A New Parallel Approach to the Block Lanczos Algorithm for Finding Nullspaces over GF(2)

Ildikó Flesch

Supervisor: Dr. Rob H. Bisseling

26th November 2002

Preface

This Master's thesis is on cryptography, the art of secret writing. It was written under the supervision of Dr. Rob H. Bisseling, at the Department of Mathematics, Utrecht University, the Netherlands. Its goal is to propose a new parallel algorithm to calculate the nullspace of a given huge matrix, in the field of GF(2). This parallel algorithm is mainly based on the Block Lanczos algorithm, but it presents a new way for distributing the matrix elements as well as minimizing the communication time. This thesis contains the necessary theoretical background and concludes with experimental results.

First of all, I wish to thank my supervisor, Rob Bisseling, for his guidance and inspiration. I would like to thank my brother, János Flesch, for the support he has given me during the last years.

Ildikó Flesch Utrecht, August 2002

Contents

1 Introduction

1.1 Cryptography and number theory

The birth of cryptography goes back to the beginning of literacy. It has been widely used in diplomacy, business correspondence, criminal activities, intelligence (quite often negative applications), but in our time of information technology, cryptography is experiencing a boost and it already plays a significant role in our everyday life (for example in communication over the Internet).

First of all, we would like to illustrate the main ideas by the following fictitious example about an astronomer in the Renaissance. It is well known that the catholic church did not want to accept other views on the structure of our universe than the official church one. Think for example of Galileo Galilei, who found out that Sun was the center of our solar system and not Earth as claimed by the church. One can imagine that the astronomer was often in danger and had to fear the inquisition. Since he wanted to exchange letters with his scientist friends, he had to find a method to write his messages in such a way that nobody else but his friends could read it. His idea was the following. With his friends, they agreed on shifting the letters in the alphabet by 3 places to the right (for example 'A' becomes 'D', 'K' becomes 'N' and 'Z' becomes 'C'; the word 'EARTH' becomes 'HDUWK'). Here, shifting the letters can be seen as the coding method, and the number of places to shift (in our example 3) is the secret key. In fact, this method was used by Julius Caesar as well. This is why it is also called Caesar method.

Suppose this astronomer wanted to send the word 'EARTH' in the message. Then, before sending his message (called **plain text**) with the word 'EARTH' in it, first he coded (i.e. **encrypted**) this text with the help of the secret key by applying this shifting method and afterwards he only sent the coded text (called **cypher text**). So, the letter contained the word 'HDUWK' instead of the original word 'EARTH'. Knowing the secret key, his friends only had to shift the letters by 3 places to the left (i.e. **decrypt** it), and thus they could reproduce the original message. By doing so, they obtained the word 'EARTH' from the code 'HDUWK'. On the other hand, nobody else was able to decode the coded text in the letter without having the secret key. In this way, the astronomer and his friends could exchange messages in a secret way.

In general, just as in the example above, one has to apply an encryption method for which it is possible to reproduce the original text from the code.

Note that the encryption method can be made public, but the key must be kept secret (this is called Kerckhoffs' Principle). The main motivation for this is the following: (1) the security of the system relies on as little information as possible (only the key), (2) the encryption method can be made a standard (for example a software package) and then it can be used by many people.

It is clear that it might be in some people's interest (like the inquisition in the above example) to find the secret key and read the plain text. Therefore it is necessary to develop encryption techniques which assure safety as much as possible. The field of cryptology is concerned with the ongoing struggle between cryptographers (developers of cryptographic systems) and cryptanalysts (breakers of cryptographic systems). Nowadays, the modern encryption techniques and methods for breaking the codes are based on manipulating numbers (for example $A = 1$, $B = 2$, ...), and therefore number theory has become an important basis for cryptography.

1.2 Symmetric and asymmetric cryptography

In cryptography there are two major classes of encryption systems, namely symmetric and asymmetric encryption.

The difference between symmetric and asymmetric cryptography lies in the way the keys are used: in the case of symmetric cryptography the sender and the receiver use the same key, while in asymmetric cryptography the keys are different.

Suppose that Alice wants to send a message to Bob and that they use a symmetric encryption system. In this case, Alice and Bob have the same secret key, with the only difference that Alice applies the key directly, while Bob decrypts with the inverse of this unique key. Of course, it is necessary that the inverse of the key exists, therefore operations like adding, subtracting, multiplying are applied.

For symmetric cryptography the shift system is a good example, as described in section 1.1. Recall that the unique key k is here equal to 3; Alice shifts the letters three places to the right and Bob shifts the received cyphertext three places in the other direction.

It is a disadvantage of symmetric cryptography that Alice and Bob use the same key, because it is difficult for Alice to send her key to Bob in a secure way. It is not a problem if they live next to each other. But suppose that Alice has a big company and Bob is a client of hers. Then Alice has to send the key by a trustable person. A bigger company has many clients, also abroad. Every client needs a personal key, so Alice has to have many agents

to distribute these keys. Therefore it is very difficult to keep all of them secret and Oscar, the dangerous competitor, has an easier task in bribing somebody to give him the key of Bob. It is also cheaper for Oscar, because he does not need any modern technology. Alice does not only need many trustable agents, but also a secure place to hide the keys of all the clients. Notice that Bob has to make sure that Oscar cannot break into his place and steal the key.

It was clear decades ago that the above problem with the keys would only become more serious as technology and Internet developed rapidly. Many experts tried to find a solution.

Whitfield Diffie, and Martin Hellman [8] were the first to publish their idea of asymmetric cryptography in 1975. As already mentioned, the sender and the receiver use different keys for encryption and decryption. Each participant in the communication has two kinds of keys: a public and a secret key. The public key can be known to everyone, in contrast with the secret key, which is personal. In our example when Alice wants to send a message to Bob, she encrypts the text with the help of Bob's public key (and by applying the given encryption system). When Bob receives the message he has to decrypt it with his own secret key. Without this personal secret key no one else is able to decrypt the message, because the public key is not invertible.

The advantage of an asymmetric cryptosystem is obvious. No distribution (no agent) is necessary for the secret key, because the sender does not need it, so the security of the system increases. Of course, everyone is still responsible for keeping his own personal key safe. Note also that if Alice wants to communicate with several people then she needs only one secret key, unlike in the case of symmetric systems. This is much safer, because she only has to be sure of the security of one single key.

In the literature, asymmetric cryptography is also called public key cryptography, because of the idea of the public key.

It was a fine idea, but mathematicians had to find a function with which asymmetric cryptography could be executed. The expert James Ellis of the British secret service in Cheltenham had already invented the idea of asymmetric cryptography in 1969 and Clifford Cocks the needed function in 1973 before Diffie, Hellman, and Merkle started their research, but they did not want to publish it for security reasons. It took mathematicians many years to find a suitable function and when found, they could publish it. Their idea has been used in different algorithms, but RSA is the most widely used public key cryptographic technique today.

1.3 The RSA algorithm

The RSA algorithm is called after the first letters of the names of its developers Rivest, Shamir, and Adleman and was invented in 1977 [21]. It has become one of the most widely used public key systems. The basic idea of RSA is that it is very difficult to find two large primes p and q given their product $N = pq$, because there exists no efficient technique to find these primes. On the other hand it is very easy to find two large prime numbers and compute their product.

Definition 1 ***Factorization****: Given a composite number N, find its divisors that are not equal to 1 or to N.*

RSA works in a really smart way. The algorithm first chooses arbitrary primes p and q and then can compute N easily. Afterwards it encrypts the plaintext in a certain manner with the values of N, p, and q, which will be explained in subsection 1.3.1. The dangerous spy Oscar, who wants to break the code, knows the value N of the encryption (since it belongs to the public key, which is known to everyone) but not p and q. Therefore he needs to find the two unique prime divisors p and q in order to obtain the decryption key and thus break the code, but this is easier said than done.

1.3.1 Description of the RSA algorithm

To understand the working of RSA, we need some preliminaries from number theory.

Definition 2 *Let m and n be two natural numbers. Then, m* ***modulo*** *n, denoted by m mod n, is equal to the remainder after the division of m by n.*

For example $5 \bmod 2 = 1$ and $10 \bmod 7 = 3$. Modulo n, as an operation, is a fine candidate for asymmetric cryptography.

Definition 3 *Let x, y, and n be natural numbers. Then x and y are called* ***congruent modulo*** *n, if $x - y$ is divisible by n. This is denoted by $x \equiv y \ (mod\ n)$.*

For example $23 \equiv 10 \pmod{13}$ and $15 \equiv 7 \pmod{8}$. Notice that x and y are congruent modulo n exactly when $x \bmod n$ and $y \bmod n$ are equal. There are infinitely many congruent pairs modulo a given value of n. The relation $\equiv$ modulo n is an equivalence relation (i.e., it is reflexive,

symmetric, and transitive), and therefore induces equivalence classes in the usual manner.

For a given n the number of equivalence classes is exactly n, but all of them are infinite. We may represent each class by an element. In this way, the set of equivalence classes is simply $\mathbf{Z}_n = \{0, 1, \ldots, n-1\}$. Addition, subtraction, and multiplication of the equivalence classes is defined with the help of these representative elements. For example, adding the classes a and b results in the class $(a + b)$ mod n. With these operations the set $\mathbf{Z}_n$ becomes a commutative ring.

For example take $n = 9$. Then $\mathbf{Z}_9 = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ and we have $2 + 3 = 5 \bmod 9 = 5$ and $5 + 5 = 10 \bmod 9 = 1$, whereas $5 \cdot 5 = 25 \bmod 9 = 7$.

Definition 4 *Let n be a natural number. Then the multiplicative ring of $\mathbf{Z}_n$, denoted by $\mathbf{Z}_n^*$ is the set of elements $a \in \mathbf{Z}_n$ for which there is an element $b \in \mathbf{Z}_n$ such that $(a \cdot b)$ mod $n = 1$.*

It is clear that 0 is always in the commutative ring $\mathbf{Z}_n$, but 0 is never a member of $\mathbf{Z}_n^*$, because it has no inverse ($(0 \cdot b)$ mod n equals 0 for all $b \in \mathbf{Z}^n$). On the other hand, if $n \geq 2$ then 1 belongs to $\mathbf{Z}_n^*$, because $(1 \cdot 1)$ mod $n = 1$.

Let us see, for example, how to compute the multiplicative ring $\mathbf{Z}_9^*$ from the elements of $\mathbf{Z}_9$. We check for every element of $\mathbf{Z}_9$ whether it satisfies the condition in definition 4. We already know that $0 \notin \mathbf{Z}_9^*$ and $1 \in \mathbf{Z}_9^*$. For the number $a = 2$, we may take $b = 5$, so that $2 \cdot 5 = 10 \bmod 9 = 1$. Thus 2 will be in $\mathbf{Z}_9^*$ (and 5 as well). Now we proceed with 3. We can easily see that, since $n = 9$ is a multiple of 3, for every b we have $3 \cdot b \bmod 9 \in \{0, 3, 6\}$. Hence, $0, 3, 6 \notin \mathbf{Z}_9^*$. With further computation we find $4 \cdot 7 = 28 \equiv 1$ and $8 \cdot 8 = 64 \equiv 1 \pmod 9$. So the result is $\mathbf{Z}_9^* = \{1, 2, 4, 5, 7, 8\}$.

Definition 5 *The number of elements of $\mathbf{Z}_n^*$ is denoted by $\phi(n)$, and ϕ is called the **Euler function**.*

The next lemma tells us how to calculate the Euler function.

Lemma 1 *The multiplicative ring $\mathbf{Z}_n^*$, where n is a prime number, is equal to $\{1, 2, \ldots, n-1\}$ and therefore $\phi(n) = n-1$. On the other hand, if $n = pq$, where p and q are distinct odd primes then $\phi(n) = \phi(p) \cdot \phi(q) = (p-1)(q-1)$.*

The proof is based on the following observation. Denote the greatest common divisor of two numbers by gcd. Let n, a be natural numbers such

that $a < n$ and $\gcd(a, n) = 1$. By using the Euclides algorithm we can obtain two integers x, y such that $1 = ax + ny$. Take $\overline{x} \in \mathbf{Z}_n$ such that $\overline{x} \equiv x \mod n$. Then

$$a \cdot \overline{x} \bmod n = a \cdot x \bmod n = (ax + ny) \bmod n = 1 \quad . \tag{1}$$

Suppose that n is prime and let $a \in \{1, 2, \ldots, n-1\}$. Then we have $\gcd(a, n) = 1$, and by the previous observation $a \in \mathbf{Z_n}^*$. This proves the first part of the lemma. The second part follows from the observation that $\mathbf{Z}_n^* = \{1, 2, \ldots, n\} - \{p, 2p, \ldots, qp\} - \{q, 2q, \ldots, pq\}$.

For example, by using this lemma, we know without computation that $\mathbf{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$ and $\phi(7) = 6$. Also for $n = 15$ we have $\phi(15) = \phi(3) \cdot \phi(5) = 2 \cdot 4 = 8$.

The following theorem states an important property of the Euler function.

Theorem 1 *Lagrange theorem: Let $b \in \mathbf{Z}_n$. Then $b^{\phi(n)} = 1$ mod n.*

For the proof we refer to [24]. The next theorem follows from lemma 1 and theorem 1.

Corollary 1 *Fermat's Little Theorem: Let $b \in \mathbf{Z}_p$ and let p be a prime number. Then $b^{p-1} = 1$ mod p.*

To prove this corollary is an easy job. If p is a prime number then $\phi(p) = p - 1$. Substituting this in the Lagrange theorem above, we get the equality of Fermat. Now we are ready to discuss the working of RSA. The first step is to choose two prime numbers p and q, and to calculate $N = p \cdot q$. This means that $\mathbf{Z}_N$ is a product ring of $\mathbf{Z}_N = \mathbf{Z}_p \times \mathbf{Z}_q$. Because of the prime property of p and q

$$\phi(N) = \phi(p) \cdot \phi(q) = (p-1)(q-1). \tag{2}$$

Given $\phi(N)$, we choose an arbitrary element $e \in \mathbf{Z}_{\phi(N)}^*$. Since e is in the multiplicative ring, there must exist an inverse $d = e^{-1} \in \mathbf{Z}_{\phi(N)}^*$. (This inverse can be calculated efficiently with the algorithm of Euclides. For more details of the Euclides algorithm see [2]). As always in asymmetric cryptography, we have two keys: a public key and a secret key. Here in RSA, the public key is (N, e), while the secret key is d. After the computation of

the secret key d, we do not need p and q anymore. For safety reasons we discard them.

