ISSN 2007-9737

A Parallel Strategy for Solving Sparse Linear Systems over Finite Fields

Luis Rivera-Zamarripa[1], Gora Adj[2], Carlos Aguilar-Ibañez[3], Nareli Cruz-Cortés[3], Francisco Rodríguez-Henríquez[1,4]

[1] Technology Innovation Institute,
Cryptography Research Centre,
United Arab Emirates

[2] Universitat de Lleida,
Department de Matemàtica,
Spain

[3] Instituto Politécnico Nacional,
Centro de Investigación en Computación,
Mexico

[4] CINVESTAV,
Computer Science Department,
Mexico

luis.riverazamarripa@tuni.fi, gora.adj@udl.cat, nareli@cic.ipn.mx,
caguilari@cic.ipn.mx, francisco@cs.cinvestav.mx

Abstract. In this paper we describe a number of parallel techniques that were applied to the problem of finding the null-spaces of thousands of large sparse matrices. This collection of matrices were derived from the discrete logarithm problem attack over the finite field $\mathbb{F}_{3^{6\cdot 509}}$ recently carried out by Adj et al. in [2]. Our software library was mainly executed in the supercomputer ABACUS [7], where in total $21,870$ large sparse linear algebra systems were processed. Solving those linear algebra problems involved a computational effort of over 138 core-years, requiring a memory space of over 645 gigabytes to store the corresponding vector solutions.

Keywords. Linear algebra, finite field, parallel computing.

1 Introduction

The computational problem of solving large sparse linear systems of equations shows up in several computer science subdisciplines. An emblematic application of sparse linear systems occurs in the process of finding the *PageRank* vector of the so-called *Google matrix*, which is a sparse Markov matrix with dimension of about ten billions ($\approx 2^{33}$) of rows and columns. Besides its famous *Google* application, the PageRank algorithm has found its way for solving problems in the areas of biology and social network analysis, among others [24].

Cryptography is another important application where the problem of solving large sparse linear systems shows up.

Indeed, when one wants to factorize extremely large integers, or to solve the Discrete Logarithm Problem (DLP) using state-of-the-art index-calculus methods, one needs to solve high dimension linear algebra problems [1, 4].

For example, solving the DLP problem over small characteristic finite fields, can lead to thousands of

linear algebra problems some of them containing up to 266,086 equations and variables and up to 289 million ($\approx 2^{28.1}$) nonzero entries in the corresponding sparse matrix.

Further, the sought solution for this system of equations must be exact because the matrices arising from these cryptanalysis applications are defined over the integers modulo a prime number [2, 10, 9, 13, 16].

In the context of cryptographic applications the aforementioned linear algebra problem can be formally stated as follows. Let $\mathcal{B}$ be an $N \times N$ square matrix defined over a finite field $\mathbb{F}_p$, where p is a large odd prime. The linear algebra problem addressed in this work consists of finding a non-trivial vector $w \in \mathbb{F}_p$ (with $w \neq 0$) such that:

$$\mathcal{B} \cdot w = 0, \tag{1}$$

were both, w and 0 in Equation (1) are considered to be $N \times 1$ vectors with integer entries in the range $[0, p-1]$. We stress that Eq.(1) has non-zero solutions if and only if $\mathcal{B}$ is a singular matrix, i.e., the determinant of $\mathcal{B}$ is zero. Equivalently, Eq.(1) has non-zero solutions if and only if $\mathcal{B}$ has not full rank.

The space of solutions of Eq.(1) is sometimes referred as the kernel or null-space of the matrix $\mathcal{B}$. The families of matrices studied in this paper will always have a kernel of dimension one, which basically means that for a given matrix $\mathcal{B}$, there is only one non-trivial vector solution w that annihilates $\mathcal{B}$. This means that there is a unique vector w such that the matrix-vector product, $\mathcal{B} \cdot w$, is equal to the zero vector.

Moreover, the case of interest in this paper are matrices with a dimension N of tens, and even hundreds of thousands columns and rows.

At the same time, the average per-row density of non-zero elements in these matrices will be very low. In the examples analyzed in this work this non-zero row density, which in the reminder of this paper will be denoted as λ, will always be less than one thousand non-zero entries, where an overwhelming majority of these entries take a small integer value.

Eq. (1) can be solved using standard Gaussian elimination at a computational complexity cost of $O(N^3)$.[1]

The main drawback of following this approach is that as the Gaussian elimination algorithm proceeds, the matrix being processed becomes more and more dense, which implies that its sparseness will get completely destroyed after some few iterations. This situation negatively affects both, the computational and the memory cost of solving this problem. For instance, a dense version of the matrices computed in this paper may need up to 8 Tera Bytes of memory per matrix.

Fortunately, several algorithms have been reported in the literature that can find the kernel of a matrix without manipulating it, but rather dealing with it using a so-called black box model [19]. Some of the most relevant algorithms that solve Eq. (1) without perturbing the sparseness of the matrix $\mathcal{B}$ include: Structured Gaussian elimination (also known as intelligent Gaussian elimination) [15], Conjugate gradient and Lanczos algorithm [21, 29], and Wiedemann algorithm [17, 19, 20, 28].

In this paper we will study the latter approach, which can find the solution vector w of Eq.(1) at a computational cost of about $3N$ matrix-vector products, where the cost of each such product is of about $\lambda \cdot N$ integer additions and multiplications. This yields an approximately overall cost of some $3\lambda \cdot N^2$ integer additions and multiplications.

