each column j, compute its weight w_j
$w_{\max} \leftarrow 2$, $c_{\max} \leftarrow 0$
while $W/n < d$ **do**
    let S be the submatrix of M formed by assembling the columns j such that $0 < w_j \leq w_{\max}$
    transpose S and store the result in R
    # *A potential column to eliminate is described by a row of* R.
    $L \leftarrow \emptyset$
    for each row of R **do**
        let j be the corresponding column of M
        using Eq. (2), obtain an upper bound c on the cost of eliminating j in M
        if $c \leq c_{\max}$, add j to L
    extract from L a large independent set J of columns with small total cost
    eliminate in-place all columns of J
    update the number of rows n, the column weights w_j and the total weight W
    if all columns of weight 2 have been eliminated **then**
        $w_{\max} \leftarrow \min(32, w_{\max} + 1)$, $c_{\max} \leftarrow c_{\max} + c_{\text{incr}}$

To take advantage from the availability of several processors, we need to be able to eliminate several columns in parallel. To this end, we must deal with two issues:

1. In order to maintain the quality of the output, we would like to stay as close as possible to the sequential greedy strategy, i.e., we would still like to process columns by increasing elimination cost.
2. Eliminating a column modifies several rows; we must ensure that this happens without conflicts, i.e., without uncontrolled concurrent modifications of the same row that would leave the matrix in an inconsistent state.

The new algorithm works by doing *passes* over the matrix. In each pass (steps 5-17 of Algorithm 2), candidate columns with low elimination cost are selected and some of these columns are eliminated in parallel.

4.1 BEING GREEDY IN PARALLEL

To attain our objective of greedily eliminating the "best" columns first, we first select all weight-2 columns; because their cost is always negative, eliminating them is always a win. Once no weight-2 column remains, we impose a limit $c_{\max}$ on the cost of columns that can be eliminated in the current pass. This limit is increased after each pass; we found experimentally that adding $c_{\text{incr}} = 13$ after each pass is close to optimal for integer factorization, and $c_{\text{incr}} = 31$ for discrete logarithm computations.

To detect columns with a low elimination cost, we could compute the actual elimination cost of *all* columns, but this would be very costly. We use two ideas to avoid doing it. First, since the elimination cost of a column is correlated with its weight, we pre-select columns of small weight, and only invest some time to compute the elimination cost for the low-weight columns. Once all weight-2 columns have been eliminated, only columns of weight less than $w_{\max}$ are considered; $w_{\max}$ is then incremented after each pass. We use a hardcoded upper-bound of 32 on the weight of selected columns: if no column of weight ≤ 32 remains, the program stops. Fig. 3 shows the number of columns matching these criteria while processing the benchmark matrices.

In practice, in the CADO-NFS implementation, columns whose weight exceeds 32 (either initially or during the algorithm) are no longer considered in step 5. This threshold of 32 was chosen somewhat arbitrarily, but it turned out to be satisfactory. Just to be sure, we tested increasing it to 64 while processing our two largest matrices (RSA-240 and RSA-250): the impact on the size of the output was clearly negligible.

Because the initial expected weight of column j is proportional to $\approx 1/j$, we expect that sparse columns cannot have low indices. We exploit this to speed up some operations: we compute an array $j_{\min}$, where $j_{\min}[w]$ stores the smallest column j of initial weight w_j less or equal to w. Thus, scanning columns of weight $\leq w$ can be done by starting at column number $j_{\min}[w]$ instead of zero. A drawback of this scheme is that if a column has initial weight greater than 32, it will never be eliminated, even though its weight could drop below 32 during the course of the algorithm. The previous experiments (increasing the threshold to 64) show that this phenomenon plays a negligible role.

Determining the actual cost of eliminating a column using a minimal spanning tree computation as described in §3 is expensive. We therefore use an upper-bound on the actual elimination cost using the "Markowitz cost" from Eq. (2). In such a way, we select columns whose elimination cost stays under a given bound. We found experimentally that using the actual cost is much more expensive and that the increased precision does not save much in the final matrix. Indeed, if the cost estimate we get is so small that the corresponding column is selected, a fortiori it would also be selected with the real cost. What can occur is that we miss a column with a small cost with the minimal spanning tree computation, because the estimate is larger than $c_{\max}$; usually it will be selected in a further pass.

Selecting columns of low-weight, then computing their estimated elimination cost is easily done in parallel, assuming that we can access the current matrix M by rows and by columns.

4.2 CONCURRENT COLUMN ELIMINATION

Once a set L of low-cost columns has been determined, we must actually eliminate them in parallel. For this, we need to address our second issue: 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 2-left, 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.

The solution we propose to overcome this issue is to compute a (large) subset J of *independent columns* of the set L of low-cost 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 2-left, the maximal independent sets of columns are $\{j_1\}$, $\{j_2, j_3\}$, $\{j_2, j_7\}$, $\{j_4\}$, $\{j_5\}$, $\{j_6\}$, and $\{j_8\}$. Once we have found a set of independent columns, then it is straightforward to eliminate them in parallel, since they will read/write separate rows.