If Alice wants to send a plaintext x to Bob, she encrypts x with $y = x^e$. Bob receives from Alice the cyphertext, and decrypts with his secret key d as follows: y^d. Bob obtains the original plaintext, because by the definition of d there is a non-negative integer k such that $ed = k\phi(N) + 1$, and then by the Lagrange Theorem $y^d = (x^e)^d = x^{ed} = x^{k\phi(N)+1} = x^1 = x (\text{mod } N)$.

The question is now, why Oscar cannot read the message x easily. Oscar knows that Alice applied RSA for encryption; he also knows the public key (N, e) and has the cyphertext. His "only" work is to find the secret key d. To find d, he needs to know $\phi(N)$, the Euler function of N. However, $\phi(N)$ cannot be found efficiently without knowing the two primes p and q. Therefore, Oscar has to find p and q or in other words, he has to factor the composite number N, which cannot be done with an efficient algorithm.

1.3.2 Historical development

After the introduction of RSA, cryptanalysts started to develop techniques to attack this algorithm. There are three possibilities to attack RSA:

- **Brute force attack**, where all possible private keys are tried.
- **Timing attacks**, which depend on the decryption algorithm.
- **Mathematical attacks** by factorization algorithms.

The most primitive attack is the brute force attack, which because of the large size of the RSA numbers, has a huge key space. This means that the calculation takes too much time and therefore cannot be used for cracking RSA.

Paul Kocher (see [23]), a cryptographic consultant, has demonstrated that a spy can determine the private key if he knows how much time the computer used for the decryption of the data. This idea is really dangerous, because one does not expect this kind of attack and besides the snooper only needs the cyphertext (therefore this attack is called a **cypertext-only attack**). Fortunately, without a spy and inside information the idea is not applicable.

In this subsection we turn attention to the development of the factorization methods. The challenge for factorization started when the three inventors in 1977, offered $100 to the person or group who would decrypt their encrypted message published in Scientific American. Their expectation was that nobody would be able to decipher it for some 40 quadrillion years.

decimal digits	year of the factoring	algorithm	MIPS years
RSA-100	1991	QS	7
RSA-110	1992	QS	75
RSA-120	1993	QS	835
RSA-129	1994	QS	5000
RSA-130	1996	NFS	1000
RSA-140	1999	NFS	2000
RSA-155	1999	NFS	8400
RSA-158	2002	NFS	

Table 1: Factoring records

Bob Silverman in 1994 showed how naive this expectation was, by using distributed computing to increase the power of the factorization algorithms. Only 1000 computers were used, and it took him only 8 months time to get the award. The challenge in the Scientific American used a public key size of 129 decimal digits which is about 428 bits. The 1000 computers used 0.03 percent of the maximal computing power of the Internet. Since institutions at that time existed that already possessed more computers, it is clear that this size of public key was not secure anymore. In Table 1, the broken RSA-challenges are listed (RSA-n abbreviates RSA with a public key of n decimal digits). This table also shows how many MIPS years were needed, where one MIPS year is equal to a million-instructions-per-second processor running for one year. (The most recent challenge can be found at the RSA Challenge List [19].)

For the last three factorizations NFS was applied which is the most successful method at this time. It is also expected to break the RSA key with 768-bit integer before 2004, and the 1024-bit integer before 2014. New factorization records are the result of increasing computing power and improvements in algorithms.

1.4 Factorization algorithms

There exist many ways to find the prime factors of a given number N, though no sufficiently efficient algorithms are known. The most primitive one is checking all the primes till the value $\sqrt{N}$. If the algorithm cannot find a divisor of N, then the given number must be prime. The disadvantage of this brute-force algorithm is its exponential running time.

Take an example for $N = 10^{40}$ from [2]. The total number of modulo

operations for this N is approximately 10^{20} and each computational step of the brute-force algorithm with a PC costs about 10^{-7} second computation time. This means that we would need about $30,000$ calendar years to factor 10^{40}.

Therefore cryptanalysts tried to develop factorization techniques which are much quicker than the above one. We divide the factorization techniques into two classes (see [15]). The first class consists of those techniques which find small prime factors quickly, such as trial division, Pollard Rho, $P \pm 1$, and the elliptic curve method. On the other hand, the second class is made up of factorization techniques that are developed to find arbitrarily large prime factors, like continued fraction, quadratic sieve, and number field sieve methods. Of course, the algorithms in the second class are much slower. They are useful if no small prime factors exists, which is the case for RSA with p, q large primes.

1.4.1 Factorization algorithms for large numbers

In this subsection, we describe the basic idea of factorization algorithms belonging to the second class.

The goal of the factorization algorithms for large numbers is to find as many pairs (X, Y) as possible that satisfy:

$$X^2 \equiv Y^2 \pmod{N} \quad \text{and } \gcd(XY, N) = 1 \quad , \tag{3}$$

because $\gcd(X - Y, N)$ is a factor of N for every such pair (X, Y). For more details we refer to [15]. Unfortunately, it is not sure that we obtain a factor different from 1 and N. It has been shown in [15] that this is the case only in 50 percent of our attempts.

Definition 6 *A number N is called* **smooth** *with respect to a bound β, if its prime factors are under the bound β.*

A well-known table, which presents the prime factors of composite numbers of the form $b^n \pm 1$ with small n is the Cunningham Table. This is extended now till $b \leq 99$.

In order to find numbers X and Y, we first try to find in some way (determined by the method) many pairs (a, b) satisfying

$$a \equiv b \ (\bmod N) \quad , \tag{4}$$

	a	b
1.	$25 \equiv$	-8
2.	$32 \equiv$	-1
3.	$1 \equiv$	-32
4.	$28 \equiv$	-5
5.	$40 \equiv$	7
6.	$35 \equiv$	2
7.	$2560 \equiv$	-14
8.	$128 \equiv$	-4
9.	$125 \equiv$	-7
10.	$343 \equiv$	-20

Table 2: Set S of possible pairs for (a, b) for $N = 33$, with $\beta = 9$.

where a and b are either squares or a square times a smooth number.

Let S denote the set of all such ordered pairs (a, b), numbered (a_i, b_i), $1 \leq i \leq k$. Next we try to find non-empty subsets S' of S such that

$$\prod_{(a,b)\in S'} a \text{ and } \prod_{(a,b)\in S'} b \quad \text{are square.} \tag{5}$$

Because of the choice of S', the product of the a's and the b's can be written in quadratic form as in (3). The only problem that remains is how to find a subset S'. During the computation of S' we will use the fact that every number has a unique prime factorization.

Let us take the example of the small composite number $N = 33$. We would like to factor 33 and find its two prime factors 3 and 11. As above, first we seek pairs which satisfy (4); such pairs are given in Table 2. We do not use the congruent pair $(49, 16)$, because both sides are already squares.

From the pairs in Table 2, we now construct another table, Table 3, that contains the prime factorizations of the a and b.

From Table 2, take for example the 4th and the 10th pairs. Then the products are both squares as the following calculation shows:

$$\begin{aligned} a_4 \cdot a_{10} &= (2^2 \cdot 7)(7^3) \\ &= 2^2 \cdot 7^4 \\ &= (2 \cdot 7^2)^2 \end{aligned}$$

a	25	32	1	28	40	35	2560	128	125	343
$p=2$	0	5	0	2	3	0	9	7	0	0
$p=5$	2	0	0	0	1	1	1	0	3	0
$p=7$	0	0	0	1	0	1	0	0	0	3
b	-8	-1	-32	-5	7	2	-14	-4	-7	-20
$p=-1$	1	1	1	1	0	0	1	1	1	1
$p=2$	3	0	5	0	0	1	1	2	0	2
$p=5$	0	0	0	1	0	0	0	0	0	1
$p=7$	0	0	0	0	1	0	1	0	1	0

Table 3: The prime factorization of the congruent pairs of Table 2. The entries in the table represent the exponents of the corresponding prime factor.

$$\begin{aligned} &= (98)^2 \\ \text{and} & \\ b_4 \cdot b_{10} &= (-1\cdot 5)(-1\cdot 2^2\cdot 5) \\ &= (-1)^2\cdot 2^2\cdot 5^2 \\ &= (2\cdot 5)^2 \\ &= (10)^2 \quad . \end{aligned} \tag{6}$$

From Table 3, we may find the subsets S' as follows. Since we have to choose elements in such a way that their product is a square, we have to make sure that the exponent of each prime is even, in a and b together. Therefore the simplified Table 4 is sufficient for our algorithm.

Let B denote the matrix of size $n_1 \times n_2$, where the elements are equal to the exponents modulo 2 of the prime factors, as given in Table 4.

$$B = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{7}$$

Observe that finding pairs (X, Y) is the same as choosing a set of columns from matrix B. In order to choose columns of matrix B we multiply B by

a	25	32	1	28	40	35	2560	128	125	343
$p=2$	0	1	0	0	1	0	1	1	0	0
$p=5$	0	0	0	0	1	1	1	0	1	0
$p=7$	0	0	0	1	0	1	0	0	0	1
b	-8	-1	-32	-5	7	2	-14	-4	-7	-20
$p=-1$	1	1	1	1	0	0	1	1	1	1
$p=2$	1	0	1	0	0	1	1	0	0	0
$p=5$	0	0	0	1	0	0	0	0	0	1
$p=7$	0	0	0	0	1	0	1	0	1	0

Table 4: The prime factorization of the congruent pairs in Table 2 with the exponents taken modulo 2.

a vector x of length n_2 whose ith component equals 1 if we choose column i, and 0 otherwise.

Because we only need to know if the exponents of the primes are even or odd, and each element of x is either 0 or 1, we are able to compute everything modulo 2. Clearly, the vector x gives us a linear combination of the column vectors of B. Since we want the exponents to be even, we would like to find an x for which $Bx \equiv 0 \bmod 2$. In other words, we want to find the nullspace of the matrix B in GF(2).

In our example for the pairs (a_4, b_4) and (a_{10}, b_{10}) we may take the vector $x = [0,0,0,1,0,0,0,0,0,1]^T$, which means choosing columns 4 and 10 of B.

From (6) we know that $X = 98$ and $Y = 10$.

Note that $\gcd(XY, N) = \gcd(980, 33) = 1$. The multiplication of the two columns of B is successful because $\gcd(X - Y, N) = \gcd(88, 33) = 11$, which is a non-trivial factor of the composite number 33.

It is also possible, especially for large numbers, that there are more solutions for the vector x. For example let $x = [1,1,1,0,0,0,0,1,0,0]$, which corresponds in Table 4 to the 1st, 2nd, 3rd and 8th columns. It satisfies also $\gcd(XY, N) = \gcd(10240, 33) = 1$, and $\gcd(X-Y, N) = \gcd(320-32, 33) = \gcd\ (288, 33) = 3$, resulting in factor 3.

It can also happen that we obtain a trivial factor. We would like to emphasize that, unfortunately, the factorization techniques for large numbers are not certain to find a non-trivial factor, but usually they work pretty well. We wish to mention that the Number Field Sieve (NFS) is one of the most commonly used methods.

1.5 Finding the nullspace of a matrix

Recall that in the third step of the NFS algorithm, the linear equation $Bx = 0$ has to be solved, where B is of size $n_1 \times n_2$. Here n_1 is equal to the number of prime factors in a and b together; n_2 gives us the total number of pairs found during the sieving phase. Because there are many pairs, the matrix B has many columns, and has a huge size.

On the other hand we usually have $n_1 < n_2$ according to [17], because each prime can be a prime factor of several values of a and b. The matrix B is a very sparse matrix, i.e., many of its elements are equal to 0, which is due to the fact that most primes are not divisors of a pair (a, b).

In linear algebra the direct method to solve $Bx = 0$ is Gaussian elimination, which has a running time $O(n^3)$ for an $n \times n$ system. It is efficient for small matrices, but here it would be too slow because of the size of B. Therefore we rather apply iterative methods. As B is sparse, it is advisable to use the Conjugate Gradient (CG) or the Lanczos iterative methods, which solve the linear equation $Ax = b$, where the matrix A is symmetric and positive definite.

Because we need symmetry, we have to start by transforming B into $A = B^T B$, and use this new matrix A in CG or in Lanczos (and set the right-hand side $b = 0$). It is a lot of work to compute the matrix A explicitly and it also takes much storage, but fortunately we only need A implicitly, as we will see in Chapter 2.

CG and Lanczos both need one matrix-vector multiplication per iteration and three vector inner products. The only difference is that CG uses two subtractions of vectors, while Lanczos calculates one subtraction and one addition of two vectors. Thus the costs are the same in terms of high-level operations. For more details we refer to [16].

In the relevant literature, both CG and the Lanczos algorithms have been applied. According to LaMacchia and Odlyzko [12], there is no significant difference in computation time. Finally, we wish to mention that during the breaking of RSA-130, RSA-140, RSA-150, and RSA-158 with NFS, the Block Lanczos algorithm was applied, which is a modification of Lanczos (see Chapter 2).

2 Sequential algorithm for computing null spaces in $GF(2)$

2.1 The sequential Lanczos algorithm

In 1950, Lanczos proposed an algorithm [13], which produces an orthogonal transformation of a symmetric matrix A into a tridiagonal matrix T. The point of the algorithm is that, due to the orthogonality of the transformation, matrix T is similar to the original matrix A (for instance, each eigenvalue of T is also an eigenvalue of A), but it is much easier to calculate with T because it has a tridiagonal form. This idea of Lanczos is frequently applied to solve eigenvalue problems and linear systems.

Let A be a symmetric matrix of size $n \times n$. The goal of the algorithm is thus to find an orthogonal matrix Q of size $n \times m$ ($m \leq \text{rank}(A)$) such that

$$T = Q^T A Q \tag{8}$$

has a tridiagonal form

$$T = \begin{bmatrix} \alpha_1 & \beta_1 & \dots & \dots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_{m-1} \\ 0 & \dots & & \beta_{m-1} & \alpha_m \end{bmatrix}. \tag{9}$$

Such a matrix Q can be found as follows. Let q_1 be an arbitrary unit vector in $\mathbf{R}^n$. The kth Krylov vector is defined as $v_k = A^{k-1} q_1$, and the kth Krylov subspace as the subspace spanned by the first k Krylov vectors:

$$K_k := \text{ span } (q_1, Aq_1, \dots, A^{k-1} q_1) \ , \tag{10}$$

where the span of the sequence of vectors $q_1, Aq_1, \dots, A^{k-1}q_1$ denotes the space of all vectors that can be written as a linear combination of the vectors q_1, $Aq_1, \dots, A^{k-1}q_1$. Clearly, there is an $m \in \mathbf{N}$ such that the vectors $v_1, v_2, \dots, v_m$ are all independent, but all the further Krylov vectors are dependent on them. Let $q_1, q_2, \dots, q_m$ denote the orthonormalization (for example by the Gram-Schmidt algorithm) of the first m Krylov vectors.