Furthermore, Wiedemann algorithm enjoys the extra plus of being amenable for parallelization, and is said to be *Las Vegas* randomized, in the sense that it never outputs an incorrect solution after its execution, although it might sometimes fail to provide a non-trivial solution [19].

Without loss of generality, in this work we will generally consider that the whole arithmetic is performed modulo an 804-bit prime number p. Considering that contemporary processors architectures have a 64-bit word-size, it becomes necessary to represent the operands as an array of $n = \lceil \frac{804}{64} \rceil = 13$ sixty-four–bit words. Moreover, every arithmetic operation (such as addition,

[1] Notice that for a matrix $\mathcal{B}$ of size $N = 266,086$, this would imply a humongous computational effort of some $\approx 2^{54}$ operations.

multiplication, etc), must give an integer result within the range of $[0, p-1]$.

Hence, modular reduction should be periodically applied after one or more integer arithmetic operations have been performed [12].

As it has been mentioned, the core operation of Wiedemann algorithm, is the multiplication of a randomly looking column vector of dimension N, by each row of the matrix being handled.

Since a matrix-vector product is the single most demanding operation in Wiedemann algorithm, we carefully optimize the computation of this operation both, for CPU and GPU platforms.

Our contributions can be described as following. We present the efficient computation of the kernels associated to $21,870$ linear algebra systems that happens to be large and sparse and that are defined over an 804-bit finite field. To solve this challenging task, we describe the specifics of our Wiedemann algorithm implementation in a multi-core environment hosted by the supercomputer ABACUS, and in a many-core GPU architecture. The results achieved in this work were a crucial building block for achieving the record computation of the discrete logarithm problem over the field $\mathbb{F}_{3^{6\cdot 509}}$ reported by Adj et al. in [2].

The remainder of this paper is organized as follows. In Sec. 2, the operand representation and field arithmetic employed in this work are briefly explained. In Sec. 3 a basic version of Wiedemann algorithm and several parallel variants are discussed. Then, in Sec. 4, we report the main implementation aspects related to the computation of the kernels of thousands of large sparse matrices in the supercomputer ABACUS and other platforms. we draw our concluding remarks in Sec. 5.

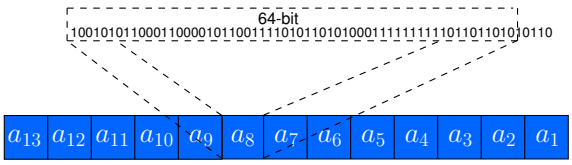

Fig. 1. Representation of an 804-bit field element

2 Finite Field Arithmetic

2.1 Field Element Representation

As it was mentioned in the Introduction, all the arithmetic operations are performed over a finite field $\mathbb{F}_p$. Following the setting specified in [2], we use the 804-bit prime $p = \frac{3^{509}-3^{255}+1}{7}$. Since our CPU servers have a 64-bit wordsize, we are forced to represent the field elements using an integer array of $\lceil \frac{804}{64} \rceil = 13$ words as depicted in Figure 1. Notice that the most significant word of such array only utilizes 36 bits.

Algorithm 1 Barrett Reduction as presented in [11]

Require: p, $b \geq 3$, $k = \lfloor log_b\ p \rfloor + 1$, $0 \leq t < b^{2k}$, and $\mu = \lfloor b^{2k}/p \rfloor$
Ensure: $r = t \bmod p$
1: $\hat{q} \leftarrow \left\lfloor \lfloor t/b^{k-1} \rfloor \cdot \mu / b^{k+1} \right\rfloor$;
2: $r \leftarrow (t \bmod b^{k+1}) - (\hat{q} \cdot p \bmod b^{k+1})$;
3: **if** $r < 0$ **then**
4: $\quad r \leftarrow r + b^{k+1}$;
5: **end if**
6: **while** $r \geq p$ **do**
7: $\quad r \leftarrow r - p$;
8: **end while**
9: **Return** r;

2.2 Field Multiplication

Let $a, b \in \mathbb{F}_p$. A classical method for obtaining the modular multiplication defined as $c = (a \cdot b) \bmod p$, consists of performing the integer multiplication $t = a \cdot b$ first, followed by a reduction step, $c = t \bmod p$. It may appear that this reduction step involves a division by p, which is a relatively expensive operation.

One can do better by using the Barrett reduction algorithm [5]. The Barrett reduction algorithm is based in the following observation: Given $t = Qp + R$, where $0 \leq R < p$, the quotient $Q = \left\lfloor \frac{t}{p} \right\rfloor$ can be written as:

$$Q = \left\lfloor \lfloor t/b^{k-1} \rfloor \cdot \left(b^{2k}/p\right) \cdot \left(1/b^{k+1}\right) \right\rfloor$$
$$= \left\lfloor \lfloor t/b^{k-1} \rfloor \cdot \mu \cdot \left(1/b^{k+1}\right) \right\rfloor.$$

For efficiency reasons, the method requires to precompute a per-modulus constant parameter:

$$\mu = \left\lfloor \frac{b^{2k}}{p} \right\rfloor,$$

where b is usually selected as a power of two close to the word size processor, and $k = \lfloor log_b\ p \rfloor + 1$.

Notice that upon finding Q, the remainder R can be obtained as, $R = t - Qp \equiv t \bmod p$.

Algorithm 1 presents the Barrett reduction, where the remainder R is computed at a cost of no more than two 804-bit integer multiplications.