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. It is well-known in the sparse linear algebra community that two columns can be eliminated in parallel without problem as long as one is not an ancestor of the other in this tree. The leaves of the column elimination

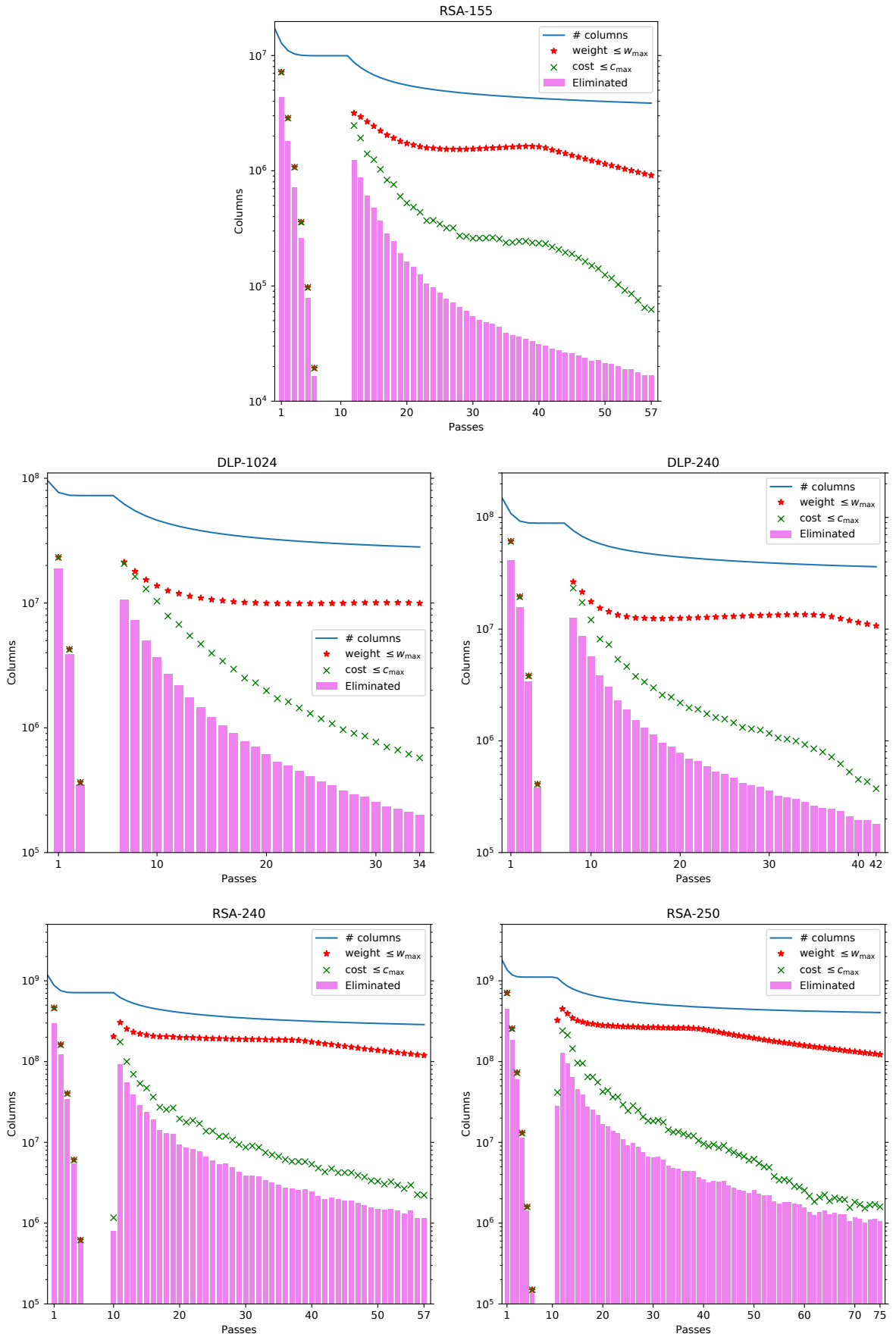

Figure 3: Evolution of column types during the progress of Algorithm 2.

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 2-left 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.

What we need is in fact an 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 $M^t M$. It is well-known that finding a maximum independent set is NP-complete. Therefore we settle for a simple greedy algorithm that only yields a maximal (not maximum) independent set. It constructs an independent set $\mathcal{I}$ incrementally: set $\mathcal{I} \leftarrow 0$; pick a vertex of minimum degree in the graph and add it to $\mathcal{I}$, then remove all its neighbors from the graph; repeat until the graph is empty.

This simple algorithm is known to have an interesting parallelization potential: each step only accesses "local" data in the graph and thus several steps can be done in parallel without conflicts. Some parallel algorithms were derived out of it, see for instance [4] and its references. We use a simple parallel variant.

When selecting a subset of columns that can be eliminated in parallel, we would like to commit to our greedy strategy and favor columns with low elimination cost. This means that, among all possible maximal independent sets, we prefer the ones that minimize the sum of the elimination costs of selected columns; in other terms, we would like a minimum-weight maximal independent set. We find it greedily using a slight variation of the above algorithm: instead of choosing a vertex of minimum degree at each step, we instead pick a vertex (i.e., a column) with minimum cost.

Once a subset of independent columns has been identified, it is eliminated immediately using the minimum spanning tree algorithm described in §3. Steps 13 and 14 of Algorithm 2 are therefore interleaved in our implementation.

The column weights w_j are updated incrementally, whenever a new column is eliminated. In such a way, the cost of updating the column weights is essentially proportional to the number of column eliminations performed. Coefficients are discrete, thus numerical issues that arise with floating-point data, like recognizing zero, cannot happen here.

5 DATA STRUCTURES, CONCURRENCY AND MEMORY MANAGEMENT

In this section, we discuss some lower-level details of our implementation of Algorithm 2. We used OpenMP to easily parallelize our code on shared-memory architectures. Inasmuch as possible we tried to avoid using expensive synchronization primitives. Most of the synchronization between threads happens through the use of *atomic* memory accesses, which are generally supported by the underlying hardware and are therefore efficient. More precisely, we essentially use the OpenMP `atomic capture` and `atomic update` directives, The former enables us to atomically fetch the value of a memory location while updating it, i.e., it does `{x = A[i]; A[i] += y;}` atomically, the latter enables us to update a memory location atomically, for example `A[i] += y`.