Suppose we have m independent Krylov vectors. Obviously $\{q_1, q_2, \ldots, q_m\}$ gives an orthonormal basis for the Krylov subspace K_m. Let $Q = [q_1, q_2, \ldots, q_m]$ and observe that Q is an orthogonal matrix of size $n \times m$.

We wish to check that $T = Q^T AQ$ has the above tridiagonal structure. Because A is symmetric, so is T (this is why it is necessary to start with a symmetric matrix). It only has to be shown that element $T_{ij} = 0$, if $i \geq j+2$. We have $T_{ij} = q_i^T A q_j$. By construction, $q_{j+2}, \ldots, q_m$ are all orthogonal to the Krylov subspace K_{j+1}. Since $Aq_j \in K_{j+1}$, we must have $T_{ij} = 0$.

The diagonal element of T is $\alpha_k = q_k^T A q_k$ and the sub diagonal element $\beta_k = q_{k+1}^T A q_k$. By construction, we have $\text{span}(q_1, \ldots, q_k) = \text{span}(q_1, Aq_1, \ldots, A^{k-1}q_1)$. Therefore

$$\begin{aligned} Aq_k \in \text{span}(Aq_1, A^2 q_1, \ldots, A^k q_1) &\subset \text{span}(q_1, Aq_1, \ldots, A^k q_1) \\ &= \text{span}(q_1, \ldots, q_{k+1}) \quad . \end{aligned} \tag{11}$$

So, we can find coefficients λ_{jk}, $0 \leq j \leq k+1$, such that

$$Aq_k = \sum_{j=0}^{k+1} \lambda_{jk} q_j \quad . \tag{12}$$

Now for each i with $0 \leq i \leq k+1$, we obtain

$$T_{ik} = q_i^T A q_k = \sum_{j=0}^{k+1} \lambda_{jk} q_i^T q_j = \lambda_{ik} \quad . \tag{13}$$

This implies

$$Aq_k = \sum_{j=0}^{k+1} T_{jk} q_j = \beta_{k-1} q_{k-1} + \alpha_k q_k + \beta_k q_{k+1}. \tag{14}$$

With the help of equation (14) we can now compute q_{k+1} recursively as follows:

$$q_{k+1} = \frac{1}{\beta_k}((A - \alpha_k I)q_k - \beta_{k-1} q_{k-1}) \quad . \tag{15}$$

The vector q_k is also called the kth **Lanczos vector**.

Now, the Lanczos algorithm is as follows:

Input: A
Output: T, Q such that $T = Q^T AQ$
$r_0 = q_1; \beta_0 = 1; q_0 = 0; k = 0;$
while $(\beta_k \neq 0)$
$q_{k+1} = r_k/\beta_k;$
$k = k + 1;$
$\alpha_k = q_k^T A q_k$
$r_k = (A - \alpha_k I)q_k - \beta_{k-1} q_{k-1}; \beta_k = \|r_k\|_2$
end (16)

The scalar β_k is chosen to be positive, which can be done without loss of generality. The advantages of this algorithm are the minimal storage requirement (only the matrix A and q_{k-1}, q, q_{k+1}) and the small number of arithmetic operations.

Finally, we wish to mention that if matrix A has a huge size then in many applications it is often sufficient to use an orthogonal transformation as above based on the first k Krylov vectors ($k < m$) if k is large enough.

2.1.1 Solving linear systems with Lanczos

The Lanczos iteration to solve $Ax = b$ with A symmetric positive definite matrix can simply be written down in the following general form

$$\begin{aligned} w_0 &= b, \\ w_i &= Aw_{i-1} - \sum_{j=0}^{i-1} c_{ij} w_j \quad (i > 0), \quad \text{where } c_{ij} = \frac{w_j^T A^2 w_{i-1}}{w_j^T A w_j} \quad . \end{aligned} \tag{17}$$

Notice that c_{ij} is well defined because A is positive definite. The iteration will break down at some step m when $w_m = 0$. Then, the exact solution x can be defined as

$$x = \sum_{j=0}^{m-1} \frac{w_j^T b}{w_j^T A w_j} w_j \quad , \tag{18}$$

which we show later.

First, we remark that the vectors $w_0, w_1, \ldots, w_{m-1}$ are **A-orthogonal**: $w_i^T A w_j = 0$ for all $i \neq j$ (here the positive definiteness of A is crucial). One can check this by induction on $\max(i,j)$. For $\max(i,j) = 0$ the statement is trivial. Suppose that the statement holds if $\max(i,j) \leq z-1$. Now we have to show that the statement is also valid for $\max(i,j) = z$. Assume without loss of generality that $i = z$ and $j < i$. Then from equation (17) we have

$$\begin{aligned} w_i^T A w_j &= (A w_{i-1} - \sum_{k=0}^{i-1} c_{ik} w_k)^T A w_j \\ &= w_{i-1}^T A^T A w_j - \sum_{k=0}^{i-1} c_{ik} w_k^T A w_j \quad . \end{aligned} \tag{19}$$

By the induction assumption it holds that $w_k^T A w_j = 0$ for all $k \in \{0, 1, \ldots, i-1\} - \{j\}$, so the definition of c_{ij} and the symmetry of A yields:

$$w_i^T A w_j = w_{i-1}^T A^T A w_j - c_{ij} w_j^T A w_j = 0 \quad . \tag{20}$$

From the A-orthogonality and positive definiteness of A it follows that $w_0, \ldots, w_{m-1}$ are linearly independent. Therefore the iteration above stops in at most n steps (so $m \leq n$).

Moreover, if $i > j+2$ then by construction and the symmetry of A

$$\begin{aligned} c_{ij}(w_j^T A w_j) &= w_j^T A^2 w_{i-1} \\ &= w_j^T (A^T A) w_{i-1} \\ &= (A w_j)^T A w_{i-1} \\ &= (w_{j+1} + \sum_{k=0}^{j} c_{j+1,k} w_k)^T A w_{i-1} \\ &= 0 \quad , \end{aligned} \tag{21}$$

thus $c_{ij} = 0$. Therefore the definition of w_i, for $i \geq 2$, simplifies to the three-term recurrence

$$w_i = A w_{i-1} - c_{i,i-1} w_{i-1} - c_{i,i-2} w_{i-2} \quad , \tag{22}$$

which requires much less computational work.

We wish to verify now that the solution for x in (18) is correct indeed. As $\frac{w_j^T b}{w_j^T A w_j}$ is a scalar for each j, we can write Ax as

$$Ax = \sum_{j=0}^{m-1} \frac{w_j^T b}{w_j^T A w_j} A w_j \quad . \tag{23}$$

Hence for every $l \in \{0, 1, \ldots, m-1\}$ we have

$$\begin{aligned} w_l^T A x &= \sum_{j=0}^{m-1} \frac{w_j^T b}{w_j^T A w_j} w_l^T A w_j \\ &= w_l^T b \quad , \end{aligned} \tag{24}$$

because w_l is A-orthogonal to all $w_j, j \neq l$. This means

$$w_l^T (Ax - b) = 0 \quad . \tag{25}$$

We can also conclude that

$$\begin{aligned} w_0 &= b \\ w_1 &= A w_0 - c_{10} w_0 \\ w_2 &= A w_1 - c_{20} w_0 - c_{21} w_1 \\ &\vdots \\ w_{m-1} &= A w_{m-2} - c_{m-1,0} w_0 - \ldots \; c_{m-1,m-2} w_{m-2} \\ w_m &= A w_{m-1} - c_{m,0} w_0 - \ldots \; c_{m,m-1} w_{m-1} \quad . \end{aligned} \tag{26}$$

Since $w_m = 0$, we can see that the vectors $w_0, w_1, \ldots w_{m-1}$ exactly span b and $Aw_0, Aw_1, \ldots, Aw_{m-1}$. From (23) and (26) it follows that

$$\begin{aligned} Ax - b &\in \text{span} \; (\; A w_0, A w_1, \ldots, A w_{m-1}, b) \\ &= \text{span} \; (\; w_0, w_1, \ldots, w_{m-1}) \quad . \end{aligned} \tag{27}$$

This conclusion is very important, because it shows that $Ax - b$ is in the span of the vectors $w_0, w_1, \ldots, w_{m-1}$. In view of this, equation (25) implies $(Ax - b)^T (Ax - b) = 0$. In conclusion, we must have $Ax = b$, hence x is the exact solution as claimed.

2.2 The Block Lanczos algorithm

Instead of the field of real numbers, we will now work over the field GF(2), which is precisely equal to the field $\mathbf{Z}_2 = \{0, 1\}$ (we refer to article [17]). So, let A be a symmetric matrix of size $n \times n$ and b an n-vector, both of them over GF(2). Now we will explain how the Block Lanczos algorithm works to solve $Ax = b$, where we also require that x is a vector over GF(2).

In the previous subsection we described how to use the Lanczos recursion for solving a linear system. However, instead of a single-vector recursion as above, we will use a block version of this recursion.

Here instead of vectors, we will work with subspaces, therefore we need some definitions.

Definition 7 *Let $\mathcal{V}$ and $\mathcal{W}$ be subspaces of $GF(2)^n$. Then, $\mathcal{V}$ and $\mathcal{W}$ are called **A-orthogonal** subspaces, if $v^T A w = 0$ for all $v \in \mathcal{V},\ w \in \mathcal{W}$. This is written as $\mathcal{V}^T A \mathcal{W} = 0$.*

Definition 8 *A subspace $\mathcal{W} \subseteq GF(2)^n$ is said to be **A-invertible**, if it has a basis W of column vectors such that $W^T A W$ is invertible.*

In the above definition, it does not matter which basis we choose, because every two bases can be transformed into each other with an invertible transformation.

Applying subspaces in Lanczos means that instead of a sequence of vectors $w_0, w_1, \ldots, w_{m-1}$ we build a sequence of subspaces $\mathcal{W}_0, \mathcal{W}_1, \ldots, \mathcal{W}_{m-1}$ with the following properties:

$$\begin{aligned} \mathcal{W}_i \quad & \text{is } A\text{-invertible} \ , \\ \mathcal{W}_j^T A \mathcal{W}_i = 0 \quad & (i \neq j) \ , \\ A\mathcal{W} \subseteq \mathcal{W}, \quad & \text{where } \mathcal{W} = \mathcal{W}_0 + \mathcal{W}_1 + \ldots + \mathcal{W}_{m-1} \ . \end{aligned} \tag{28}$$

Let W_j form a basis of the subspace $\mathcal{W}_j$. Then

$$x = \sum_{j=0}^{m-1} W_j (W_j^T A W_j)^{-1} W_j^T b \ . \tag{29}$$

solves $Ax = b$ over GF(2). The proof is similar to that of the non-blocked case. Expression (29) generates (18) for subspaces of dimension 1.

The matrix W_i generated in process iteration i has to be A-invertible. Therefore we do the following. First we take an initial matrix V_0 of size $n \times N$, where N is the computer word size (usually 32 or 64 bit). Now let W_0 consist of as many columns of V_0 as possible, subject to the requirement that W_0 is A-invertible. We proceed by building a matrix V_1, also of size $n \times N$, which is A-orthogonal to W_0. Next, we construct W_1 from columns of V_1 just as above. In general, at step i, we build a matrix V_i which is A-orthogonal to all earlier W_j, and we build W_i from V_i.

The selection of the columns of V_i that are included in W_i can be written as

$$W_i = V_i S_i \quad , \tag{30}$$

where matrix S_i has the following properties: (1) S_i is of size $N \times N_i$, where N_i equals the number of columns selected, (2) each column of S_i contains exactly one entry which equals 1 and all other entries in the column are zeros, (3) entry (j,k) of S_i equals 1 if we wish to select column j from V_i. For example, if $N = 3$, then

$$S_i = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{31}$$

selects the second and the third columns of V_i.

Now we describe how to construct V_{i+1} at step i, given that each column of V_i is A-orthogonal to $W_0, W_1, \ldots, W_{i-1}$. Let

$$\begin{aligned} V_{i+1} &= AW_iS_i^T + V_i - \sum_{j=0}^{i} W_j C_{i+1,j} \quad (i \geq 0), \quad \text{with} \\ C_{i+1,j} &= (W_j^T A W_j)^{-1} W_j^T A (AW_i S_i^T + V_i) \quad . \end{aligned} \tag{32}$$

The interpretation is as follows. The columns of V_i which have not been selected for membership in W_i are A-orthogonal to $W_0, W_1, \ldots, W_{i-1}$ (because of the induction assumption), but unfortunately not to W_i. The term $V_i - W_i C_{i+1,i}$ in (32) is used to select those columns to take care that they become A-orthogonal to W_i as well.

We stop with the iteration when at some step m we get $V_m^T A V_m = 0$.

Let

$$\mathcal{W}_i = \text{ span } < W_i > \quad , \tag{33}$$

which means that the columns of W_i span the subspace $\mathcal{W}_i$. By construction, $\mathcal{W}_i$ satisfies the first two properties of (28). As shown in [17], if $\mathcal{W}$ denotes $\mathcal{W}_0 + \mathcal{W}_1 + \ldots, +\mathcal{W}_{m-1}$, then $A\mathcal{W} \subset \mathcal{W}$, hence (28) is completely satisfied.

However, applying Block Lanczos in the above way would take too much computational time, therefore we try to find a recurrence again, hoping that most of the terms will vanish.

It can indeed be achieved that all vectors in V_j are used in at most two steps, namely each column vector will either be used in W_j or in W_{j+1}. Then, just like in the standard Lanzcos procedure, the calculation of V_{i+1} simplifies to the recurrence

$$V_{i+1} = AW_iS_i^T + V_i - W_iC_{i+1,i} - W_{i-1}C_{i+1,i-1} - W_{i-2}C_{i+1,i-2} \quad . \tag{34}$$

Write

$$W_i^{\text{inv}} = S_i(W_i^TAW_i)^{-1}S_i^T = S_i(S_i^TV_i^TAV_iS_i)^{-1}S_i^T \quad . \tag{35}$$

Then after a few steps, see [17], we obtain the recurrence

$$V_{i+1} = AV_iS_iS_i^T + V_iD_{i+1} + V_{i-1}E_{i+1} + V_{i-2}F_{i+1} \quad . \tag{36}$$

$$\begin{aligned} D_{i+1} &= I_N - W_i^{\text{inv}}(V_i^TA^2V_iS_iS_i^T + V_i^TAV_i) \\ E_{i+1} &= -W_{i-1}^{\text{inv}}V_i^TAV_iS_iS_i^T \\ F_{i+1} &= -W_{i-2}^{\text{inv}}(I_N - V_{i-1}^TAV_{i-1}W_{i-1}^{\text{inv}}) \\ &\quad (V_{i-1}^TA^2V_{i-1}S_{i-1}S_{i-1}^T + V_{i-1}^TAV_{i-1})S_iS_i^T \quad . \end{aligned} \tag{37}$$

The Block Lanczos recurrence over GF(2) in (36) saves a lot of time in our application due to the fact that the matrix B has only elements 0 or 1. Computers work very efficiently with bitwise operators, where a bit can be equal to 0 or 1. Let N be the computer word size. During the Block Lanczos algorithm, we generate a sequence of subspaces $\mathcal{W}_i$ with the help of matrices V_i of size $n \times N$. Therefore, every row of these matrices can be stored as one integer. This implies that a whole matrix V_i can be stored in a vector of n integers. So, all the matrix multiplications during the algorithm can be simplified to vector operations. Also, because everything is in $GF(2)$, we can use bitwise operators instead of adding, multiplying, and subtracting. These characteristics will make the algorithm really powerful.