2.3 Main Matrix-Vector Product Arithmetic Operations

The core arithmetic operation needed to perform a matrix-vector product is $v_i = (v_i + \alpha \cdot w_i) \bmod p$. This operation involves the computation of additions, multiplications, and a modular reduction over the integers.

For the set of matrices considered in this work, the coefficients α happen to be small values. Indeed, in some matrices, for any given row, all the non-zero entries but the first one are equal to one. Besides, the first entry is a 804-bit integer.

We took advantage of this fact (as shown in Fig. 2) by using the function *row_multqu* that starts a row-vector multiplication computing a field multiplication using the function *Fp_mulC.* The result of this function is stored in the accumulator C. Then, all non-zero values are added and stored in the accumulator C using the *in_addN* function.

Taking advantage of the so-called lazy reduction technique, this addition function does not perform any modular reduction at all. Therefore, at the end of the function *row_multqu*'s loop, our *Barret_Reduction* function is invoked to guarantee that the final result (labelled as *out*) has an integer value into $[0, p-1]$.

For the families of matrices where α has a small value different than one, we compute the multiplication $\alpha \cdot w_i$ by performing a shift-and-add strategy. This way, we avoid performing costly full field multiplications as much as possible.

```
void row_multqu(elt out, unsigned int row, elt_v W[], elt p)
{
        unsigned int j, index = 0;
        ui64 C[WORDSIZE];

        // The first entry of the matrix row and vector W are multiplied
        // the multiplication output is saved in the accumulator C
        Fp_mulC(C, Vd[row].x, W[0].x, p);

        j = 1;
        index = matrix[row][j];

        // Main loop. Since the non-zero entries of the Matrix row contain only ones
        // All the values of the vector W are added
        while(index != -1)
        {
                in_addN(C, C, W[index - 1].x);
                j++;
                index = matrix[row][j];
        }
        // We apply Lazy reduction: only one Barret Reduction is applied.
        Barret_Reduction(out, C, p);
}


void vector_matrix_mult_thread(Matrix T[], Vector W[], int ini, int final, elt p)
{
        unsigned int i;

        // A Matrix-vector multiplication is performed but only in a
        // sub-region of the matrix B that starts at row ini and ends at row final
        for(i=ini; i<final; i++)
        {
                vector_row_mult(T[i].x, i, W, p);
        }
}

void vector_matrix_mult_multicore_main4c(Matrix T[], Vector W[], elt p)
{
        int div = Nm/4;

        // This code distributes the matrix-vector computation among four cores
        #pragma omp parallel sections num_threads(4)
        {
                #pragma omp section
                vector_matrix_mult_thread(T, W, 0, div, p);
                #pragma omp section
                vector_matrix_mult_thread(T, W, div, 2*div, p);
                #pragma omp section
                vector_matrix_mult_thread(T, W, 2*div, 3*div, p);
                #pragma omp section
                vector_matrix_mult_thread(T, W, 3*div, Nm, p);
        }
}


void krylov(elt_v *o, elt p)
{
        unsigned int i;

        // This code computes the sequence S by invoking the Matrix-vector multiplication
        // 2N times, where N is the matrix Dimension.
        for(i=0; i < 2*N; i++)
        {
                vector_matrix_mult_multicore_main2c(V[1], V[0], p);
                types_copy(o[2*i].x, V[1][0].x);
                vector_matrix_mult_multicore_main2c(V[0], V[1], p);
                types_copy(o[2*i+1].x, V[0][0].x);
        }
}
```

Fig. 2. OpenMP code of the Krylov Sequence computation distributed into four cores

3 Wiedemann Algorithm

In this section, we describe our strategy for computing the kernel of thousands of sparse matrices.

We begin by describing the mathematics behind the Wiedemann algorithm and continue giving the main parallel strategies followed to compute the kernel of thousands of large sparse matrices.

We are targeting multi-core CPU servers, the super-computer ABACUS [7], and GPU Kepler architecture TITAN cards.

3.1 Basic Version of Wiedemann Algorithm

Let w be a randomly chosen column N-vector defined over $\mathbb{F}_p$, and let u be the unit row N-vector, whose first entry is equal to one and all other entries are zero. Then, given an $N \times N$ matrix $\mathcal{B}$ defined over $\mathbb{F}_p$, a basic version of Wiedemann algorithm [28], computes the $2N-$term Krylov sequence $\mathcal{S}$ defined as:

$$\mathcal{S} = \{u \cdot \mathcal{B}^i \cdot w\}_{1 \leq i \leq 2N}.$$

Notice that each one of the coefficients in the sequence $\mathcal{S}$ is an integer in the field $\mathbb{F}_p$. In a second phase, the Berlekamp/Massey algorithm [6, 22] can be employed to find the minimal polynomial $f^{\mathcal{B},w}$ from the sequence $\mathcal{S}$ with high probability, at a computational cost of $O(N^2)$ field operations. The monic polynomial:

$$f^{\mathcal{B},w}(\lambda) = \lambda^l + c_{l-1}\lambda^{l-1} + \ldots + c_0,$$

is known as the minimum l-degree polynomial of w, with $l < N$ [20]. In particular, the polynomial $f^{\mathcal{B},w}$, satisfies the following relations:

$$f^{B,w}(\mathcal{B})w = 0,$$
$$\mathcal{B}^{\delta}(\mathcal{B}^{l-\delta}w + c_{l-1}\mathcal{B}^{l-\delta-1}w + \ldots + c_{\delta}w) = \mathcal{B}^{\delta}\hat{w} = 0. \quad (2)$$

The exponent δ is strictly greater than zero for singular matrices $\mathcal{B}$. Then, for some integer t with $1 \leq t \leq l$, and $\mathcal{B}^t \cdot \hat{w} = 0$, which implies that:

$$w = \mathcal{B}^{t-1}\hat{w} \neq 0,$$

hence, $\mathcal{B} \cdot w = 0$.