5.1 SPARSE MATRIX FORMATS

An $n \times m$ sparse matrix A with nz non-zero entries can be seen as a set of nz triples (i, j, x) such that $A_{ij} = x$. We use two different representations of sparse matrices in memory. For a more in-depth presentation, we refer the reader to [15].

The *compressed sparse rows* (CSR) format groups the triples by row indices (this essentially avoids storing the row indices). The order of entries in each row is unspecified. It uses an array of "row pointers" `Ap` of size $n + 1$ and two arrays `Aj` and `Ax` of size nz. It allows direct access to the rows: the column indices and values of entries on row i are found in `Aj` and `Ax` at locations $\texttt{Ap}[i], \ldots, \texttt{Ap}[i + 1] - 1$. Of course, $\texttt{Ap}[0] = 0$ and $\texttt{Ap}[n] = nz$. This representation (or its "by column" twin) is commonly used in sparse matrix software tools. Note that in the integer factoring case where non-zero entries can only be 1, storing them is superfluous and the `Ax` array is not used. Matrices in CSR format are mostly read-only; we use this representation to store the transpose R in Algorithm 2.

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 $\texttt{Mp}[i]$. Each row is described by its length and an array of pairs (j, x)—again, in the integer factoring case where $x = 1$, storing x is not necessary. 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 2 to store the current matrix M. We enforce that entries on a row are sorted by increasing column indices.

5.2 COMPUTING THE ROWS FOR EACH COLUMN ELIMINATION

Columns with weight less or equal to $w_{\max}$ are potential candidates for elimination, and we need to be able to iterate efficiently over their non-zero entries. This is required to bound the cost of each potential column elimination (step 11 of Algorithm 2) and to actually eliminate these columns (step 14).

Because the current matrix M is stored by rows as explained above, iterating over the entries of a given column is not readily possible. This is why we need to compute the transpose R of low-weight columns. Because R is only read and not modified, it can be stored in CSR format.

Extracting R takes a significant fraction of the total computation time. Algorithm 2 thus presents one of the rare situations where transposing sparse matrices is a bottleneck, and where a sparse parallel transpose algorithm is required.

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 [27, §5.2]). This yields an optimal sparse transposition algorithm that runs in $O(n + m + nz)$ operations. This is usually the algorithm of choice, and it is almost always implemented sequentially. It is for instance the case in several widely-used software packages (UMFPACK [14], CHOLMOD [12], SuperLU [31], Octave [22], SciPy [23], PaStiX [21], . . .). Indeed, transposition is often done only once and is almost never critical for performance.

Wang, Liu, Hou and Feng [42] discuss other applications where the parallel scalability of sparse matrix transposition is potentially a bottleneck, such as the computation of strongly connected components of a sparse graph (for which some parallel algorithms require both the graph and its mirror), or Simultaneous Localization and Mapping (SLAM) problems (which repeatedly require the computation of A^tA for sparse A). Wang, Liu, Hou and Feng also propose two parallel sparse transposition algorithms, which are in fact a parallel bucket sort and a parallel merge sort. Kruskal, Rudolph and Snir [28] suggested earlier to use a parallel radix sort with the intention of obtaining a theoretical parallel algorithm.

We tried several possible solutions: extracting the submatrix S and writing it to memory first then transposing it afterwards or transposing R "on the fly" directly from M, using various parallel transposition algorithms. But in the end, we found a simple procedure to be quite effective.

Fig. 3 shows that 15–30% of the columns of M belong to the extracted submatrix (curve weight $\leq w_{\max}$). To avoid the wasteful situation where the transpose contains 70–85% 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 $j = qinv[k]$. We compute R in two steps:

- Low-weight columns are identified and the "row pointers" of the CSR representation of R are computed. For this, we iterate over the column weight array w in order to accumulate the number of selected columns and the number of entries in these columns. For each column j, if $0 < w_j \leq w_{\max}$, then do: $\texttt{Rq}[j] \leftarrow Rn$, $\texttt{Rqinv}[Rn] \leftarrow j$ (on-the-fly column renumbering), $Rnz \leftarrow Rnz + w_j$, $\texttt{Rp}[Rn] \leftarrow Rnz$, $Rn \leftarrow Rn + 1$ (row pointers computation). Note that after this, the row pointers indicate the first entry *after* the row.

  This amounts to two prefix-sum operations (one on Rq, one on Rp) and is easily parallelized. The parallelization of the prefix-sum operation needs to read the w array twice, and performs twice more arithmetic operations than a naive sequential version. In total, this takes 1–5% of the total wall-clock time needed to obtain R.

- Given these informations, storage to hold R is allocated (the `Rj` array has size Rnz, which has just been determined) and R is written. To actually extract R from the current matrix M, we iterate in parallel over the rows of M. For each row, we copy entries belonging to selected columns to R: if row i contain an entry on column j with $w_j \leq w_{\max}$, then $j' \leftarrow \texttt{Rq}[j]$ is the target row in R. Then, we need to do $k \leftarrow \texttt{Rp}[j']$ and $\texttt{Rp}[j'] \leftarrow k - 1$ atomically (this is done with an `atomic capture` directive); finally, $\texttt{Rj}[k-1] \leftarrow i$.

This procedure is memory-bound: it needs to read the whole M (in our largest examples, this is several hundred of Gigabytes); the memory bandwidth of the machine thus imposes a lower-bound on the time this takes. In addition, this accesses `w`, `Rq` and `Rp` randomly, which causes a lot of cache misses.