The pseudo code of the Block Lanczos process can be found in Algorithm 1.

Algorithm 1 The pseudo code of Block Lanczos.

Input: Matrices B of size $n_1 \times n_2$ and Y of size $n_2 \times N$

Output: The matrices X and V_m

Cmt: The algorithm finds solutions of $B^T(BX) = V_0 = B^T(BY)$.

1.Initialize: $W_{-2}^{\text{inv}}{}^{N\times N} = W_{-1}^{\text{inv}}{}^{N\times N} = 0$

$V_{-2}{}^{n_2\times N} = V_{-1}{}^{n_2\times N} = 0$

$BV_{-1}{}^{n_1\times N} = 0$

$V^TB^T{}_{-1}^{N\times n_1} = 0$

$SS^T{}_{-1}^{N\times N} = I_N$

$X^{n_2\times N} = 0$

2. $V_0{}^{n_2\times N} = B^T * (B * Y)$

3. $BV_0{}^{n_1\times N} = B * V_0$

4. $V^TB^T{}_0{}^{N\times n_1} = V_0^T * B^T$

5. $Cond_0{}^{N\times N} = V^TB_0^T * BV_0$

6. $i = 0$

while $Cond_i \neq 0$ **do**

8. $[W_i^{\text{inv}}, SS_i^T]$ = generateMatricesWandS $(Cond_i, SS_{i-1}^T, N, i)$

9. $X = X + V_i * (W_i^{\text{inv}} * (V_i^T * V_0))$

10. $B^TBV_i = B^T * BV_i$

$K_i^{N\times N} = (V^TB_i^T * (B * (B^TBV_i))) * SS_i^T + Cond_i$

11. $D_{i+1} = I_N - W_i^{\text{inv}}(K_i)$

12. $E_{i+1} = -W_{i-1}^{\text{inv}}(Cond_i * SS_i^T)$

13. $F_{i+1} = -W_{i-2}^{\text{inv}}(I_N - Cond_{i-1} * W_{i-1}^{\text{inv}})(K_{i-1})SS_i^T$

14. $V_{i+1} = B^TBV_i * SS_i^T + V_i * D_{i+1} + V_{i-1} * E_{i+1} + V_{i-2} * F_{i+1}$

15. $V^TB_{i+1}^T = V_{i+1}^T * B^T$

16. $BV_{i+1} = B * V_{i+1}$

17. $Cond_{i+1} = V^TB_{i+1}^T * BV_{i+1}$

18. $i = i + 1$

20. $V_m = V_i$

21. return X and V_m.

2.3 Applying Block Lanczos to solve $Bx = 0$

In this subsection we will explain how to find the null space of a matrix B of size $n_1 \times n_2$ by using the Lanczos algorithm, which however requires a symmetric input matrix. As was explained above, instead of B, we use the matrix $A = B^T B$, because it is symmetric. Then, every solution of $Bx = 0$ will also solve $Ax = 0$.

Calculating the null space has the following three main phases: preprocessing, Lanczos procedure, and post processing. Let N be the computer word size (this is most of the time 32 or 64 bit). During the preprocessing, we generate a random matrix Y of size $n_2 \times N$ in $GF(2)$. During the Lanczos procedure we try to find a matrix X that satisfies $AX = AY$. Here, AY is the given right-hand side vector, hence it is a linear system. The main idea is that if we succeed in finding such an X then, because of $A(X - Y) = 0$, the column vectors of $X - Y$ will belong to the null space of A.

The preprocessing phase needs no further explanation. In the second phase, the Lanczos algorithm tries to find a matrix X as described above. The process starts with the initial matrix $V_0 = AY$ and terminates at step m if subspace V_m satisfies $V_m^T A V_m = 0$. Then (29) implies

$$\begin{aligned} X &= \sum_{j=0}^{m-1} W_j \left(W_j^T A W_j\right)^{-1} W_j^T V_0 \\ &= \sum_{j=0}^{m-1} V_j W_j^{\text{inv}} V_j^T V_0 \quad . \end{aligned} \tag{38}$$

Let $\mathcal{W} = \mathcal{W}_0 + \mathcal{W}_1 + \ldots + \mathcal{W}_{m-1}$ and $\text{span}(V_m) = \mathcal{W}_m$. Then W_m is A-orthogonal to itself and to W.

We distinguish two cases. If $V_m = 0$ then $AX = AY$. In the other case when $V_m \neq 0$, which happens quite often, we find that V_m also belongs to the null space of A.

Recall that we wanted to find the null space of B, which as we mentioned above, is surely contained in the null space of A. Therefore in the post processing phase, we define a matrix $Z = [X - Y, V_m]$. Of course, if $V_m = 0$, then we just added columns of zeros, which does not make any difference. Afterwards, we compute a basis U of the null space of BZ. Since the size of BZ is $n_1 \times 2N$, the nullspace of BZ has dimension at most $2N$. This means that U contains at most $2N$ basis vectors. Then notice that $B(ZU) = (BZ)U = 0$. Therefore the basis of ZU is included in the nullspace of B. The pseudo code of the main algorithm can be found in Algorithm 2.

Algorithm 2 The pseudo code of findDependency (Main algorithm).

Input: The matrix B
Output: The nullspace X
Cmt: The algorithm finds a part of the nullspace of the given matrix B
1. Select a random $Y^{n_2 \times N}$ matrix in GF(2)
2. $[X, V_m] = \text{blockLanczos}(B, Y, N, n_1, n_2)$
3. $Z = [X - Y, V_m]$
4. $BZ = [B * (X - Y), B * V_m]$
5. $U =$ nullspace of BZ
6. $ZU = Z * U$
7. Return the basis of matrix ZU.

The algorithm may fail, if there exists a non-empty subset C of the column set of $V_0 = B^T BY$ satisfying

$$c_i^T A c_i = 0, \quad \text{where } c_i \in C \quad , \tag{39}$$

which means that the column c_i is A-orthogonal to itself. Then, the process explained above for finding the nullspace does not work, because the columns in C will not be selected in the next two iteration steps for the membership in V, which is a requirement. (The expression in (39) can also be related to the positive definiteness of matrix A.) To avoid the above problem, when generating the random matrix Y, we will choose the columns in such a way that no column is A-orthogonal to itself.

2.4 Basic operations with bitwise operators

In this subsection, we will show how the running time of the algorithm can be decreased by using the fact that every computational operation of the whole process is over the field GF(2). Nowadays, computer operations are executed in the binary number system. Therefore, the basic unit of information, called a bit, can take one of two possible values, 0 and 1.

This leads us to the idea to represent each vector component in the Block Lanczos algorithm (which works over GF(2)) by a bit.

Let N denote the word size of our computer, namely the number of bits needed to represent an unsigned integer. Each unsigned integer has a unique combination of N bits.

Applying the idea to store the elements as bits converts vectors to numbers and certain matrices to vectors, as follows: if we have an N-vector with components from GF(2), this can be stored in one integer, where every bit

a	b	$\&$	$\wedge$
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

Table 5: Bitwise operators acting on 1-bit variables.

of this integer corresponds to a component of the vector. This method can be extended to store matrices of size $n \times N$ or $N \times n$ as an n-vector. For example, let C be a matrix of size $n \times N$ over the field GF(2). This matrix has in every row N elements so that each row can be stored in one integer. Because C has n rows, the total number of integers is equal to n and therefore we obtain an n-vector from a matrix. Hence, the bit decomposition of the ith component in this vector corresponds with the ith row of the matrix. A given matrix D of size $N \times n$ can be stored in the same way as the matrix C with the only difference that here every integer of the vector contains one column of D.

This way of storing data has two advantages. The algorithm needs less memory storage, because every element of the matrices is a bit. Moreover, we can use bitwise operators over integers, which decreases the running time tremendously. The basic bitwise operations we will need are & (AND) and $\wedge$ (XOR), which are defined in Table 5. The appealing property of these operators is that they can be used bitwise over integers. We would like to emphasize that this works faster than performing an N-loop and calculating each bit operation separately (even though the result is the same).

Now we will explain how the operations addition, subtraction, and multiplication can be executed with the help of these bitwise operators.

Clearly, addition and subtraction of two bits can be computed with the help of the operator XOR, see Table 5. (We would like to mention that computing in GF(2) is the same as computing mod 2). On the other hand, multiplication of two bits can be executed by the operator AND.

To add or subtract two N-vectors of elements is the same as applying the operation XOR for the two corresponding integers. For example let $N = 6$ and suppose we would like to add two bit arrays v_1 and v_2, where $v_1 = [1, 1, 1, 0, 1, 0]$ and $v_2 = [1, 0, 0, 1, 0, 0]$. If we compute the result bitwise over GF(2), we get $v = v_1 + v_2 = [1 \wedge 1, 1 \wedge 0, 1 \wedge 0, 0 \wedge 1, 1 \wedge 0, 0 \wedge 0] = [0, 1, 1, 1, 1, 0]$. Let b denote the total cost of one bitwise operation (i.e. XOR or AND) over

one bit. Then the cost of the computation of the bit vector v in the above way is $6 \cdot b$. In our example $v_1 = 58$ ($v_1 = 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$) and $v_2 = 36$ ($v_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$). By applying bitwise XOR over these two integers we obtain the same result $v = v_1 \wedge v_2 = 58 \wedge 36 = 30$. Denote the total cost of one bitwise operation (i.e. XOR or AND) over an integer by I, so the cost of the latter calculation is I. In practice, $I \ll N \cdot b$.

The bitwise multiplication of two N-bits arrays can be done similarly, with the only difference that the operation AND has to be applied. Then the total cost will be $I \ll N \cdot b$.

Suppose we would like to add two matrices C and D, where both of them have size $n \times N$ over GF(2). As described above, we represent them as two n-vectors $C = [c_1, c_2, \ldots, c_n]$ and $D = [d_1, d_2, \ldots, d_n]$. To add the two matrices, each pair (c_i, d_i) has to be XORed, as $c_i \wedge d_i$, with $1 \leq i \leq n$. Thus $E = C + D = [c_1, c_2, \ldots, c_n] \wedge [d_1, d_2, \ldots, d_n] = [c_1 \wedge d_1, c_2 \wedge d_2, \ldots, c_n \wedge d_n] = [e_1, e_2, \ldots, e_n]$. The total computation is $n \cdot I$, i.e., n times the cost of a XOR over integers.

The multiplication of two matrices is more complicated. There are two different ways of computing the product: by inner product and outer product. Let $E = C \cdot D$ where matrix C has size $n \times m$ and D has size $m \times N$. The inner product of C and D is computed as

$$E_{ij} = \sum_{k=1}^{m} c_{ik} \cdot d_{kj} \quad , \tag{40}$$

whereas the outer product is computed as

$$E = \sum_{k=1}^{m} c_k \cdot d_k \quad , \tag{41}$$

where c_k is the kth column of matrix C and d_k is the kth row of matrix D. Note that the matrix D is stored during the implementation as an m-vector.

The most efficient way to compute the inner product E of the two matrices C and D is the following. Suppose we want to compute the element e_{ij}. Then, we have to AND each bit pair (c_{ik}, d_{kj}) $(1 \leq k \leq m)$ with each other and XOR these pairs such that $e_{ij} = (c_{i1} \& d_{1j}) \wedge (c_{i2} \& d_{2j}) \wedge \ldots \wedge (c_{im} \& d_{mj})$. Because E is of size $n \times N$, during the matrix multiplication we need to compute $n \times N$ elements. For each element we need m AND operations and $m - 1$ XOR operations on a bitwise level. The total cost of the matrix multiplication is thus $n \cdot N \cdot (m \cdot b + (m-1) \cdot b)) \approx n \cdot N \cdot m \cdot (2b)$. Unfortunately, for inner products we cannot make use of bitwise operations over integers.

Algorithm 3 The pseudo code of the outer product.

```
Input: matrix C of size n × m and D of size m × N
Output: matrix E of size n × N, E = CD.
Initialize the n-vector E = 0
for i = 1, ..., n do
  for k = 1, ..., m do
    if c_ik = 1 then
      e_i = e_i ∧ d_k
```

The most efficient way to compute the outer product of C and D is given in Algorithm 3 and works as follows. The result E has size $n \times N$, which is represented as an n-vector. First we initialize the vector E as $e_i = 0$ with $1 \leq i \leq n$ (so e_i represents row i of matrix E). Then we take a for-loop inside every column k of C and check the value of every bit. If $c_{ik} = 1$ then we XOR row d_k with row e_i, and we set $e_i = d_k \wedge e_i$. On the other hand if $c_{ik} = 0$, we do nothing. Due to the fact that the matrices D and E are stored by their rows in a vector, we can apply the XOR operator over the two integers d_k and e_i. Denote the total number of nonzero elements in matrix C by $nz(C)$. Then we need $nz(C)$ XOR operations over integers. The total cost of the outer product of the two matrices C and D is thus $m \cdot n + nz(C)I \ll m \cdot nN(2b)$. It is clear that the calculation of outer products is more efficient than the calculation of inner products, because here we could use XOR over integers.

As a consequence, the matrix-matrix outer product of C and D can be seen as a matrix-vector multiplication. Since we already know that outer products can be calculated more efficiently than inner products, we try to use them as often as possible. Finally note that if C has size $N \times m$ instead of $n \times m$ then we obtain a vector-vector operation. However, we cannot exploit this to reduce the running time further.

2.5 Cost analysis

In this subsection we are going to give the total number of operations and their costs in the sequential algorithm. We start with the cost of the algorithm for finding the inverse matrix W^{-1}. There is a for-loop of length N in which there are two computations that take time: 1) finding the pivot of a matrix and 2) XORing a row with the row of the pivot, which is executed at most $N - 1$ times in one iteration. The total cost of this part is equal to $T_{s1} = O(N^2 mI)$.