Therefore, in a third phase of Wiedemann algorithm, the minimum polynomial $f^{\mathcal{B},w}$ as shown in Eq. (2) is evaluated, and this evaluation yields the desired solution w. Summarizing, Wiedemann algorithm requires the computation of three main phases [8, 9]:

1. **Krylov sequence**: To obtain the Krylov sequence $\mathcal{S}$ as defined above, we must compute $2N$ matrix-vector products of column N-vectors by a matrix $\mathcal{B}$. When we compute the first of such products, we get as output a vector w', whose first entry corresponds to the first coefficient of $\mathcal{S}$. The column N-vector w' becomes then the input vector for the next matrix-vector product. After computing the second product, the first entry is obtained and stored in the second position of the sequence $\mathcal{S}$, and so on until the $2N$ coefficients of the Krylov sequence $\mathcal{S}$ are obtained. The Krylov phase just described is illustrated in Figure 3. We stress that $\mathcal{S}$ is computed without manipulating the matrix $\mathcal{B}$. Furthermore, Appendix A depicts the C coded function *krylov*, used to computed this phase of the algorithm, where the product $\mathcal{B}^{2N} \cdot w$ is computed by iteratively exploiting the identity,

$$\mathcal{B}^{2N} \cdot w = \underbrace{\mathcal{B} \cdot \ldots (\mathcal{B} \cdot (\mathcal{B} \cdot w))}_{2N \text{ times}}.$$

Once again, the first coefficient of each one of the $2N$ matrix-vector products shown above, must be sequentially appended to the Krylov sequence $\mathcal{S}$.

2. **Obtaining the minimum Polynomial** $f^{\mathcal{B},w}$: From the Krylov secuence $\mathcal{S}$, the Berlekamp-Massey algorithm [22] can be used to find the corresponding minimum polynomial $f^{B,w}$. The Berlekamp-Massey algorithm is a classical procedure that has been extensively studied in the literature (see for example [6, 8, 26]).

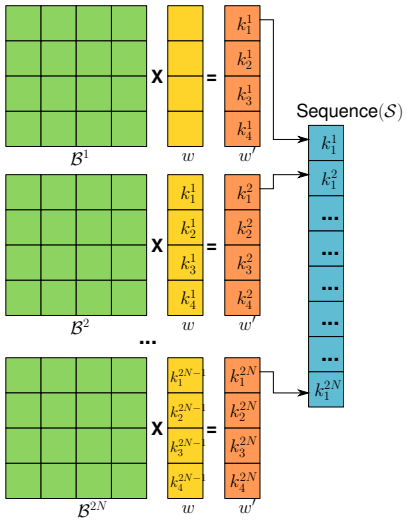

Fig. 3. First step of Wiedemann algorithm: The computation of the Krylov Sequence $\mathcal{S}$

The complexity of the Berlekamp-Massey algorithm is $O(N^2)$. However, a fast version based on a variant of the extended greatest common divisor algorithm, known as the half-gcd algorithm, can compute the minimum polynomial with a complexity of only $O(N \log(N)^2)$ [15, §3.5.2].

3. **Minimum Polynomial Evaluation**: It finds the solution $\widehat{w}$ (see Eq. (2)). If $\widehat{w} \neq 0$, then for some integer t, with $1 \leq t < N$, $\mathcal{B}^t \widehat{w} = 0$, but $w = \mathcal{B}^{t-1}\widehat{w} \neq 0$. This implies that, $\mathcal{B} \cdot w = 0$. Let us recall that the minimum polynomial is evaluated by iteratively performing matrix-vector multiplications until the output vector is equal to zero. This process executes at most $N - 1$ matrix-vector multiplications.

Hence, the computational cost of Wiedemann algorithm is dominated by phases 1 and 3, with an approximately overall cost of about $3N$ matrix-vector multiplications. This basic version described here, can be accelerated by applying parallel strategies as explained next.

3.2 Exploiting Parallelism Opportunities in Wiedemann Algorithm

In 1994, Coppersmith famously presented a parallel version of Wiedemann algorithm [8]. The next few years, Kaltofen and Villard published in [18, 27], a comprehensive analysis of Coppersmith's block version of this procedure. Arguably the first full implementation of this algorithm was reported by Kaltofen and Lobo in [19].

In a nutshell, Coppersmith idea was to simultaneously process n random vectors w and to store m coefficients of the corresponding matrix-vector product, with $m \geq n$. To this end, the vector w becomes a matrix of n vectors of dimension $N \times n$, whereas the vector u becomes a canonical (or highly sparse), matrix of dimension $m \times N$. Coppersmith then ingeniously generalized the Berlekamp-Massey algorithm, observing that the linear recurrence can be determined by the first $\lceil \frac{N}{m} \rceil + \lceil \frac{N}{n} \rceil$ coefficients of the sequence $\mathcal{S}$.