We found out that Simultaneous Multi-Threading (called "*Hyperthreading*" commercially by Intel) helped improve the performance of computing R. Usually, contemporary mainstream processors have k physical cores and $2k$ hardware execution contexts. Using $2k$ software threads results in a ≈ 1.5 speedup for computing R compared to using only k threads. Most likely, this enables some threads to make progress while some others are stalled by the memory subsystem.

5.3 SELECTING COLUMNS TO ELIMINATE

Given a list of columns with low elimination cost, we first need to identify an independent subset that can be eliminated in parallel without expensive synchronizations.

We use a parallelization of the well-known greedy algorithm. Because we would like to favor columns with the lowest elimination cost, we first sort the list L of columns by increasing elimination cost. Then, all available threads run the following procedure until the list L has been exhausted: pick the next unprocessed column $j \in L$; begin a transaction; check if all the adjacent rows are free; if not, abort the transaction and give up on the column; if yes, mark the adjacent rows as busy; add j to the set of independent columns; commit the transaction.

The transaction can only commit successfully if all the memory locations it accesses have not been modified by another concurrent transaction during its lifetime. When a transaction commits, all its memory writes are performed atomically and become visible to other transactions. Turning this sketch of algorithm into concrete code requires a way to implement the "transaction" part. We considered several possible ways to do so, including using critical sections, per-row locks, software transactional memory or even (restricted) hardware transactional memory available on high-end processors. All these solutions turned out to be inferior to the simple following lock-free algorithm.

We use a shared array `Busy` of n booleans that identifies the rows already involved in a previously selected column; they are all false initially. When a thread considers a new potential column to eliminate, it first checks if one of the rows where that column appears is busy. If this is the case, then that potential column is discarded (it is in conflict with another previously selected column). So far, this is a read-only procedure that does not require synchronization.

If a column survives the initial scrutiny described above, the rows where that column appears are marked as busy while atomically checking their previous status: this can be done with an `atomic capture` directive. If that second check succeeds, then all rows are now marked as busy and the column can safely be eliminated without conflicts. This second check might fail if some rows have been set as busy by other threads in the meantime, but in practice this is very rare (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.

5.4 ROW ADDITION

When we eliminate a column of weight k, we have $k - 1$ row combinations and one row deletion. If done naively, this would entail $k - 1$ calls to `malloc` to allocate the new rows, and k calls to `free` to discard the old rows. Now if many concurrent threads are 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 these routines scale with many threads.

In addition, two rows of size ℓ_1 and ℓ_2 can be added in $\ell_1 + \ell_2$ operations; this is particularly simple because the column indices in each row are sorted. However, the number of non-zero coefficients in the sum is not known beforehand, because of numerical cancellation: it can be anywhere in the range $[0; \ell_1 + \ell_2]$. We tried several solutions: overallocate the output row to size $\ell_1 + \ell_2$ (this wastes space), compute the sum to a temporary buffer, allocate just the right output size, then copy the buffer to the allocated space (this wastes time).

While we observed that using Google's `tcmalloc` library (thread-caching malloc, a drop-in replacement of the `malloc` and `free` calls) improves the overall performance of Algorithm 2 by a factor about 2, we nevertheless resorted to an in-house memory management strategy relying on stop-and-copy garbage collection. Threads allocate pages of memory of a fixed size to store rows, and each thread has a single active page in which it writes new rows. When the active page is full, the thread grabs an empty page that becomes its new active page. In each page, there is a pointer *ptr* to the beginning of the free space. To allocate b bytes for a new row, it suffices to note the current value of *ptr* and then to increase it by b—if this would overflow the current page, then it is marked as full and a new active page is obtained. Rows are stored along with their index, their size and the list of their coefficients. To delete a row, we just mark it as deleted by setting its index to -1. Thus, row allocation and deallocation are thread-local operations that are very fast. Last but not least, the last allocated row can easily be shrunk (by diminishing *ptr*). This solves one of our problems: when doing a row addition, we allocate space for the sum assuming that no numerical cancellation takes place; once the sum has been computed, we shrink the allocated space to fit the actual size of the sum. This avoids both over-allocating and extra copying.

After each pass, memory is garbage-collected. All threads do the following procedure in parallel, while possible: grab a full page that was not created during this pass; copy all non-deleted rows to the current active page; mark the old full page as empty. Note that moving row i to a different address in memory requires an update of the "row pointer" `Mp`$[i]$; this is the main reason why rows are stored along with their index. When they need a new page, threads first try to grab an existing empty page. If there is none, a new page is allocated from the operating system. There are global doubly-linked lists of full and empty pages, protected by a lock, but these are infrequently