The Block Lanczos algorithm is already more complicated. Note that matrix $V^T B^T$ in lines $4, 10, 15$ and 17 in the pseudo code of Block Lanczos is exactly the transpose of matrix BV. Therefore $V^T B^T$ will be not stored; we can compute it with BV instead. There are three kinds of outer products with input vectors u and v:

1. Vector u is $n_2 \times N$ and v is of size $N \times N$. The outer product uv is $n_2 \times N$. This outer product is denoted now by op1.
2. Vector u is of size $N \times N$ and v is of size $N \times N$. The outer product uv is of size $N \times N$. This outer product is denoted now by op2.
3. Vector u is of size $n_1 \times N$ or $n_2 \times N$ and v is of size $n_1 \times N$ or $n_2 \times N$. We want to compute the outer product $u^T v$, which is of size $N \times N$. This outer product is denoted now by op3.

Then the Block Lanczos algorithm needs the following outer products, matrix-vector multiplications and XOR-operations:

- Line 2: 2 matrix-vector multiplications
- Line 3: 1 matrix-vector multiplication
- Line 5: 1 op3 outer product (1 op3)
- Line 9: 1 n_2-XOR and 3 outer products (1 op1, 1 op2 and 1 op3)
- Line 10: 1 N-XOR, 2 matrix-vector multiplications and 2 outer products (1 op3 and 1 op2)
- Line 11: 1 N-XOR and 1 outer product (1 op2)
- Line 12: 2 outer products (2 op2)
- Line 13: 1 N-XOR and 4 outer products (4 op2)
- Line 14: 3 n_2-XORs and 4 outer products (4 op1)
- Line 16: 1 matrix-vector multiplication
- Line 17: 1 outer product (1 op3)

In this process, lines between 7 and 18 have to be computed in each iteration step i with $i < \text{rank}(B^T B) = m$.

The maximum cost of an outer product occurs if all elements in the vector u are equal to 1, because then after every check an integer XOR operation has to be executed. The total number of operations and their maximum cost in Block Lanczos are

1. $5m$ op1 outer products. The costs are $5m(n_2NI)$.
2. $9m$ op2 outer products. The costs are $9m(N^2I)$.
3. $3m+1$ op3 outer products. The costs are $m(2(n_1NI) + 1(n_2NI)) + 1(n_1NI)$.
4. $3m$ N-XOR operations. The costs are $3m(NI)$.
5. $4m$ n_2-XOR operations. The costs are $4m(n_2I)$.
6. $3m+3$ matrix-vector multiplications. For the matrix-vector multiplication the values of matrix B stay the same. Let the number of non-zeros in matrix B be denoted by $\mathrm{nz}(B)$. Then during the multiplication we need to execute $\mathrm{nz}(B)$ integer XOR operations. Thus the total costs of the matrix-vector multiplications are $(3m+3)(\mathrm{nz}(B)I)$.

Then the total costs of the second part are equal to

$$\begin{aligned} T_{s2} &= 3nz(B)I + n_1NI + \\ &\quad (6n_2N + 9N^2 + 2n_1N + 3N + 4n_2 + 3nz(B))Im \\ &\approx (6n_2 + 2n_1)mNI \quad . \end{aligned} \tag{42}$$

The third part of the main algorithm is finding the nullspace of a matrix (see line 5 and 6 in Algorithm 2). Here we do not explain how the process sequentially works, but its parallel version will be discussed later.

The main algorithm (see Algorithm 2) has the following costs:

- Line 3: 1 n_2-XOR operation. The costs are n_2I.
- Line 4: 2 matrix-vector multiplications. The costs are $2nz(B)I$.
- Line 5 and 6: 1 n_1 for-loop to row-echelon the matrix has costs n_1^2I. To compute the values in U and multiplying with matrix Z costs $2 * 8N^3I + n_2I$. Then costs are $n_1^2I + 2 * 8N^3I + n_2I$.
- Line 7: n_2^2I.

The maximum costs of the third part are

$$T_{s3} = n_2I + 2nz(B)I + n_1^2I + 2 * 8N^3I + n_2I + n_2^2I \quad . \tag{43}$$

Adding the costs of the three part together we get:

$$T_s = T_{s1} + T_{s2} + T_{s3} \tag{44}$$

3 Parallel algorithm

3.1 Data distribution

As was explained in subsection 2.3, matrix B is a large matrix, and therefore it takes much computational work and time to find its null space. To do it faster, we parallelize our algorithm, in other words we use the power of several processors together to compute the solution. Since it is not efficient to store the whole set of data on all the processors, the data need to be distributed.

Suppose we would like to compute Av, where A is a sparse matrix and v a real vector. We only need to store the nonzero elements of A and each nonzero element will be assigned to one of the processors. Because each processor has only a part of the data for the computation, it may need sometimes data stored by another processor. So the processors have to communicate with each other to send and to receive the needed data. It is clear that this communication causes an increase in the total computation time of the algorithm. Therefore, to minimize the communication costs and to spread the computation and the communication evenly over the processors, an efficient data distribution is needed. Such a data distribution has to distribute the elements of the sparse matrix (A) and the input and output vectors (u and v) over the processors in such a way that most computations are locally executed, but it also tries to store the same number of nonzeros on each processor. In the first part of this subsection, the basic idea for our matrix and vector distributions is described, which is a short summary of [25], while in the second part we will explain how this data distribution is applied in our case.

Suppose we would like to compute $u = Av$, where A is a sparse matrix of size $m \times n$ and v is an n-vector. This matrix-vector multiplication consists of four phases:

1. On each processor only a part of the nonzero elements a_{ij} is stored. During the multiplication, each a_{ij} has to be multiplied by v_j, but every component of v is stored only by one processor. Therefore, during the first phase each processor sends its v_j to those processors that possess a nonzero a_{ij}.

2. To compute an element u_i, we need to calculate the inner product $a_i \cdot v = \sum_{j=1}^{n} a_{ij}v_j$. In the second phase each processor calculates $a_{ij}v_j$ for the locally stored nonzeros a_{ij}, and adds the local products of the same row.

3. After the second phase, each processor has a local sum of the products. Note that if one processor owns a complete row, then the multiplication $u_i = a_i \cdot v$ is complete. If the row is distributed over more processors, then these processors have to send their local sums u_{is} to the processor that is going to store u_i.

4. Each processor adds the received sums and thus obtains u_i.

Observe that communication occurs in phase 1 and 3; in other words, in these phases data are sent and received by the processors. The communication volume is defined as the total number of data sent. Let us make the assumption that every element v_j is possessed by a processor which contains a nonzero element a_{ij}. (If the whole column j contains only zeros, then it will be assigned to a processor with fewer components v_j assigned thus far.) It is clear that it does not make any difference for the communication which processor stores the element v_j, given that it has a nonzero a_{ij}. Given this assumption, the communication volume does not depend on the vector distribution.

Definition 9 *A* $\mathbf{k-way}$ **partitioning** *of A is a set $\{A_0, A_1, \ldots, A_{k-1}\}$ of non-empty, mutually disjoint subsets of A that satisfy $\cup_{r=0}^{k-1} A_r = A$.*

Definition 10 *Let A be an $m \times n$ sparse matrix and let $A_0, A_1, \ldots A_{k-1}$ be mutually disjoint subsets of A, where $k \geq 1$. Let p_i be the number of subsets that intersect row i of A and q_j be the number of subsets that intersect column j of A. Define $p'_i = \max(p_i - 1, 0)$ and $q'_j = \max(q_j - 1, 0)$. The communication volume for the subsets $A_0, \ldots, A_{k-1}$ is defined as*

$$V(A_0, \ldots, A_{k-1}) = \sum_{i=0}^{m-1} p'_i + \sum_{j=0}^{n-1} q'_j \quad . \tag{45}$$

With the definition of the communication volume we are able to give the following theorem.

Theorem 2 *([25]) Let A be an $m \times n$ matrix and let $A_0, \ldots, A_{k-1}$ be mutually disjoint subsets of A, where $k \geq 2$. Then*

$$V(A_0, \ldots, A_{k-1}) = V(A_0, \ldots, A_{k-3}, A_{k-2} \cup A_{k-1}) + V(A_{k-2}, A_{k-1}) \quad . \tag{46}$$

For the proof we refer to article [25]. The significance of this theorem is that if we want to split a subset of A into two or more parts, the extra communication costs are independent of the other partitions of the matrix.

Definition 11 *Let A be an $m \times n$ matrix and $A_0, \ldots A_{k-1}$ be mutually disjoint subsets of A, where $k \geq 1$. Then the maximum amount of computational work for the subsets $A_0, \ldots, A_{k-1}$ is*

$$W(A_0, \ldots, A_{k-1}) = 2 \cdot \max_{0 \leq r < k} nz(A_r) \quad . \tag{47}$$

Note that only phases 2 and 4 have computational work.

The goal of the parallelization is to find a P-way partitioning of A with

$$W(A_0, \ldots, A_{P-1}) \leq (1 + \epsilon) \frac{W(A)}{P} \quad , \tag{48}$$

where ϵ is the so-called **load imbalance parameter**. Hence, our aim is to have $\epsilon > 0$ small. (If $\epsilon = 0$, then the parallel algorithm is called perfectly balanced.) Let the number of processors be $P = 2^k$. In the case of $P \geq 2$ we have to divide the matrix, where the splitting function h is defined as

$$(A_0, A_1) \leftarrow h(A, \text{ sign}, \epsilon) \quad . \tag{49}$$

The sign gives the splitting direction of the matrix, which can be vertical or horizontal. It is clear that the splitting of the whole matrix can be done recursively. The process starts with partitioning horizontally (in other words, assigning the rows to the processors) or vertically (in other words, assigning the columns to the processors), and afterwards it continues with cutting the pieces in the best direction at each step of the recursion.

The algorithm selects the elements of the matrix for the processors in such a way that the communication costs are minimal and the load balance criterium is also satisfied. For the binary matrix decomposition when the algorithm starts with a horizontal split and the splitting direction is chosen alternatively, we have $q_j \leq \sqrt{P}$ if the power k is even, and otherwise $q_j \leq \sqrt{2P}$.

Now we will describe how to distribute the vector v over the processors with the help of the distribution of matrix A, with regards to the communication load balance and the even spreading of the vector components. This vector distribution plays an important role, because matrices (other than the sparse matrix involved) are stored as vectors in our Block Lanczos process. Our requirement is that the vector components v_j have to be assigned to a processor which has a nonzero a_{ij}, except if the whole column $a_{*,j}$ is zero. The output vector u has no relation to the input vector v, and it is distributed in a similar way. During the distribution we have three possibilities to decrease the communication: we can minimize the total number of sends or the total number of receives or their sums.

In our application, we have to multiply v by $A = B^T B$, either as $(B^T B)v$ or $B^T(Bv)$. The most efficient way is to choose for the latter possibility. Remark: $A = B^T B$ is never formed explicitly, instead B^T and B are multiplied by the vector. When computing $B^T(Bv)$, phases 1 and 3 exchange roles when we multiply by B and B^T. The vector distribution process of elements v_j first assigns columns to processors that have one or more nonzero a_{ij}'s. The vector component v_j will be assigned to one of the processors that has the least amount of communication so far. But we also have to take care that no processor gets v_j if it has many more elements than the others. This is not minimized by the program Mondriaan [25], but in practice the vector distribution is still reasonably balanced with respect to the number of components.

(For more details, see the article [25].)

There are other iterative methods where, for the calculation of the matrix-vector multiplication Av, the input and output vectors are distributed in the same way and their distributions depend on each other, but in our case we have the freedom to choose the distribution independently, which is a great advantage. In our process we need to compute $B^T Bv$ (hence, the result $B^T Bv$ has the same distribution as the input vector). The multiplication with B and with B^T are the same except their direction. Therefore, for computations of the transposed matrix B^T the distributed B can also be applied and B^T takes the same communication cost now as multiplication by B. This vector partitioning is implemented in the program Mondriaan.

After the explanation of the ideas for the vector- and matrix distributions, we will now illustrate how this is applied in our algorithm. In our Block Lanczos algorithm, as shown in subsection 2.5, there are matrices of four different sizes:

1. A matrix of size $n_1 \times n_2$. Only the matrix B belongs to this group.
2. Matrices of size $n_2 \times N$, namely $Y,\ X,\ V,\ V_0$.
3. A matrix of size $n_1 \times N$. Only matrix BV has this size.
4. Matrices of size $N \times N$. These matrices are W^{inv}, SS^T, $Cond$, K, D, E and F.

The bit matrices in groups $2, 3$, and 4 are stored as integer vectors. Each matrix in group 2 has to be multiplied by the matrix B. Therefore all these vectors will be distributed in the same way and they will be distributed as an input vector assigned by Mondriaan. Observe that the parallelization is efficient, because n_1 and n_2 are large numbers, while the size N is small.

The algorithm that generates W^{inv} and SS^T only needs computations with matrices of size $N \times N$, therefore it makes no sense to parallelize it. We replicate this computation, so that all processors carry it out redundantly. Since matrix $Cond$ is also needed for the computation of W^{inv} and SS^T, it will be replicated as well.

Furthermore, because N is small, matrices K, D, E and F are also replicated. Therefore the total communication time will decrease, at the expense of extra computation. In group 3, there is only one matrix BV, which has to be multiplied by B^T and hence needs an output vector distribution.

In the findDependency algorithm (Algorithm 2), see section 2.3, we also use a bit matrix Z. In the implementation it is stored as two integer-arrays of length n_2: one array is called Z_1 consisting of $X - Y$ of size $n_2 \times N$, while the other array is equal to V_m. The multiplication $B \cdot Z$ is therefore made up of two multiplications. Because the distribution of V_m was optimized for this multiplication, Z_1 has to be distributed in the same way as V_m. Finally, ZU (with maximum size $n_2 \times 2N$) can also be stored in two n_2 arrays, and has to be distributed due to its size n_2.

At every iteration step of the Lanczos process, W^{inv} and SS^T have to be generated which is done in Algorithm (4) based on [[17], Fig.1]. Here, we want to find the inverse in equation (35) of the $N \times N$ input matrix $V_i^T A V_i$. Normally, to find the inverse of matrix $V_i^T A V_i$ we need to bring the left side of the matrix $[V_i^T A V_i | I_N]$ into row-echelon form so that we obtain the identity matrix on the left side. Then we have $[I_N | (V_i^T A V_i)^{-1}]$ where the right side is exactly the inverse of the input matrix. Our algorithm is just a little bit different. First of all, the rows and the columns are renumbered before the row-echelon process in such a way that the columns that have already been selected at a previous step come last (because we would rather like to choose new columns if possible). The other difference is during the row-echelon process. Normally, in step k of the row-echelon process, there has to be a pivot element (thus a nonzero) among the matrix elements (j, k) with $k \leq j \leq N$. If no pivot is found in one of the iteration steps, the input matrix has no inverse. In our process it is different. If there exists no pivot in column k, this column is a linear combination of earlier columns and therefore we will skip this column (and it will be excluded from S_i).