This setting has two advantages. Firstly, the total number of iterations is reduced from $3N$ matrix-vector products (for the sequential version), to $\lceil \frac{N}{n} \rceil + \lceil \frac{N}{m} \rceil$ matrix-block-of-n-vectors products [15, §3.5.3]. Secondly, the computation of each one of the matrix-block-of-n-vectors products can be readily parallelized using n different cores with or without shared memory capabilities.[2] If however, one only has one core available to process the kernel of a matrix $\mathcal{B}$, then the advantage of these approach significantly diminishes.

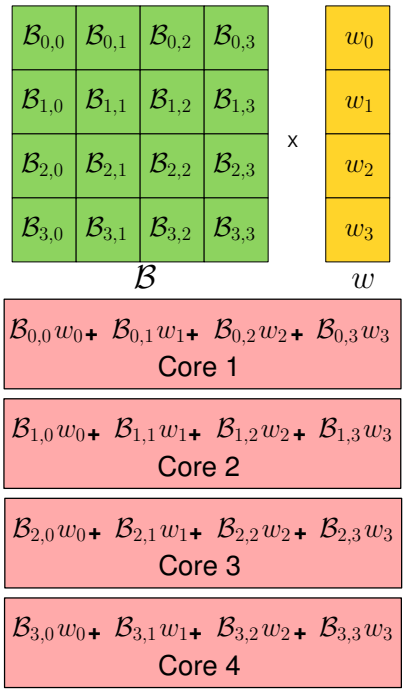

Fig. 4. Parallel computation of the matrix-vector product

Let us recall that for the main case study of this work, we are interested in computing the kernel of thousands of matrices that were obtained from the discrete logarithm problem attack over the finite field $\mathbb{F}_{3^{6 \cdot 509}}$ reported in [2]. Furthermore, we had access to several multi-core servers as well as the super-computer ABACUS [7] (see Table 1 for more details). Under this scenario, we found that the following parallelization setting was simpler and that gave us a competitive performance as

[2] In the latter case at the price of providing a copy of the matrix $\mathcal{B}$ to all the participant processing cores.

Table 1. Hardware used for solving the linear algebra problems

Platform	Nodes	Cores	Frequency (@ GHz)	Compiler
Pakal	1	20	2.4	gcc 4.8.2
Abacus I	318	8904	2.6	gcc 4.8.3
Titan	1	2688	0.876	CUDA assembly

Table 2. Main features of the three families of linear algebra systems under study

	Quadratics	Cubics	Quartics
Number of matrices	1	728	21,141
Dimension N	266,086	177,391	≈ 55 K
Per-row density of non-zero entries(λ)	1,086 –1,096	226 – 262	112
number of nonzero entries	≈ 290M	≈ 43M	≈ 6M

compared with the parallel strategy followed in other works such as [15].

The matrix-vector product $\mathcal{B} \cdot w$ can be trivially parallelized on k cores by partitioning the rows of A into k submatrices $A_1, A_2, \ldots, A_k$ and computing $A_i v$ on the ith core.

Figure 4 depicts such partitioning for the case when four processing cores can be assigned to compute the kernel of a matrix $\mathcal{B}$. In § 4.2 we discuss in detail how this strategy was coded in C using the OpenMP environment.

4 Implementation Aspects

Wiedemann algorithm was successfully implemented for computing the kernel of $21,870$ large and sparse linear algebra problems.

To this end, we use the Pakal CPU server,[3] the supercomputer ABACUS and a GPU Kepler architecture TITAN card, whose salient features are summarized in Table 1.

We begin this section by presenting the families of matrices considered in this work. Then, we give our theoretical cost estimates of computing the kernel associated to those matrices along with the real computational time obtained from our experiments.

[3]available to us in the Computer Science Department of CINVESTAV

4.1 Matrix Representation

In this work we only stored the non-zero values of a sparse matrix $\mathcal{B}$. This was accomplished by adopting a row representation of ordered pairs of the form, (mat_ind_i, mat_val_i), which indicate the column of a non-zero entry in the i-th row of $\mathcal{B}$ and its integer value, respectively. Using this representation, a matrix $\mathcal{B}$ is represented using about $\lambda \cdot N$ pairs, where λ is matrix $\mathcal{B}$'s per-row average number of non-zero entries.

For a comprehensive analysis of different techniques to store a sparse matrix, the interested reader is referred to [15, § 5.2].

4.2 Parallel Programming API

There exist several software tools for performing parallel computing, such as, Message Passing Interface (MPI), CUDA for GPU platforms, OpenMP, among others. In this work, we chose to use OpenMP [23] for our CPU server Pakal and the ABACUS supercomputer, and CUDA for our GPU Kepler architecture Titan card (refer to Table 1).

4.2.1 OpenMP for CPU Platforms

OpenMP defines a set of instructions for the C/C++ and Fortran programming languages, which makes easier to execute shared-memory parallelism.

A typical program using OpenMP instructions, starts with a single active master thread. Then,

when the master thread execution reaches an OpenMP directive, such as the pragma directive *omp parallel*, it creates a set of threads specified by the programmer, where each of these threads will process a data set. Once that all the threads have performed their job, the master thread destroys the threads so created, and then it continues the sequential processing.

The segment that is processed in parallel depends on the given directive. For example, the directive *sections* divides the code into blocks and distributes each one of them among the participants threads. This directive is not sequential.

The parallel matrix-vector product strategy depicted in Figure 4, was implemented in C using OpenMP as reported in Appendix A. Indeed, the C coded function *vector_matrix_mult_thread*, performs a matrix-vector multiplication, but only in a sub-region of the matrix $\mathcal{B}$ that starts at row ini and ends at row $final$. Then, the C coded function *vector_matrix_mult_multicore_main4c*, issues the openMP pragma directives that evenly distribute the matrix-vector computation into four different cores.