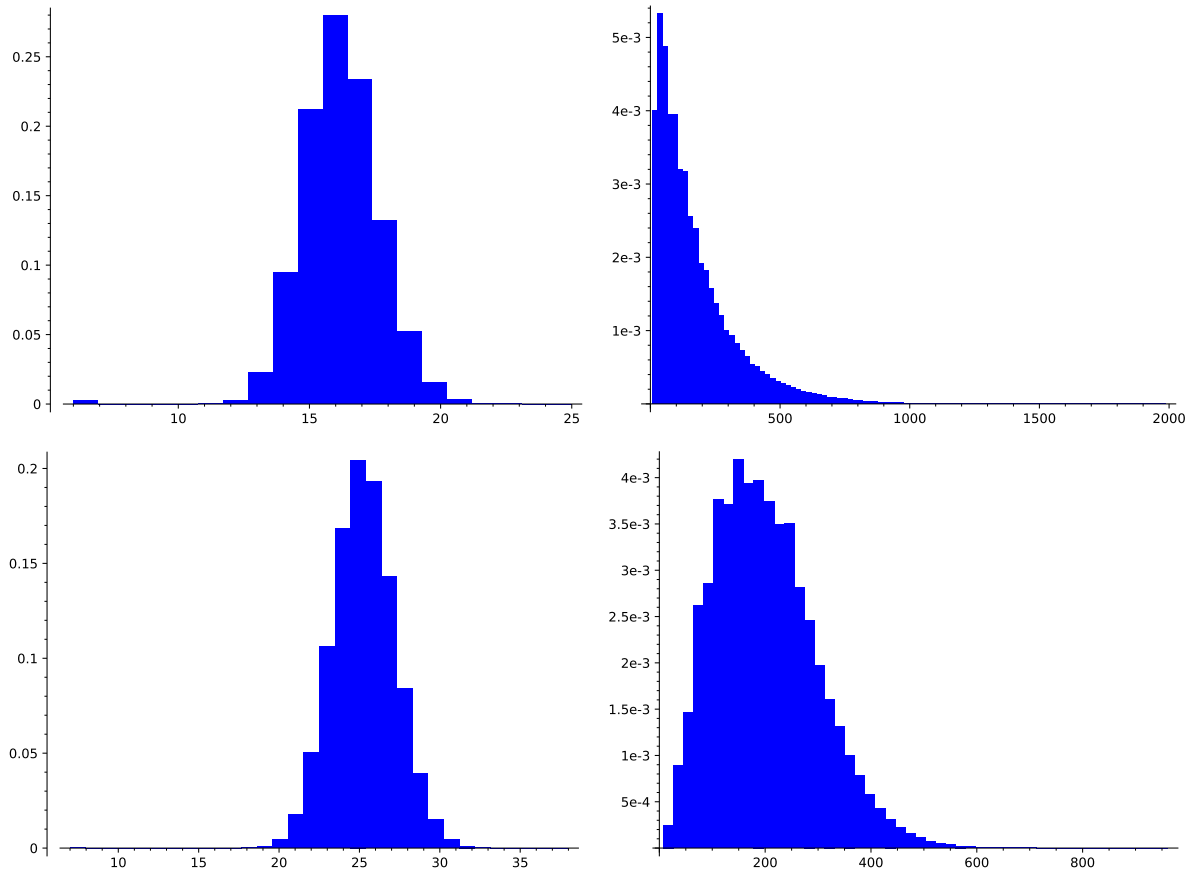

Figure 4: Top: Distribution of row sizes for RSA-155 before (left) and after (right) SGE. Bottom: same for DLP-1024. Note that the scales of the x-axis are not the same between left and right graphs.

accessed. The garbage collection process is memory-bound.

Lastly, 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 rows i and i' (this is in the IF 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. These operations can be performed in parallel using atomic memory updates.

5.5 LOAD BALANCING

Figure 4 shows the distribution of row sizes for two benchmark matrices (RSA-155 for integer factoring, DLP-1024 for discrete logarithm) before and after Structured Gaussian Elimination. We see that in both cases, the row sizes follow somehow a normal distribution at input (with mean around 16 for RSA-155 and 25 for DLP-1024), while the output distributions are far from normal, and can contain quite dense rows (up to 2000 non-zero elements for RSA-155, up to 1000 for DLP-1024).

Column weights are clearly unbalanced (recall Figure 1), and the variance of row weights increases as the algorithm progresses. This mandates dynamic load balancing in some parallel loops. We observed that using the OpenMP `schedule(guided)` clause in parallel `for` loops gives close to optimal results. In only one step of Algorithm 2, namely the computation of the merge costs, we found that `schedule(dynamic,chunksize)` gave better results, with the optimal value of `chunksize` determined experimentally. Dynamic load balancing is used:

- To assemble the transpose R (§5.2), because the row lengths are uneven (see Figure 4). Using a static work repartition yields a $\approx 15\%$ slowdown.
- To estimate the elimination costs using Eq. (2), because the time needed to obtain the estimated cost depends on the weight of the column, and columns of larger index have smaller weight (cf Fig. 1). A static schedule also yields a $\approx 15\%$ slowdown.
- To eliminate the selected columns (§5.3-§5.4), because this also depends on the weights of the columns. A static schedule yields a $\approx 7\%$ slowdown. We used the `schedule(guided)` clause.

RSA-155	2.3.0	This article						
Threads	1	1	2	4	8	16	32	64*
output rows (M)	4.27	3.86	3.86	3.86	3.86	3.86	3.86	3.86
time (s)	2273	651	383	192	98	51	29	21
speedup	-	1	1.7	3.4	6.6	12.8	22.4	31
memory (GB)	6.5	3.5	3.6	3.7	3.8	3.9	4.3	7.1

DLP-1024	2.3.0	This article						
Threads	1	1	2	4	8	16	32	64*
output rows (M)	28.8	28.2	28.2	28.2	28.2	28.2	28.2	28.2
time (s)	24294	4112	3099	1246	613	321	184	140
speedup	-	1	1.3	3.3	6.7	12.8	22.3	29.4
memory (GB)	92.4	51.7	52.4	52.7	53.1	53.6	54.6	55.4

Table 3: Experimental results for RSA-155 (target density: 170 entries/row) and DLP-1024 (target density: 200 entries/row) compared to CADO-NFS 2.3.0. Reported numbers are the median of seven runs.
* The machine has 32 physical cores, runs with 64 threads using two hardware execution contexts per core.

5.6 OUTPUT

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 these lines per thread, otherwise the lock set by `fprintf` will make the other threads wait for their turn, and decrease the overall speedup (we hit that issue in a preliminary version of our implementation).