3.2 Matrix-vector multiplication

In our algorithm, matrix-vector multiplications have the form $Av = u$, where v is an n_2-vector, u is an n_1-vector and matrix $A = B^T B$ with B of size $n_1 \times n_2$. As explained already, we do not use the matrix A explicitly, because this

Algorithm 4 The pseudo code of finding the inverse of matrix W_i.

Input: $Cond = V^T B_i^T * BV_i$ and S_{i-1} which was used in W_{i-1}.
Output: $S_i S_i^T$, where the set S is representing the non-zero diagonal elements of this matrix and ${W_i}^{\text{inv}} = S_i(S_i^T * Cond * S_i)^{-1} S_i^T$.
Cmt. Algorithm finds ${W_i}^{\text{inv}}$ with help of the row-echelon process and row operations.
Cmt. Construct an $N \times 2N$ block matrix M with $Cond$ on the left and I_N on the right.
Cmt. Number columns and rows of $Cond$ as $c_1, c_2, \ldots, c_N$, where the columns that were used during the last iteration step are coming last.
$S = \emptyset$.
for $j = 1$ to N **do**
  2. $k = j$
  while $M(j, j) = 0$ and $k \leq N$ **do**
    if $M(k, j) \neq 0$ **then**
      5. Exchange row j and k of M.
    7. $k = k + 1$
  if $M(j, j) \neq 0$ **then**
    10. Add multiples of row j to other rows of M, to zero rest of column j. $S = S \cup \{j\}$.
  else
    12. $k = j$
    while $M(j, j + N) = 0$ and $k \leq N$ **do**
      if $M(k, j + N) \neq 0$ **then**
        15. Exchange row j and k of M.
      17. $k = k + 1$
    19. Add multiples of row j to zero the rest of the column $j + N$.
    20. Zero row j of M.
23. Compute $SS_i^T = S_i * S_i^T$ based on S.
24. Return SS_i^T and W_i^{inv}, which is the right half of the matrix M.

would be expensive both in time and memory. Therefore, a matrix-vector multiplication by A is in fact two matrix-vector multiplications, namely $v' = Bv$ and $u = B^T v'$. Here, v' is an n_1-vector. The four phases of the matrix multiplication were already given in section 3.1. Recall that there are two main phases (phases 1 and 3) in which interprocessor communication is needed. During the first main phase for $v' = Bv$, the processors have to send some distributed components v_i to the other processors, which will need these for their local multiplications in main phase 2.

Recall that the distribution of all n_2-vectors is permanent and during the whole process remains the same. This is also the case for matrix B. This means that in phase 1, for each multiplication a processor needs to send the same components to other processors. Take for example phase 1 with $P = 2$, where P denotes the total number of processors. We call the two processors $P(0)$ and $P(1)$. Let v_0 be assigned to $P(0)$. If $P(1)$ has a value b_{20} in column 0 of matrix B, it needs to receive v_0 from $P(0)$. After phase 2, the processors need to send the local products which belong to other processors. The number of elements sent or received by a processor depends on the distribution of the n_1-vectors, the n_2-vectors, and the non-zero elements of matrix B.

Let us return to the above example. Suppose that v'_2 is assigned to $P(0)$. Then the local sum containing the product $b_{20}v_0$ has to be sent to $P(0)$. Observe that the number of components of v sent in phase 1, and the number of the local sums sent in phase 3 are not influenced by the exact value of the sent or received components, which is also the case for matrix B.

Because the distribution of the n_1-vectors and the n_2-vectors during the whole algorithm is permanent and the same holds for all vectors of the same length, and also for the matrix B, for each multiplication, a processor needs to store and receive the components with the same indices of the n_2-vectors, and also needs to store and receive the components with the same indices of the n_1-vectors. It is clear that the only difference is the value of these components.

Instead of letting each processor, at each multiplication, compute which components it needs to send to or receive from other processors, we rather compute this information at the beginning of the algorithm and use it for each matrix-vector multiplication. Therefore in phase 1 we need:

1. An index array (called n_2 send index array), which contains the indices of the components that need to be sent.

2. Two offset arrays (n_2 send and receive offset array). When sending

data from one processor to another, the processor needs to know at which index of the given array it has to start sending the required amount of data.

This index is called an offset. The n_2 send offset array of length P is meant to store the offsets, while the receive offset array tells us at which index the received components have been written.

3. When receiving data, each processor needs to know the indices of the received components. For this purpose, at the beginning of the process, we also make an n_2 receive index array.

Consider a small example for $P = 3$ given in figure 1. Suppose that vector v has six elements which are assigned to the processors as follows: $P(0)$ has components v_0 and v_2, $P(1)$ has components v_1 and v_5, and $P(2)$ components v_3 and v_4. The distribution of the non-zero elements of matrix B is also given in this figure. It is easy to see that in phase 1, $P(0)$ has to send the v_0 component to $P(2)$, $P(1)$ has to send v_1 to $P(0)$ and v_5 to $P(2)$, while $P(2)$ has to send v_3 and v_4 to $P(0)$ and v_3 to $P(1)$. Processors which have to receive components also possess a list with the indices of the received elements. Thus for example $P(0)$ will receive elements v_1 from $P(1)$ and v_3, v_4 from $P(2)$. The indices of the communicated components are stored in the send and receive index arrays. If a processor wants to send a set of components to another processor, it needs to know the offset of the local array in which the components are stored, the offset in the receiving array and the number of elements to be sent. With the help of the send offset array each processor knows where it has to start sending the data and it can also compute the amount of data it needs to send. For example $P(2)$ sends data from offset 0 to $P(0)$ and from offset 2 to $P(1)$. Furthermore, $P(1)$ starts to send v_3 from offset 2. The sending processor can find in the local receive offset array also where it needs to store the data in the receiving processor.

During the whole program the send and receive index arrays remain unchanged. In phase 1 each processor only needs to go over its own send index array and make an array with the values corresponding to the indices in this array. So $P(2)$ will have an array with three elements in our example, where the first component will be the value of v_3, the second one the value of v_4 and the third one the value of v_3. With this array the processors can start sending the needed data to a receiving array. Afterwards each processor knows, with the help of the receive index data, how the values correspond to the components in the receiving array.

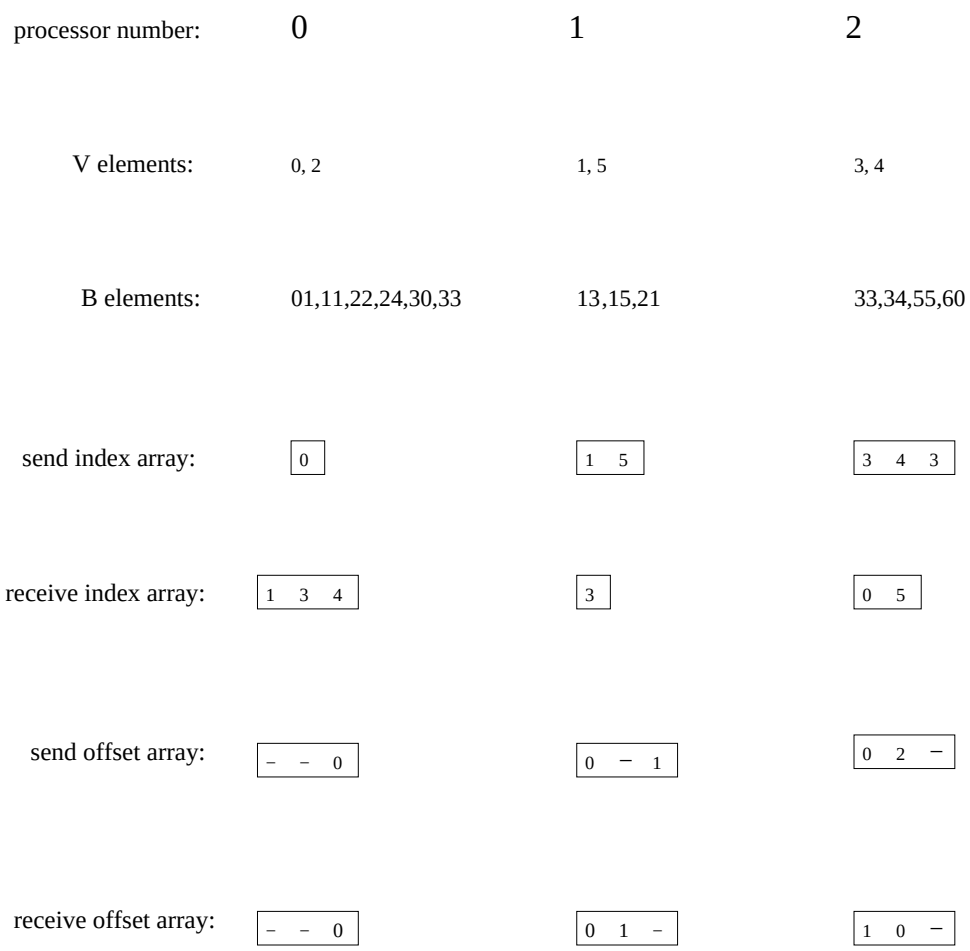

Figure 1: Example for a distribution with $P = 3$ and $n_2 = 6$.

It is clear that for the execution of phase 3, we need similar send and receive index arrays and two offset arrays for the n_1-vector components.

After the multiplication by B, the product $B^T v' = u$ has to be computed. Here, v' is a vector with n_1 components. Observe that for this second multiplication phase 1 becomes phase 3 and phase 3 becomes phase 1 with opposite direction. This means in our above example for $P = 2$ that in phase 1, $P(0)$ will send v'_2 to $P(1)$, and in phase 3 $P(1)$ will send the sum containing the product $b_{20}v'_2$ to $P(0)$.

Because the difference between multiplication of the matrix and its transpose is only directional, the same index arrays can be used with the only difference that sending and receiving change roles, while the offset arrays and offsets are different.

3.3 Remaining parts

In the Block Lanczos algorithm (see Algorithm 1), we need to calculate the outer product multiplication $u^T v = m$, where both u^T and v are integer n-vectors. (Each component of the vector u^T is in fact a columm-vector consisting of N bits; while each component of the vector v is in fact a row-vector consisting of N bits. This way, u^T can be seen as a matrix of size $N \times n$ and v as a matrix of size $n \times N$.) As explained in the previous subsection, for matrix-vector multiplication we could apply each time the same index and offset arrays due to the fact that the distribution of vectors and of the matrix and also the values of the matrix components are constant.

But the values of the two integer vectors u and v for the multiplication $u^T v = m$ are no constants and they change during the algorithm. Now we will explain how this multiplication can be executed in parallel. In the case of $u_i^T \neq 0$ the ith column of u, when we see u as a matrix, has at least one non-zero component. Suppose that this non-zero component is u_{ji}. Then the processor which owns u_{ji} needs to have the value of the jth component of vector v for the multiplication. Here we need no communication, because all vectors with the same length have the same distribution, thus the values u_i and v_i will be distributed to the same processor. This means that each processor can execute the outer product computation in Algorithm 3 without communication. The result at each processor is a vector m with length N. Because all the N-vectors are replicated in our process, each processor needs to send its local vector m to all other processors and XOR the components with the same indices of the received values.

It would also be possible that each processor receives only a part of the local products depending on the indices of the components, and after the

XOR operation each processor will send its part of the end result to the other processors. The advantage is that this way of computing needs less computational work in the end and that this causes less communication volume, NP instead of NP^2. On the other hand, the processors have to communicate in two steps instead of one. Since the unsigned integer size of the computer is normally $N = 32$ or $N = 64$, the two communication steps would cost too much time, especially if we use more processors. Therefore, the first approach is the better option. (We would like to mention that we also implemented the second approach but it took more computational time.)

The other more interesting part of the algorithm is how one finds the nullspace of the matrix BZ in parallel, which has two main phases:

1. To obtain the row-echelon form of the matrix BZ of size $n_1 \times 2N$. (To be able to compute with bitwise operators, the matrix BZ is divided into two matrices of size $n_1 \times N$ in the implementation. Therefore we get much more programming code, but the running time is much better.) In this phase we also need to find the free components of vector x, where x of size $2N$ is the solution of $BZx = 0$. Note that component x_j corresponds to column j in matrix BZ. Component x_j is a free component, if in that column no pivot has been found.

2. The values of the free components and the pivot components are calculated.

The first main step to row-echelon the distributed matrix is an easy job, because we have a row distribution of BZ over the processors. In the beginning, each processor searches for the local pivot in its elements, which will be sent to the other processors. (If a processor does not have a pivot anymore, the other processors will know that without communication). Each processor calculates the global pivot locally. The procedure to find the global pivot is as follows. First we choose the local pivot with the smallest column index. If there are more pivots with the smallest column index, then the global pivot of these pivots will be the one with the smallest row index. Denote this global pivot element by (i, j). To get the row-echelon form, each element of the column of the global pivot has to disappear. Therefore, each row k where the current $BZ_{kj} \neq 0$ will be XORed with row i. In iteration step t, by exchanging rows, we make sure that the global pivot is in row t. (We need extra communication for exchanging rows when these two rows are assigned to different processors.) The other goal of this step is to achieve

a partitioning of the components into two sets (a component can be a free component or a pivot component).

In the second step, the row-echelon matrix will be replicated between the processors. Since we compute in GF(2), each free component can take values 0 or 1, Our goal is to have as many null vectors as possible to find many (X, Y) pairs. Therefore we compute the basis of the null vectors. This can be done as follows. We have as many null vectors as the total number of free components. In each null vector we will set one free component equal to 1 and the others equal to 0. Each time a different free component is set to 1. After the computation of the free components the values of the pivot components can be calculated simply from the equation $(BZx)_i = 0$, because only one pivot value is unknown in this equation. First, we solve the row with the highest index i in which there is a pivot component. Now substituting this value in the other equations, we can similarly proceed upwards in the equations.

3.4 Cost analysis

Recall that the $N \times N$ matrices are replicated in the algorithm. Therefore operations with $N \times N$ matrices need no communication between the processors at all. Therefore, the algorithm to find the inverse matrix W^{inv} does not need to be parallelized, so $T_{p1} = T_{s1} = O(N^2 mI)$.

In the Block Lanczos process the following computations have to be parallelized:

- matrix-vector multiplications, as already explained above
- outer products op3.