4.2.2 CUDA for GPU Platforms

Graphic Processing Units (GPU), are massively parallel processors consisting of hundreds of cores. By taking advantage of the highly parallel architecture of the GPUs, one can speed up several computations where high computing power is required.

A typical GPU is composed by several processors. Each processor has a large number of cores, and each core execute threads. A GPU architecture groups the threads into blocks, and the blocks into a grid.

In the case of the implementation reported in this work, for each block we assign 32 threads to process 32 adjacent rows of the sparse matrix being handled. This process was repeated until all the columns of the matrix have been assigned. We implemented all the field arithmetic (addition, subtraction, multiplication and reduction) using CUDA assembly.[4]

4.3 Families of Matrices

We considered three different families of matrices. These matrices arise from the index-calculus method used in [2] to attack the discrete logarithm problem over the field $\mathbb{F}_{3^{6\cdot 509}}$, were the authors classified the matrices generated by their attack into three classes, namely, quadratics, cubics and quartics families of matrices.

— **Family of Quadratics**: A single N-dimension sparse matrix with $N = 266,086$. Its non-zero elements can take the integer values $\pm 1, 2, 3$ or 4.

— **Family of Cubics**: 728 N-dimension sparse matrices with $N = 177,391$. For each one of these matrices, all the non-zero entries of any given row, have a value equal to one, except for the first entry that has a randomly looking 804-bit integer value.

— **Family of Quartics**: $21,141$ N-dimension sparse matrices with $N \approx 55,000$. For each one of these matrices, all the non-zero entries of any given row, have a value equal to one, except for the first entry that has a randomly looking 804-bit integer value.

Following the convention used in [2, 3], the first column of each one of the Cubics and Quartics matrices contains a large 804-bit non-zero entry. The salient features of these three systems are summarized in Table 2.

From the characteristic of each matrix family, one can estimate the number of operations that a sequential execution would require for solving each linear algebra system.

[4] CUDA is the NVIDIA GPU hardware and software infrastructure that enables the execution of C programs.

Table 3. A comparison of the GPU and CPU elapsed time for computing the kernels of the three linear algebra system families

	Quadratics	**Cubics**	**Quartics**
Matrix size(N)	$\approx 266K$	$\approx 177K$	$\approx 55K$
# Matrices	1	728	21141
Cores (CPU)	20	7	2
Threads (GPU)	3584	3584	3584
Linear Algebra CPU (hours)	216	$\approx$ 74.4	$\approx$ 19.62
Linear Algebra GPU (hours)	223.2	$\approx$ 24.3	$\approx$ 0.9605
Speedup	-	3.06	20.42
MinPoly (hours)	64.8	$\approx$ 39.6	$\approx$ 2.64

Table 4. A comparison of linear algebra solvers in GPU platforms

Author	**GPU**	**Matrix size**	**Modulus size**	**Timing (hours)**
[14]	GeForce GTX 580 @1544MHz, 512 cores	650K	217 bits	16
This work	Titan, 2688 cores	531K	155 bits	18.8

Table 5. CPU times of the kernel computation of all the sparse matrices derived in [2]

Computation stage	cores per matrix	**CPU time (years)**	**CPU frequency (GHz)**
1 Quadratics matrix	20	0.49	2.40
728 Cubics matrices	7	43.28	2.60
21,141 Quartics matrices	2	94.70	2.60
Total CPU time (years)	138.47		

4.3.1 Finding the Kernel of the Quadratics Matrix

Since the non-zero values of the Quadratics matrix are smaller than four, each scalar multiplication can be performed using a shift-and-add approach. As a first order approximation, let us assume that the cost of each one of these scalar multiplications is equivalent to one addition. Moreover, at the end of a row-vector multiplication, a Barrett reduction using Algorithm 1 is performed with an associated cost of at most two multiplications.

Then, the cost of a matrix-vector multiplication is of, $(\lambda - 1) \cdot N \approx 2^{28.11}$ and $2N \approx 2^{19.02}$ additions and multiplications, respectively. Since the cost of the sequential version of Wiedemann algorithm is dominated by the computation of about $3N$ matrix-vector multiplications, the estimated cost of finding the kernel of the single Quadratics matrix is of about $2^{47.72}$ and $2^{37.63}$, additions and multiplications, respectively.

The Quadratics matrix was solved using our C implementation of Wiedemann's algorithm; the computation took $4,320$ CPU hours in the server Pakal. Using the twenty cores available in this server, the elapsed time for this computation was of 216 hours. This matrix was also implemented in a GPU GeForce GTX Kepler architecture TITAN card running at 876MHz. Using $3,584$ GPU threads, the linear algebra computation took 223.2 hours.

4.3.2 Finding the Kernels of the Cubics Matrices

Since except for the first entry, the per-row non-zero values of the Cubics matrices are

all equal to one, the cost of a row-vector multiplication is of $\lambda - 1$ additions plus five field multiplications. Hence, the cost of a matrix-vector multiplication can be estimated in about $(\lambda - 1) \cdot N \approx 2^{25.36}$ and $5N \approx 2^{19.75}$ additions and multiplications, respectively. Since the cost of the sequential version of the Wiedemann algorithm is dominated by the computation of about $3N$ matrix-vector multiplications, the estimated cost of finding the kernel of all the 728 Cubics matrices is of about, $2^{53.89}$ and $2^{48.28}$, additions and multiplications, respectively.