6 EXPERIMENTS

In this section, we report on the performance of our implementation of Algorithm 2 in the development version of CADO-NFS. We compare it with that of CADO-NFS 2.3.0, which implements Algorithm 1 sequentially. We also report on large computations that pushed our implementation to its limits. For reproducibility, all experiments were performed with commit `51989419` of CADO-NFS.

6.1 PARALLEL SCALABILITY

To measure parallel scalability, 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 turbo-boost disabled. Table 3 demonstrates the efficiency of our parallel algorithm on the RSA-155 and DLP-1024 benchmarks, which run quickly enough to be repeated many times. The "output rows" values are interpolated according to the density before and after the last pass, since Algorithm 2 might not stop exactly when the target density is attained. The time is wall-clock time, not including the time to read the matrix.

We observe from this table that, compared to CADO-NFS 2.3.0, the new algorithm: always produces a smaller final matrix; uses less memory (except for RSA-512 with 64 threads); runs faster even with a single thread[4]. In addition, the new algorithm scales reasonably well with the number of threads, reaching a quasi-optimal speedup of ≈ 30 using the full 32 physical cores of the machine.

Table 4 shows the detailed timings and speedup of the different steps of Algorithm 2, 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.

6.2 RECORD COMPUTATIONS

Algorithm 2 was used to perform the "merge" phase of the record factorizations of RSA-240 and RSA-250, as well as the computation of a 795-bit discrete logarithm (DLP-240) [5]. These computations were carried on a "fat node" with 1512GB of RAM and four Intel Xeon E7-4850 v3 CPUs with 14 physical cores each, running at 2.20GHz. This makes 56 physical cores in total; with "Hyperthreading", this makes 112 hardware execution

[4]With one thread, we run the same algorithm and code than with 2, 4, ... threads. This is suboptimal, because more efficient purely sequential algorithms exist for certain operations, notably prefix-sums.

RSA-155	Time (s)	Speedup					
Threads	1	2	4	8	16	32	64*
Computing R	205.1	1.5	2.9	5.8	11.1	20.5	33.6
Computing L	22.8	1.6	3.1	6.0	10.9	16.3	19.0
Eliminations	392.7	1.9	3.8	7.5	15.0	29.1	41.8
Garbage collection	20.6	1.2	2.4	4.3	6.1	6.1	6.9
Total	651.0	1.7	3.4	6.6	12.8	22.4	31.0

DLP-1024	Time (s)	Speedup					
Threads	1	2	4	8	16	32	64*
Computing R	1048.2	1.2	3.0	6.0	12.0	22.7	34.6
Computing L	148.1	0.9	3.0	6.4	11.8	18.3	20.0
Eliminations	2649.5	1.7	3.7	7.6	15.1	29.8	41.4
Garbage collection	179.7	0.4	1.9	3.9	5.7	5.9	6.3
Total	4112.0	1.3	3.3	6.7	12.8	22.3	29.4

Table 4: Detailed timings and speedup of the different steps of Algorithm 2. Reported numbers are the median of seven runs.
* The machine has 32 physical cores, runs with 48 or 64 threads using two hardware execution contexts per core.

Matrix	RSA-155	DLP-1024	DLP-240	RSA-240	RSA-250
Computing R (s)	5.5	29	57	864	2034
Computing L (s)	1.8	11	20	316	708
Eliminations (s)	5.5	35	64	372	743
Garbage collection (s)	3.9	43	77	599	1365
Total time (s)	18	136	244	2285	5068
Output size (GB)	2.7	46	73	454	819
Peak Mem. use (GB)	12.2	63	95	581	1003

Table 5: Running time of Algorithm 2 on a "fat node" using 56 physical cores and 112 hardware execution contexts. The machine has 1512GB of RAM. The two largest matrices (RSA-240 and RSA-250) require the use of 64-bit integers.

contexts. Because these computations are expensive, we only carried them out using the maximum possible number of threads (we did not try to measure parallel scalability on these large benchmarks).

Table 5 summarizes the results. The "Output size" row indicates the size of the final matrix in memory, in CSR representation, as in Table 2. For RSA-240 and RSA-250, 64-bit integers must be used (there are more than 2^{32} non-zero entries and more than 2^{32} prime ideals in the input matrix, even though less than 2^{32} are actually present in the relations). This automatically increases memory consumption by a factor of two.

For RSA-240 and RSA-250, the main problem was not to run the merge phase as fast as possible, but to actually be able to carry it out without running out of memory in the first place. Indeed, for RSA-250, a single array with one 64-bit integer per row/column already requires about 14GB of memory. Algorithm 2 essentially stores the current matrix, a transpose of a submatrix, and a few such arrays. Compared to the more complex data-structures used in CADO-NFS 2.3.0, the new algorithm is more memory efficient. Assuming a nearly optimal speed-up, we estimate that the computation for RSA-250 would have taken about 80 hours sequentially, which is inconvenient but not impractical.

7 CONCLUSION

In this article, we propose a parallel algorithm for the Structured Gaussian Elimination step of the Number Field Sieve. We show 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). The time needed to process the RSA-155 benchmark (resp. the DLP-1024 benchmark) goes down from 38 minutes (resp. 6.7 hours) with CADO-NFS 2.3.0 (which implements a sequential algorithm) to 21 seconds (resp. 2.3 minutes) with the parallel algorithm, using 64 threads on a 32-core machine.

The goal of SGE is to make a subsequent iterative method faster by applying it to M' instead of M in Eq. (1).

It follows from Eq. (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 consists in computing instead $y_n = x_n(-A \mid I)$, followed by $x_{n+1} = y_n(M_{01} \mid M_{11})^t$ [26]. 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 2 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.