Now we turn our attention to the analysis of the costs of these two parallel operations. The parallel process of the outer product op3 has already been explained. The supersteps of this operation are shown in Algorithm 5. Denote the total number of distributed values of the n-vector of $P(s)$ by $nr_n(s)$. Then the first part of superstep 0 has total costs $(\max_{0 \leq s < P} nr_n(s))NI$, while the second part is a communication step where each processor sends N elements to $P-1$ processors and receives $(P-1)*N$ elements. The total cost of this superstep is $(P-1)Ng$ where g is the time to send one data element. During superstep 1 we XOR (over integers) PN elements together. This means total $(P-1)*N$ integer XOR. The total cost of one parallel execution of the outer product op3 is

$$T_{op3} = (\max_{0 \le s < P} nr_n(s))NI + (P-1)Ng + (P-1)NI + l \quad , \tag{50}$$

where l is the synchronization time. For a good distribution, this is about

$$\begin{aligned} T_{op3} &\approx \frac{n}{P}NI + PNg + PNI + l \\ &= nNI + PNg + l \quad , \end{aligned} \tag{51}$$

Algorithm 5 Parallelized outer product operation

{Algorithm op3 which computes $u^T v = y$.}
{Input: u of size $n \times N$ and v of size $n \times N$.}
{Output: y of size $N \times N$.}

{Superstep 0} Initialize vector for $y = 0$. Let $I_n(s)$ be the set of indices of processor s.
for all $i : 1 \le i \le n \wedge i \in I_n(s)$ **do**
  for all $j : 1 \le j \le N$ **do**
    if $u_{ij} = 1$ **then**
      $y[j] = y[j] \wedge v[i]$;

for all $p : 0 \le p < P$ **do**
  for all $j : 1 \le j \le N$ **do**
    Put $y[j]$ in $getResults[s * N + j]$ in $P(p)$;

{Superstep 1}
for all $i : 1 \le i \le P * N$ **do**
  $y[((i-1) \bmod N) + 1] = y[((i-1) \bmod N) + 1] \wedge getResults[i]$;

Now we are going to analyze the total costs of the parallel matrix-vector multiplication. As we see in Algorithm 6, there are 3 supersteps, which involve the 4 phases of the matrix-vector multiplication in section 3.1. Superstep 0 and superstep 1 of these 3 supersteps are those which need interprocessor communication. Denote the length of array $sendIndexArray_1$ by s_1, the length of $sendIndexArray_2$ by s_2, the length of $matrixElements$ by $\mathrm{nz}(B)_s$ and $receiveIndexArray_2$ by r_2. Superstep 0 has total costs $(\max_{0 \le s < P}\ s_1(s)(1+g))$. Superstep 1 has total costs $(\max_{0 \le s < P}\ nz(B)_s)I+$

$(\max_{0\leq s<P}(s_2(s))g$. In the last superstep we need to have $(\max_{0\leq s<P}\ r_2(s))I$ computations. (We only analyze the sends. The costs of the receives are similar). The total cost of executing one parallel matrix-vector multiplication is

$$\begin{aligned} T_{matvec} &= (\max_{0\leq s<P}\ s_1(s)(1+g)) + (\max_{0\leq s<P}\ nz(B)_s)I + \\ &\quad (\max_{0\leq s<P} s_2(s))g + (\max_{0\leq s<P} r_2(s))I + 2l \quad . \end{aligned} \tag{52}$$

Using the costs in (50) and (52), the parallel Block Lanczos algorithm has total costs

$$\begin{aligned} T_{p2} &= 5m(n_2NI) + 9m(N^2I) + (3m+1)(T_{op3}) + \\ &\quad 3m(NI) + 4m(n_2I) + (3m+3)(T_{matvec}) + l \quad , \end{aligned} \tag{53}$$

where the extra synchronization is in the beginning after the memory allocation.

Now we are going to analyze the total costs of the last part of our parallel program and then sum all costs together. Algorithms 7 and 8 show how finding the null space of matrix BZ can be divided into supersteps. Their costs are:

$$\begin{aligned} T_{nullspace} &= 2N\{2Nnr_{n_1}(s) + 2(P-1)g + \\ &\quad 2P + 3 + 2(P-1)g + 2g + 1 + \\ &\quad nr_{n_1}(s)I + 2l\} + \\ &\quad 2N + 2N * 2(P-1)g + 2N(2 + nr_{n_2}(s)) + \\ &\quad 2N * (2 * 4N^2I + nr_{n_2}(s)I) + 2l \quad . \end{aligned} \tag{54}$$

During the computation in (54) the total *pivot* can be maximal $2N$. Moreover, the total $dependingValues$ and $freeValues$ can be also maximal $2N$ and there are maximal $2N$ rows of the row-echelon form of BZ, which contain non-zero elements.

To find the basis of ZU we need to row-echelon the matrix again. Finding this basis costs $2N\{2Nnr_{n_2}(s) + 2(P-1)g + 2P + 3 + 2(P-1)g + 2g + 1 + nr_{n_2}(s)I + 3l\}$.

Then the total cost of the third part is

Algorithm 6 Parallelized matrix-vector multiplication

{Compute Bv.}
{Input: B of size $n_1 \times n_2$ (B is stored with the non-zeros in vector $matrixElement$ as pairs, first the rowNumber and then the columnNumber) and v of size $n_2 \times N$.}
{Output: v' of size $n_1 \times N$.}

{Superstep 0}
for all $i : 1 \leq i \leq s_1$ **do**
  $sendValues_1[i] = v[sendIndexArray_1[i]]$;
for all $i : 1 \leq i \leq s_1$ **do**
  Put $sendValues_1[i]$ to the processor that needs it;

{Superstep 1}
for all $i : 1 \leq i \leq nz(B)_s$ **do**
  if $u[matrixElements[2 * i]]$ is distributed to this processor **then**
    $u[2 * i] = u[2 * i] \wedge v[matrixElements[2 * i + 1]]$;
  else
    The calculated u value has to be sent. Find the place k of this value in the sending array.
    $sendVector_2[k] = sendVector_2[k] \wedge v[matrixElements[2 * i + 1]]$;

for all $i : 1 \leq i \leq length(sendVector_2)$ **do**
  Put $sendVector_2[i]$ to the processor that needs it for the result;

{Superstep 2}
for all $i : 1 \leq i \leq r_2$ **do**
  Check which u value corresponds to $receiveVector[i]$. Let this be value u_j;
  $u_j = u_j \wedge receiveVector[i]$;

$$
\begin{aligned}
T_{p3} \quad = \quad & nr_{n_2}(s)I + 2 * T_{matvec} + T_{nullspace} + \\
& 2N\{2Nnr_{n_2}(s) + 2(P-1)g + \\
& 2P + 3 + 2(P-1)g + 2g + 1 + \\
& nr_{n_2}(s)I + 3l\}
\end{aligned} \tag{55}
$$

and the total cost of the whole parallel algorithm is

$$
T_p = T_{p1} + T_{p2} + T_{p3} \quad . \tag{56}
$$

Algorithm 7 Parallel row-echelon form

{Compute the row-echelon form of matrix BZ.}
{Input: BZ of size $n_1 \times 2N$.}
{Output: BZ in row-echelon form.}

$nrPivots := 0$;
for all $i : 1 \leq i \leq n_1$ **do**
  {Superstep 0}
  for all $j : i \leq j \leq 2N$ **do**
    for all $k : i \leq k \leq n_1 \wedge k \in I_1(s)$ **do**
      if $BZ[k, j] = 1$ **then**
        $pivotRow = k$; $pivotColumn = j$; Exit of for-loop over j;

  for all $p : 0 \leq p < P$ **do**
    Put $pivotRow$ and $pivotColumn$ into $pivotRow[s]$ and into $pivotColumn[s]$ in $P(p)$;

  if there is at least one pivot found by the processors **then**
    {Superstep 1}
    Find the global pivot and the pivotProcessor which owns this global pivot;
    Add $pivotColumn[pivotProcessor]$ into $dependingValues$;
    $pivot := pivotRow[pivotProcessor]$; $nrPivots = nrPivots + 1$;
    if $s = pivotProcessor$ **then**
      for all $p : 0 \leq p < P$ **do**
        Put BZ_{pivot} in $rowToXOR$ in $P(p)$;
    if row BZ_i is assigned to $P(s)$ **then**
      Put BZ_i in BZ_{pivot} in $P(pivotProcessor)$;
    $BZ_i = rowToXOR$;

    {Superstep 2}
    for all $l : 1 \leq l \leq n_1 \wedge l \neq i \wedge l \in I_1(s)$ **do**
      if $BZ[l, pivotColumn[pivotProcessor]] = 1$ **then**
        $BZ[l] = BZ[l] \wedge rowToXOR$;
  else
    Exit of for-loop over i;

Algorithm 8 Parallelized algorithm for finding nullspace and its multiplication with a matrix

{Find the nullspace U of the matrix BZ of size $n_1 \times 2N$ and compute ZU.}

{Superstep 0}
Call Algorithm 7 to find the row-echelon form of BZ;
for all $j : 1 \leq j \leq 2N$ **do**
  if $j \notin dependingValues$ **then**
    Add j into $freeValues$;
for all $i : 1 \leq i \leq n_1 \wedge i \in I_1(s)$ **do**
  if row $BZ[i] \neq 0$ **then**
    for all $p : 0 \leq p < P$ **do**
      Put BZ_i in BZ_i^{glob} in $P(p)$;

{Superstep 1}
$nullVector = 0$;
for all $j : 1 \leq j \leq \#freeValues$ **do**
  $k = freeValues[j]$;
  $nullVector[k, j] = 1$;
  for all $l : 1 \leq l \leq n_2 \wedge l \in I_2(s)$ **do**
    $ZU[l, j] = Z[l, k]$;
for all $j : 1 \leq j \leq \#freeValues$ **do**
  for all $i : i = nrPivots$ to 1 $step - 1$ **do**
    Compute $nullVector[dependingValues[i], j]$ using BZ_i^{glob};
  if $nullVector[dependingValues[i], j] = 1$ **then**
    for all $l : 1 \leq l \leq n_2 \wedge l \in I_2(s)$ **do**
      $ZU[l, j] = Z[l, dependingValues[i]] \wedge ZU[l, j]$;

p	g	l	$T_{comm}(0)$
1	363	805	1616
2	274	17113	5093
4	345	43778	8325
8	412	115516	17310
16	409	299802	51035

Table 6: Benchmarked BSP parameters p, g, l and the time of a 0-relation. All times are in flop units.

name	*rows*	*columns*	*nonzeros*
c82	16307	16338	507716
c98a	56243	56274	2075889

Table 7: The properties of the two given matrices.

4 Experimental results

In this section we present the results of our parallel implementation to find the null space of a given matrix.

The tests were carried out on the Teras supercomputer (the 1024 processor SGI Origin 3800 of SARA in Amsterdam), the main supercomputer in the Netherlands. BSPlib's default values for the BSP parameters can be found in Table 6 for $P = 8$.

We had two matrices, c82 and c98a to analyze. These matrices are from the MPQS algorithm used to factor integers with 82 and 98 decimal digits, respectively and their properties are given in Table 7.

We applied three different kinds of analysis for the scalability of our parallel program: 1) running time, 2) relative speedup and 3) relative efficiency. The running time is equal to the total computing time, which is denoted by T_P for P processors. The relative speedup, denoted by S_P, is equal to

$$S_P = \frac{T_1}{T_P} \quad ; \tag{57}$$

whereas the relative efficiency is defined as

$$E_P = \frac{T_1}{PT_P} \quad . \tag{58}$$

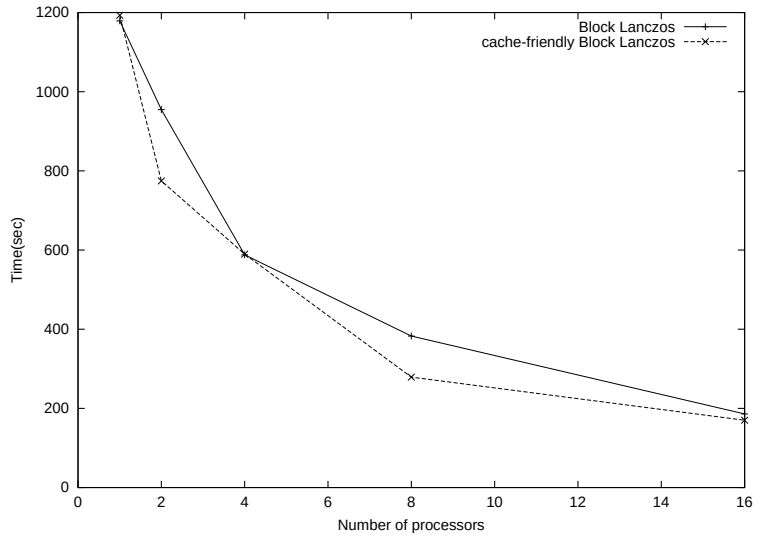

Figure 2: The running time of Block Lanczos for matrix c98 with and without cache friendly implementation.

From now on, we are going to focus on the results with the larger matrix, thus with matrix c98. However, the graphics for running time, relative speedup, and relative efficiency for matrix $c82$ can be found at the end of this section in Figure 5, Figure 6, and Figure 7. During the local parallel matrix-vector multiplication we applied a cache-memory friendly method for the computation. This means that we divided the matrix into four equal squares with numbers $0, 1, 2$, and 3. Then the nonzero values in the matrix are sorted in such a way, that first the non-zeros of the square 0 are handled, then those from square 1 and so on. This means that during the multiplications the matrix elements which come after each other belong to the same square and therefore the processor does not need to ask for other parts of vector u or vector v in the cache memory. The running times without and with this optimization are shown in Figure 2. From the results we can see that this optimization improves the running time. It is expected that this optimization, perhaps further refined, will save even more time for huge matrices.

Most of the time of the whole algorithm is consumed by the Block Lanczos algorithm, see Algorithm 1. Now, we are going to analyze the computation costs of each line of the pseudocode for the sequential algorithm and for the parallelized algorithm for $P = 8$. The running times as a percentage

line number	*sequential*	*parallel*
8	0.00	0.03
9	7.55	4.91
10	39.46	42.96
11	0.00	0.01
12	0.00	0.01
13	0.000	0.00
14	15.36	10.30
16	18.77	20.86
17	18.81	20.88

Table 8: Comparing the costs of the sequential and parallel Block Lanczos program, as a percentage of total time.

of the total time can be found in Table 8.

In this table, we can see that the lines which need communication and therefore synchronization become more important in the parallel implementation, for example lines 10, 16 and 17. Lines in which there is no communication are much faster and therefore they take a smaller percentage of the whole time (see line 14). From this table, we can also conclude that it was a good choice to replicate all matrices with size $N \times N$ and therefore not to parallelize the algorithm for finding the matrix W^{inv}.