The 728 Cubics matrices were solved using our C implementation of Wiedemann's algorithm. Each linear system was solved in parallel on 7 ABACUS cores. The 728 linear systems were solved simultaneously using 5096 ABACUS cores. The total execution time was $379,142$ CPU hours.

This time, and also the time for the linear algebra for the quartics (see §4.3.3), was more than expected in part because ABACUS was still running in an experimental phase and the machine was under-clocked to prevent over-heating. The increased CPU time did not have a significant impact on the total calendar time because of the large number of cores that we were at our disposal. The 728 solution vectors found were stored in files whose total size is of 26.4 gigabytes.

Using seven cores per matrix, the elapsed time for the linear algebra computation of each one of the Cubics matrices was of about 74.4 hours. Significantly, the Titan implementation of this problem took only 24.3 hours by using $3,584$ GPU threads.

4.3.3 Finding the Kernels of the Quartics Matrices

Since except for the first entry, the per-row non-zero values of the Quartics matrices are all equal to one, the cost of a row-vector multiplication is of $\lambda - 1$ additions plus five field multiplications.

Hence, the cost of a matrix-vector multiplication can be estimated in about $(\lambda - 1) \cdot N \approx 2^{22.54}$ and $5N \approx 2^{18.06}$ additions and multiplications, respectively. Since the cost of the sequential version of Wiedemann algorithm is dominated by the computation of about $3N$ matrix-vector multiplications, the estimated cost of finding the kernel of all the $21,141$ Quartics matrices is of about, $2^{54.24}$ and $2^{49.75}$, additions and multiplications, respectively.

The $21,141$ Quartics matrices were solved using our C implementation of Wiedemann's algorithm in $829,573$ CPU hours on ABACUS. Each linear system was solved in parallel on 2 cores. We used approximately 5000 cores to solve all 21,141 linear systems. The solution vectors were stored in files whose total size is 618 gigabytes.

Using two cores per matrix, the elapsed time for the linear algebra computation of each one of the Quartics matrices was of about 19.62 hours. Significantly, the Titan implementation of this problem took only 0.96 hours by using $3,584$ GPU threads.

4.4 Comparison

Table 3 reports a comparison of the elapsed time required by our linear algebra solver CPU and GPU implementations. It is interesting to note that a GPU solution is remarkably faster than a CPU implementation for the case of the Cubics and Quartics families. However, the CPU implementation outperform its GPU counter part for the computation of the quadratics matrix.

We believe that this behavior is due to the large size of this matrix that causes a costly divergence among the thread computations.

Due to the unique characteristics of the matrices derived in [2], it is in general difficult to compare our implementation with other related works. However, for the sake of completeness, we report in Table 4 a comparison of our GPU implementation against the one reported in [14]. The state-of-the-art software for computing Wiedemann algorithm is the CADO-NFS C implementation of the Number Field Sieve (NFS) algorithm for integer factorization and for computing discrete logarithms over finite fields [25]. However, we stress that the software in [25] specializes in the computation of usually one huge sparse matrix, but not in the simultaneous computation of thousands of large sparse matrices as is the case addressed in this work. Moreover, the GPU-based implementation of the CADO-NFS software basically corresponds

to the work reported in [14], which is included in Table 4.

5 Conclusion

In this work, we reported the successful implementation of the null-spaces associated with $21,870$ large sparse linear algebra systems.

The CPU years associated with this task is summarized in Table 5, where the CPU frequency column lists the average clock speed of the cores used. In order to accomplish the solution of this task in a reasonable calendar time, we use several parallel and supercomputing techniques both, in CPU and GPU platforms.

Acknowledgments

We would like to thank the ABACUS-CINVESTAV staff especially Isidoro Gitler, Daniel Ortiz-Gutiérrez and Omar Nieto-Ayalo, and the system manager Santiago Domínguez-Domínguez (CINVESTAV), for granting us computer access and for their technical support.

This project has received funding from Instituto Politécnico Nacional through projects SIP20211168 and SIP20211676, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 804476).