Acknowledgments. 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 [18] who kindly made the data of their computation available, and Jens Gustedt for fruitful discussions about atomic operations. Finally, the authors thank eight out of nine anonymous reviewers so far for their numerous and very helpful comments.

REFERENCES

[1] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. "An Approximate Minimum Degree Ordering Algorithm". In: *SIAM J. Matrix Analysis Applications* 17.4 (1996), pp. 886–905. DOI: 10.1137/S0895479894278952. URL: http://dx.doi.org/10.1137/S0895479894278952.

[2] Shi Bai et al. "Better Polynomials for GNFS". In: *Mathematics of Computation* 85 (2016), pp. 861–873.

[3] Ward Beullens, Thorsten Kleinjung, and Frederik Vercauteren. *CSI-FiSh: Efficient Isogeny based Signatures through Class Group Computations*. Cryptology ePrint Archive, Report 2019/498. https://eprint.iacr.org/2019/498. 2019.

[4] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. "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*. Ed. by Guy E. Blelloch and Maurice Herlihy. ACM, 2012, pp. 308–317. ISBN: 978-1-4503-1213-4. DOI: 10.1145/2312005.2312058. URL: https://doi.org/10.1145/2312005.2312058.

[5] Fabrice Boudot et al. "Comparing the difficulty of factorization and discrete logarithm: a 240-digit experiment". In: *CRYPTO*. Ed. by Thomas Ristenpart Daniele Micciancio. Advances in Cryptology – CRYPTO. Santa Barbara CA, United States: Springer, Aug. 2020, pp. 62–91. DOI: 10.1007/978-3-030-56880-1_3. URL: https://hal.inria.fr/hal-02863525.

[6] Charles Bouillaguet, Claire Delaplace, and Marie-Emilie Voge. "Parallel Sparse PLUQ Factorization modulo p". In: *Proceedings of the International Workshop on Parallel Symbolic Computation, PASCO, Kaiserslautern, Germany*. Ed. by Jean-Charles Faugère, Michael B. Monagan, and Hans-Wolfgang Loidl. ACM, 2017, 8:1–8:10. ISBN: 978-1-4503-5288-8. DOI: 10.1145/3115936.3115944. URL: https://doi.org/10.1145/3115936.3115944.

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

[8] J. P. Buhler, H. W. Lenstra, and Carl Pomerance. "Factoring integers with the number field sieve". In: *The development of the number field sieve*. Ed. by Arjen K. Lenstra and Hendrik W. Lenstra. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 50–94. ISBN: 978-3-540-47892-8.

[9] Stefania Cavallar. "Strategies in Filtering in the Number Field Sieve". In: *Algorithmic Number Theory, 4th International Symposium, ANTS-IV, Leiden, The Netherlands*. Ed. by Wieb Bosma. Vol. 1838. Lecture Notes in Computer Science. Springer, 2000, pp. 209–232. ISBN: 3-540-67695-3. DOI: 10.1007/10722028_11. URL: https://doi.org/10.1007/10722028_11.

[10] Stefania Cavallar. "On the Number Field Sieve Integer Factorisation Algorithm". 108 pages. PhD thesis. University of Leiden, 2002.

[11] Stefania Cavallar et al. "Factorization of a 512–bit RSA Modulus". In: *Proceedings of Eurocrypt*. Ed. by Bart Preneel. Vol. 1807. Lecture Notes in Computer Science. Bruges, Belgium: Springer-Verlag, 2000, pp. 1–18.

[12] Yanqing Chen et al. "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate". In: *ACM Trans. Math. Softw.* 35.3 (2008), 22:1–22:14. ISSN: 0098-3500. DOI: 10.1145/1391989.1391995. URL: http://doi.acm.org/10.1145/1391989.1391995.

[13] Don Coppersmith. "Solving Homogeneous Linear Equations Over GF(2) via Block Wiedemann Algorithm". In: *Mathematics of Computation* 62.205 (1994), pp. 333–350. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2153413.

[14] Timothy A. Davis. "Algorithm 832: UMFPACK V4.3—an Unsymmetric-pattern Multifrontal Method". In: *ACM Trans. Math. Softw.* 30.2 (2004), pp. 196–199. ISSN: 0098-3500. DOI: 10.1145/992200.992206. URL: http://doi.acm.org/10.1145/992200.992206.

[15] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006. ISBN: 0898716136.

[16] Thomas F. Denny and Volker Müller. "On the Reduction of Composes Relations from the Number Field Sieve". In: *Algorithmic Number Theory, Second International Symposium, ANTS-II, Talence, France*. Ed. by Henri Cohen. Vol. 1122. Lecture Notes in Computer Science. Springer, 1996, pp. 75–90. ISBN: 3-540-61581-4. DOI: 10.1007/3-540-61581-4_43. URL: https://doi.org/10.1007/3-540-61581-4_43.

[17] R. Fletcher. "Conjugate gradient methods for indefinite systems". In: *Numerical Analysis*. Ed. by G. Alistair Watson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1976, pp. 73–89. ISBN: 978-3-540-38129-7.

[18] Joshua Fried et al. "A kilobit hidden SNFS discrete logarithm computation". In: *36th Annual International Conference on the Theory and Applications of Cryptographic Techniques - Eurocrypt*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Advances in Cryptology – EUROCRYPT 2017. Paris, France: Springer, 2017. DOI: 10.1007/978-3-319-56620-7_8. URL: https://hal.inria.fr/hal-01376934.

[19] Alan George. "Nested Dissection of a Regular Finite Element Mesh". In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363. ISSN: 0036-1429 (print), 1095-7170 (electronic).