The efficiency and speedup of the algorithm can be found in Figure 3 and Figure 4. For $P = 16$ we achieved a speedup of over 7. One reason for this suboptimal speedup is that during the outer product operations we execute an integer XOR operation if the value is equal to 1. The values of these vectors are changing all the time, and we do not know in which part of the matrix there will be more ones than in other parts. Therefore the distribution of the vectors with length n_1 or n_2 can not take this problem into consideration. Another reason is that the first version of the Mondriaan distribution (which is applied in our algorithm) does not give efficient vector distributions for vectors u and v. The unbalanced vector work causes that a processor which is ready with its operations, still has to wait due to a syncronization for other processors which have more work to do. In each step of the Block Lanczos algorithm, a total of 9 synchronization's needed which makes the achieved speedup of the algorithm somewhat smaller.

The running times of Algorithm 2 for the two matrices are given in Table 9 and in Table 10. In both cases, the running time for $P = 1$ is already

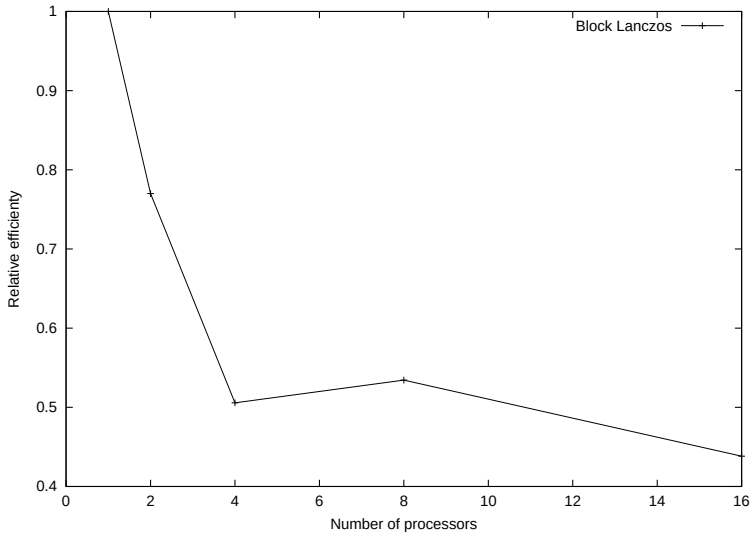

Figure 3: Efficiency for Block Lanczos of matrix c98.

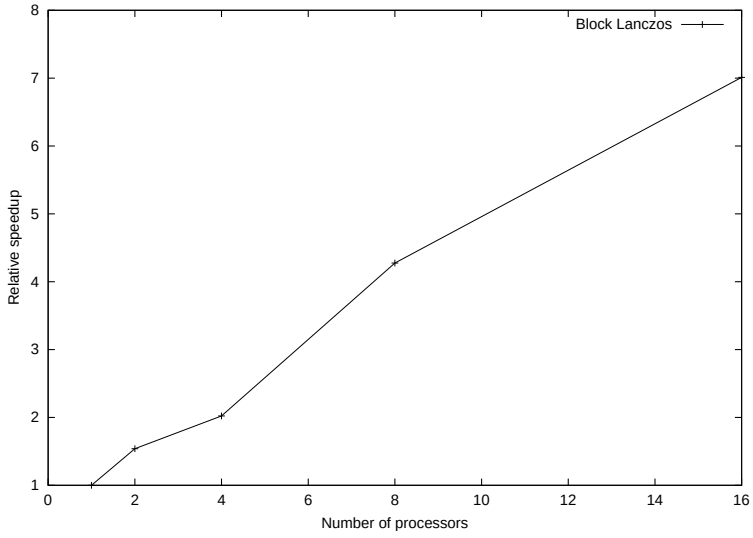

Figure 4: Speedup for Block Lanczos of matrix c98.

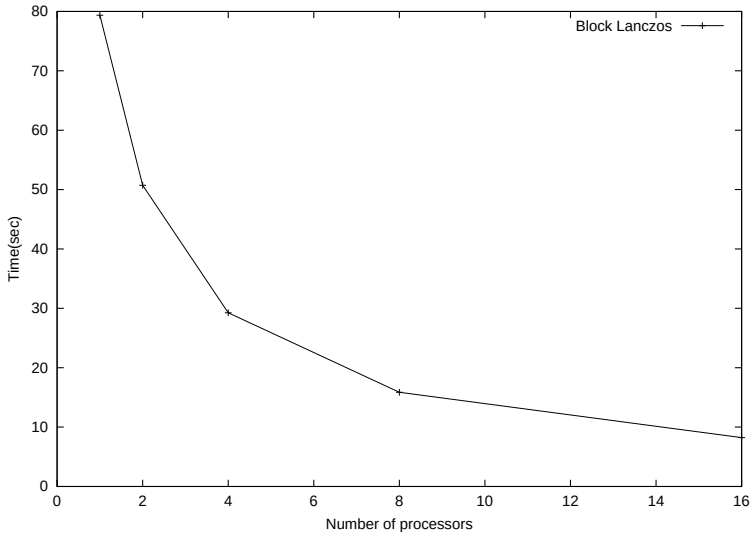

Figure 5: Running time Block Lanczos of matrix $c82$.

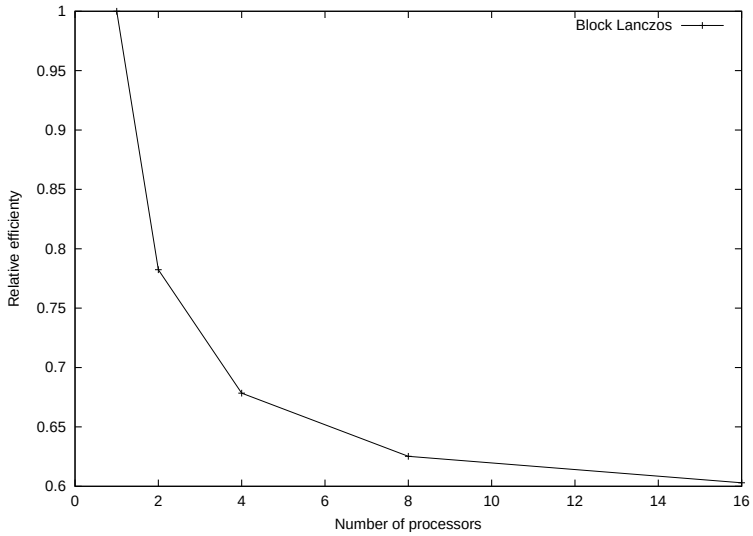

Figure 6: Efficiency for Block Lanczos of matrix $c82$.

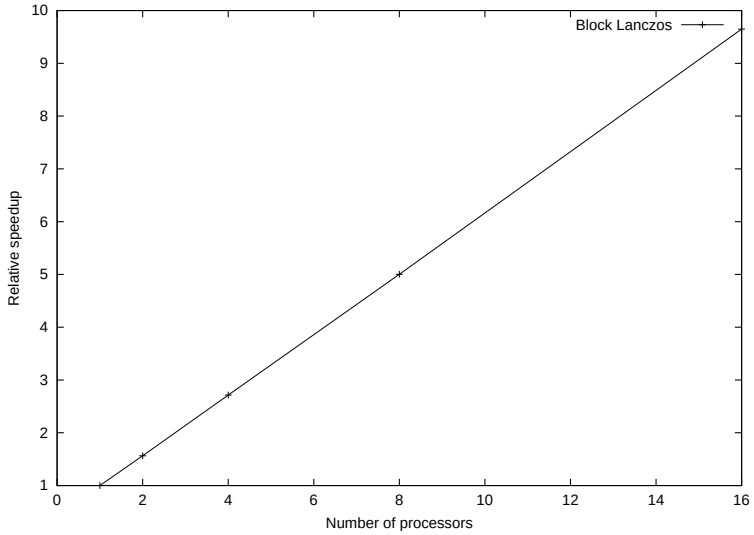

Figure 7: Speedup for Block Lanczos of matrix $c82$.

number of processors	*Time(sec)*
1	0.501
2	0.349
4	0.211
8	0.344
16	0.534

Table 9: The running time for Algorithm 2 of matrix $c82$ after line 2.

really small, therefore the experimental results would be more interesting for a much larger matrix.

number of processors	*Time(sec)*
1	4.483
2	1.240
4	1.015
8	1.193
16	1.554

Table 10: The running time for Algorithm 2 of matrix $c98a$ after line 2.

5 Conclusions and future work

5.1 Conclusions

Nowadays, there is a rapidly growing interest in cryptography, which is hardly surprising in the light of the large number of applications. Cryptography is strongly relying on number theory, because the modern techniques are based on manipulating numbers.
We have discussed how encryption methods work to secure information, and have explained the working of the most widely used encryption method, the so-called RSA algorithm. To find the original message of an encrypted text the unique RSA key is needed, which was applied for the encryption. To attack the RSA algorithm and break its encrypted message we need to find non-trivial prime factors of a known composite number.
Cryptoanalysts developed several factorization techniques to find arbitrarily large prime factors. The bottle-neck of these techniques is finding the nullspace of a huge given bit matrix. In this thesis we have investigated the issues of a parallel approach to find such a nullspace of a given non-symmetric positive-definite matrix.
The Block Lanczos algorithm by Montgomery is applied for finding the nullspace of a matrix in GF(2). Our parallel implementation of the Block Lanczos method contains several improvements. For the distribution of the matrix and vector components the Mondriaan distribution is applied. In contrast with the approach of Montgomery, Mondriaan exploits the matrix sparsity to reduce the communication costs too (so not only the computation costs) and it results in a good balance of communication and computation together. The running time of the whole algorithm is decreased by using bitwise operators over integers due to the fact that every computational operation of the whole process is over the field GF(2). Because the distribution of the vectors and matrices are the same during the whole process, the costs

of the parallel sparse matrix-vector multiplication could be reduced considerably.
The experimental results show that the optimizations have led to an improvement in the running time.

5.2 Future work

However, there are still details in our algorithm which can be improved.
First of all, we have seen that during the outer product operations of two matrices the total number of execution of a bitwise operator is equal to the total number of 1's in one of the matrices. Unfortunately, we do not know before calling the outer product procedure which elements of the two matrices are equal to 1 or to 0. Therefore it can happen that the distribution of the 1's is not efficiently done. If this is the case, one processor needs to execute more bitwise operations than another one so that the computation of the outer product operation is more or less unbalanced. The speedup of our algorithm can grow immediately if the 1's are better distributed over the processors. At this moment we do not have any solution to this problem, and it remains a challenge for the future.
Another important possibility for an improvement is in the usage of the specific diagonal structure of the SS^T matrix. Because of this structure, multiplying by this matrix means zeroing certain rows or columns. This also means that if we have to calculate a product of matrices including SS^T, we do not need to multiply each row with each column because SS^T will zero some rows or columns anyway.
Finally, improvement is expected if the next version of Mondriaan results in a better vector distribution. This would lead to reduced communication costs in our algorithm.

6 References

References

[1] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, PA, 1994.

[2] Beukers, F. Elementaire Getaltheorie, Utrecht University, the Netherlands, 2001.

[3] Brent, R. P., and Pollard, J. M. Factorisation of the Eighth Fermat Number. In *Mathematics of computation* (April 1981), vol. 36, pp. 627–630.

[4] Cavallar, S. Strategies in Filtering in the Number Field Sieve. In *Algorithmic Number Theory - ANTS-IV Delft, the Netherlands* (2000), W. Bosma, Ed., vol. 1838 of Lecture Notes in Computer Science, Springer, pp. 209–231.

[5] Cavallar, S., Dodson, B., Lenstra, A. K., Leyland, P. C., Lioen, W. M., Montgomery, P. L., Murphy, B., te Riele, H., and Zimmermann, P. Factorization of RSA-140 Using the Number Field Sieve. In *ASIACRYPT* (1999), pp. 195–207.

[6] Cavallar, S., Dodson, B., Lenstra, A. K., Lioen, W. M., Montgomery, P. L., Murphy, B., te Riele, H., Aardal, K., Gilchrist, J., Guillerm, G., Leyland, P. C., Marchand, J., Morain, F., Muffett, A., Putnam, C., Putnam, C., and Zimmermann, P. Factorization of a 512-bit RSA Modulus. In *Theory and Application of Cryptographic Techniques* (2000), pp. 1–18.

[7] Cullum, J. K., and Willoughby, R. A. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations Vol.I Theory.* Birkhäuser, Boston, Basel, Stuttgart, 1985.

[8] Diffie, W., and Hellman, M. E. New directions in cryptography. *IEEE Transactions on information theory*, IT-30 (1984), pp. 587–594.

[9] Dongarra, J. J., Duff, I. S., Sorenson, D. C., and van der Vorst, H. A. *Numerical Linear Algebra for High-Performance Computers.* SIAM, Philadelphia, PA, 1998.

[10] Fraleigh, J. B., and Beauregard, R. A. *Linear Algebra*, third ed. Addison-Wesley, 1995.

[11] Golub, G. H., and Van Loan, C. F. *Matrix Computations*, third ed. Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, MD, 1996.

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

[13] Lanczos, C. An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators. *Journal of Research of the National Bureau of Standards 45*, 4 (Oct. 1950), pp. 255–282.

[14] Lenstra, A., Lenstra, H. J., Manasse, M., and Pollard, J. The number field sieve. In *22nd Annual ACM Symposium on the Theory of Computation* (1990), pp. 564–572.

[15] Montgomery, P. A survey of modern integer factorization algorithms. Tech. Rep. Quaterly 7, CWI, Amsterdam, the Netherlands, 1994.

[16] Montgomery, P. L. Square roots of products of algebraic numbers. In *Mathematics of Computation 1943-1993: a Half-Century of Computational Mathematics* (May 1994), Walter Gautschi, Ed., Proceedings of Symposia in Applied Mathematics, American Mathematical Society, pp. 567–571.

[17] Montgomery, P. L. A Block Lanczos Algorithm for Finding Dependencies over GF(2). In *EUROCRYPT '95* (1995), vol. 921, pp. 106–120.

[18] Nguyen, P. A Montgomery-like Square Root for the Number Field Sieve. *Lecture Notes in Computer Science 1423* (June 1998), 151–168.

[19] http://www.rsasecurity.com/rsalabs/challenges/factoring/index.html.

[20] Odlyzko, A. M. The future of integer factorisation. *CryptoBytes 1*, 2 (July 1995), 5–12.

[21] Rivest, R., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public key cryptosystems. *Comm. ACM 21* (February 1978), pp. 120–126.

[22] Singh, S. *The code book: the science of secrecy from ancient Egypt to quantum cryptography*, 1. ed. London: Fourth Estate, 1999.

[23] Stalling, W. *Cryptography and Network Security (Principles and Practice)*, 2nd ed. Prentice Hall, Inc., New Jersey, 1998.

[24] Tel, G. Cryptografie: Theorie, Algoritmen, Toepassingen, Utrecht University, the Netherlands, 2000.

[25] Vastenhouw, B., and Bisseling, R. H. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. Preprint 1238, Department of Mathematics, Utrecht University, Utrecht, the Netherlands, May 2002.