References

1. **Adj, G. (2016).** Discrete logarithms in small characteristic finite fields: attacking Type 1 pairing-based cryptography. Ph.D. thesis, Departamento de Computación, CINVESTAV, Mexico.
2. **Adj, G., Canales-Martínez, I., Cruz-Cortés, N., Menezes, A., Oliveira, T., Rivera-Zamarripa, L., Rodríguez-Henríquez, F. (2016).** Computing discrete logarithms in cryptographically-interesting characteristic-three finite fields. Cryptology ePrint Archive, Report 2016/914.
3. **Adj, G., Menezes, A., Oliveira, T., Rodríguez-Henríquez, F. (2014).** Computing discrete logarithms in $\mathbb{F}_{3^{6\cdot 137}}$ and $\mathbb{F}_{3^{6\cdot 163}}$ using magma. **Koç, Ç. K., Mesnager, S., Savas, E.**, editors, Arithmetic of Finite Fields - 5th International Workshop, WAIFI 2014, volume 9061 of Lecture Notes in Computer Science, Springer, pp. 3–22.
4. **Adj, G., Menezes, A., Oliveira, T., Rodríguez-Henríquez, F. (2014).** Weakness of $\mathbb{F}_{3^{6\cdot 509}}$ for discrete logarithm cryptography. **Cao, Z., Zhang, F.**, editors, Pairing-Based Cryptography - Pairing 2013 - 6th International Conference, volume 8365 of Lecture Notes in Computer Science, Springer, pp. 20–44.
5. **Barrett, P. (1987).** Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. Proceedings on Advances in cryptology—CRYPTO '86, pp. 311–323.
6. **Berlekamp, E. R. (1968).** Algebraic Coding Theory. McGraw-Hill, New York.
7. **Cinvestav (2017).** ABACUS supercomputer – Cinvestav.
8. **Coppersmith, D. (1994).** Solving homogeneous linear equations over GF[2] via block Wiedemann algorithm. Mathematics of Computation, Vol. 62, No. 205, pp. 337–350.
9. **Geiselmann, W., Shamir, A., Steinwandt, R., Tromer, E. (2005).** Scalable hardware for sparse systems of linear equations, with applications to integer factorization. **Rao, J. R., Sunar, B.**, editors, Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, volume 3659 of Lecture Notes in Computer Science, Springer, pp. 131–146.
10. **Granger, R., Kleinjung, T., Zumbrägel, J. (2014).** Breaking '128-bit secure' supersingular binary curves - (or how to solve discrete logarithms in $\mathbb{F}_{2^{4\cdot 1223}}$ and $\mathbb{F}_{2^{12\cdot 367}}$. **Garay, J. A., Gennaro, R.**, editors, Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Part II, volume 8617 of Lecture Notes in Computer Science, Springer, pp. 126–145.
11. **Hankerson, D., Menezes, A., Vanstone, S. (2003).** Guide to Elliptic Curve Cryptography. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
12. **Hoffstein, J., Pipher, J., Silverman, J. (2008).** An Introduction to Mathematical Cryptography. Springer Publishing Company, Incorporated, 1 edition.

13. **Jeljeli, H. (2014).** Accelerating iterative SpMV for the discrete logarithm problem using GPUs. **Koç, Ç. K., Mesnager, S., Savas, E.**, editors, Arithmetic of Finite Fields - 5th International Workshop, WAIFI 2014, volume 9061 of Lecture Notes in Computer Science, Springer, pp. 25–44.
14. **Jeljeli, H. (2014).** Resolution of linear algebra for the discrete logarithm problem using GPU and multi-core architectures. **Silva, F. M. A., de Castro Dutra, I., Costa, V. S.**, editors, Euro-Par 2014 Parallel Processing - 20th International Conference, volume 8632 of Lecture Notes in Computer Science, Springer, pp. 764–775.
15. **Jeljeli, H. (2015).** Accélérateurs logiciels et matériels pour l'algèbre linéaire creuse sur les corps finis. Ph.D. thesis, Doctorat de l'Université de Lorraine.
16. **Joux, A., Pierrot, C. (2014).** Improving the polynomial time precomputation of frobenius representation discrete logarithm algorithms - simplified setting for small characteristic finite fields. **Sarkar, P., Iwata, T.**, editors, Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Part I, volume 8873 of Lecture Notes in Computer Science, Springer, pp. 378–397.
17. **Joux, A., Pierrot, C. (2015).** Nearly sparse linear algebra and application to discrete logarithms computations. Cryptology ePrint Archive, Report 2015/930.
18. **Kaltofen, E. (1995).** Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. Mathematics of Computation, Vol. 64, No. 210, pp. 777–806.
19. **Kaltofen, E., Lobo, A. (1999).** Distributed matrix-free solution of large sparse linear systems over finite fields. Algorithmica, Vol. 24, No. 3–4, pp. 331–348.
20. **Kaltofen, E., Saunders, B. D. (1991).** On Wiedemann's method of solving sparse linear systems. **Mattson, H. F., Mora, T., Rao, T. R. N.**, editors, Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, volume 539 of Lecture Notes in Computer Science, Springer, pp. 29–38.
21. **Lanczos, C. (1950).** An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. J. Res. Nat'l Bur. Std., Vol. 45, pp. 255–282.
22. **Massey, J. (1969).** Shift-register synthesis and BCH decoding. IEEE Trans. Inf. Theor., Vol. 15, No. 1, pp. 122–127.
23. **OpenMP (2015).** The OpenMP API specification for parallel programming.
24. **Tan, X. (2017).** A new extrapolation method for PageRank computations. Journal of Computational and Applied Mathematics, Vol. 313, No. Supplement C, pp. 383 – 392. DOI: https://doi.org/10.1016/j.cam.2016.08.034.
25. **The CADO-NFS Development Team (2017).** CADO-NFS, an implementation of the number field sieve algorithm. Release 2.3.0.
26. **Tromer, E. (2007).** Hardware-Based Cryptanalysis. Ph.D. thesis, Weizmann Institute of Science.
27. **Villard, G. (1997).** Further analysis of coppersmith's block Wiedemann algorithm for the solution of sparse linear systems (extended abstract). **Char, B. W., Wang, P. S., Küchlin, W.**, editors, Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, ISSAC '97, ACM, pp. 32–39.
28. **Wiedemann, D. H. (1986).** Solving sparse linear equations over finite fields. IEEE Trans. Inf. Theor., Vol. 32, No. 1, pp. 54–62.
29. **Yang, L. T., Huang, Y., Feng, J., Pan, Q., Zhu, C. (2017).** An improved parallel block Lanczos algorithm over GF(2) for integer factorization. Inf. Sci., Vol. 379, pp. 257–273.

Article received on 24/09/2020; accepted on 16/09/2021.
Corresponding author is Nareli Cruz-Cortés.