[20] Fred G. Gustavson. "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition". In: *ACM Trans. Math. Softw.* 4.3 (1978), pp. 250–269. ISSN: 0098-3500. DOI: 10.1145/355791.355796. URL: http://doi.acm.org/10.1145/355791.355796.

[21] Pascal Hénon, Pierre Ramet, and Jean Roman. "PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems". In: *Parallel Computing* 28.2 (2002), pp. 301–321. URL: https://hal.inria.fr/inria-00346017.

[22] Sören Hauberg John W. Eaton David Bateman and Rik Wehbring. *GNU Octave version 5.1.0 manual: a high-level interactive language for numerical computations*. 2018. URL: http://www.gnu.org/software/octave/doc/interpreter.

[23] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. 2001–. URL: http://www.scipy.org/.

[24] Antoine Joux. *Algorithmic cryptanalysis*. CRC Press, 2009.

[25] Antoine Joux and Reynald Lercier. "Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the Gaussian integer method". In: *Math. Comput.* 72.242 (2003), pp. 953–967. DOI: 10.1090/S0025-5718-02-01482-5. URL: http://dx.doi.org/10.1090/S0025-5718-02-01482-5.

[26] Thorsten Kleinjung. *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. https://event.cwi.nl/wcnt2011/slides/kleinjung.pdf. 2011.

[27] Donald E. Knuth. *Searching and sorting*. Second. Vol. 3. The Art of Computer Programming. Reading, Massachusetts: Addison-Wesley, 1998.

[28] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. "Techniques for parallel manipulation of sparse matrices". In: *Theoretical Computer Science* 64.2 (1989), pp. 135–157. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(89)90058-3. URL: http://www.sciencedirect.com/science/article/pii/0304397589900583.

[29] B. A. Lamacchia and A. M. Odlyzko. "Solving Large Sparse Linear Systems Over Finite Fields". In: *Advances in Cryptology (CRYPTO'90)*. Ed. by A. J. Menezes and S. A. Vanstone. Vol. 537. Lecture Notes in Computer Science. Springer, 1991, pp. 109–133.

[30] Cornelius Lanczos. "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators". In: *J. Res. Natl. Bur. Stand. B* 45 (1950), pp. 255–282. DOI: 10.6028/jres.045.026.

[31] Xiaoye S. Li. "An overview of SuperLU: Algorithms, implementation, and user interface". In: *ACM Trans. Math. Softw.* 31.3 (2005), pp. 302–325. DOI: 10.1145/1089014.1089017. URL: https://doi.org/10.1145/1089014.1089017.

[32] Joseph W. H. Liu. "The Role of Elimination Trees in Sparse Factorization". In: *SIAM J. Matrix Anal. Appl.* 11.1 (1990), pp. 134–172. ISSN: 0895-4798. DOI: 10.1137/0611010. URL: http://dx.doi.org/10.1137/0611010.

[33] Harry M. Markowitz. "The elimination form of the inverse and its application to linear programming". In: *Management Sci.* 3 (1957), pp. 255–269. DOI: https://doi.org/10.1287/mnsc.3.3.255.

[34] Peter L. Montgomery. "A Block Lanczos Algorithm for Finding Dependencies over GF(2)". In: *Advances in Cryptology — EUROCRYPT'95*. Ed. by Louis C. Guillou and Jean-Jacques Quisquater. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 106–120. ISBN: 978-3-540-49264-1.

[35] Michael A. Morrison and John Brillhart. "A Method of Factoring and the Factorization of F_7". In: *Mathematics of Computation* 29.129 (1975), pp. 183–205. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2005475.

[36] Jason Papadopoulos. *A Self-Tuning Filtering Implementation for the Number Field Sieve*. Slides presented at CADO workshop on integer factorization. http://cado.gforge.inria.fr/workshop/abstracts.html. 2008.

[37] Jason Papadopoulos. *Msieve*. https://sourceforge.net/projects/msieve/. Release v1.53. 2016.

[38] Carl Pomerance. "A Tale of Two Sieves". In: *Notices of the AMS* 43.12 (1996), pp. 1473–1485.

[39] Carl Pomerance and J. W. Smith. "Reduction of Huge, Sparse Matrices over Finite Fields Via Created Catastrophes". In: *Experimental Mathematics* 1.2 (1992), pp. 89–94.

[40] Y. Saad and M. Schultz. "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems". In: *SIAM Journal on Scientific and Statistical Computing* 7.3 (1986), pp. 856–869. DOI: 10.1137/0907058. URL: https://doi.org/10.1137/0907058.

[41] The CADO-NFS Development Team. *CADO-NFS, An Implementation of the Number Field Sieve Algorithm*. http://cado-nfs.gforge.inria.fr/. Release 2.3.0. 2017.

[42] Hao Wang et al. "Parallel Transposition of Sparse Data Structures". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS'16. Istanbul, Turkey: ACM, 2016, 33:1–33:13. ISBN: 978-1-4503-4361-9. DOI: 10.1145/2925426.2926291. URL: http://doi.acm.org/10.1145/2925426.2926291.

[43] Douglas H. Wiedemann. "Solving sparse linear equations over finite fields". In: *IEEE Trans. Information Theory* 32.1 (1986), pp. 54–62. DOI: 10.1109/TIT.1986.1057137. URL: http://dx.doi.org/10.1109/TIT.1986.1057137.

[44] Mihalis Yannakakis. "Computing the Minimum Fill-In is NP-Complete". In: *SIAM Journal on Algebraic Discrete Methods* 2.1 (1981), pp. 77–79. DOI: 10.1137/0602010. eprint: http://dx.doi.org/10.1137/0602010. URL: http://dx.doi.org/10.1137/0602010.