Mondriaan sparse matrix partitioning for attacking cryptosystems by a parallel block Lanczos algorithm — a case study

Rob H. Bisseling[a], Ildikó Flesch[b]

[a]Department of Mathematics, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, The Netherlands (Rob.Bisseling@math.uu.nl)

[b]Department of Information and Knowledge Systems, Institute for Computing and Information Sciences, Radboud University Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands (ildiko@cs.ru.nl)

Abstract. A case study is presented demonstrating the application of the Mondriaan package for sparse matrix partitioning to the field of cryptology. An important step in an integer factorisation attack on the RSA public-key cryptosystem is the solution of a large sparse linear system with 0/1 coefficients, which can be done by the block Lanczos algorithm proposed by Montgomery. We parallelise this algorithm using Mondriaan partitioning and discuss the high-level components needed. A speedup of 8 is obtained on 16 processors of a Silicon Graphics Origin 3800 for the factorisation of an integer with 82 decimal digits, and a speedup of 7 for 98 decimal digits.

Key words: bulk synchronous parallel, cryptology, integer factorisation, matrix partitioning, sparse matrix

1. Introduction

The security of the widely used RSA public-key cryptosystem [18] is based on the fact that finding the prime factors of a large integer is extremely time-consuming. The state-of-the-art in integer factorisation methods tells us how large the keys used in RSA must be to withstand attacks based on trying to find the prime factors for a given public-key value.

On May 9, 2005, Bahr et al. [1] announced a new record factorisation: with help of te Riele and Montgomery they factorised the 200 decimal-digit number originally posed as the RSA-200 challenge in 1991. This factorisation used the Number Field Sieve (NFS) [15], which is currently the best factorisation method for large integers. In practice, the NFS almost always finds a nontrivial factor of a composite number within a few attempts. The two most time-consuming parts of this method are the *sieving step* and the *matrix step*.

The sieving step took from December 2003 to December 2004, and was done by farming out jobs to a variety of computers, taking a total of 55 CPU years (at the equivalent speed of a 2.2 GHz Opteron processor). The matrix step is more tightly coupled and needs more memory since it involves a large sparse matrix, with 64 million rows and columns and 11×10^9 nonzeros for RSA-200. Therefore, it must be carried out on a parallel computer. The matrix step took about three months on a cluster of 80 Opterons. The linear system of the matrix step was solved by a block Wiedemann algorithm [5]. An alternative method would be the block Lanczos algorithm proposed by Montgomery [16].

In the present work, we will discuss the high-level components needed in a parallel computation of the matrix step, such as Mondriaan matrix partitioning. We focus on the block Lanczos algorithm, but the same high-level components are also needed for the block Wiedemann algorithm.

2. Preliminaries

2.1. RSA algorithm

The RSA algorithm was invented in 1977 [18] and it has become one of the most widely used public-key systems. The basic idea of RSA is that there is no efficient algorithm to find nontrivial prime factors of a large composite number. However, it is easy to take two large prime numbers and compute their product.

To explain how RSA works, we need some preliminaries from number theory (see e.g. [6, Chap. 31]). We omit the proofs for the sake of brevity.

Definition 1 *Let x, y be integers and n a natural number. Then x and y are called **congruent modulo** n, if $x - y$ is divisible by n. This is denoted by $x \equiv y \pmod{n}$.*

For example $23 \equiv 10 \pmod{13}$ and $15 \equiv 7 \pmod{8}$.

The relation $\equiv \pmod{n}$ 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. We may simply represent each equivalence class by a single element, giving the set of equivalence classes $\mathbf{Z}_n = \{0, 1, \ldots, n-1\}$. Addition, subtraction, and multiplication of the equivalence classes is defined with the help of these representative elements. With these operations, the set $\mathbf{Z}_n$ becomes a commutative ring.

Definition 2 *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 $ab \equiv 1 \pmod{n}$.*

Definition 3 *The number of elements of $\mathbf{Z}_n^*$ is denoted by $\phi(n)$, and ϕ is called* Euler's totient function.

For example, $\mathbf{Z}_9^* = \{1, 2, 4, 5, 7, 8\}$ and $\phi(9) = 6$.

The next lemma tells us how to calculate ϕ in two special cases.

Lemma 1 *If p is a prime number, then $\mathbf{Z}_p^* = \{1, 2, \ldots, p-1\}$ and therefore $\phi(p) = p-1$. If $n = pq$, where p and q are distinct odd primes, then $\phi(n) = \phi(p)\phi(q) = (p-1)(q-1)$.*

Furthermore, we need the following theorem.

Theorem 1 *(Lagrange) Let n be an integer and $b \in \mathbf{Z}_n$. Then $b^{\phi(n)} \equiv 1 \pmod{n}$.*

A well-known corollary is:

Corollary 1 *(Fermat's Little Theorem) Let p be a prime number and $b \in \mathbf{Z}_p$. Then $b^{p-1} \equiv 1 \pmod{p}$.*

Now, we are in a position to discuss the details of RSA encryption and decryption. The first step is to choose two (large) prime numbers p and q, and to calculate $n = pq$. Because of the primality of p and q, we have

$$\phi(n) = \phi(p)\phi(q) = (p-1)(q-1).$$

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 $d = e^{-1} \in \mathbf{Z}_{\phi(n)}^*$. 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 by computing y^d. Bob obtains the original plaintext, because by the definition of d there is a nonnegative integer k such that $ed = k\phi(n) + 1$, and then by the Lagrange Theorem $y^d = (x^e)^d = x^{ed} = x^{k\phi(n)+1} \equiv x^1 (\text{mod}\, n) = x$, provided n is sufficiently large.

The question is now, why spy 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 coded text. His "only" work is to find the secret key d. To find d, he needs to know $\phi(n)$, but $\phi(n)$ 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.

2.2. The principle of factorisation algorithms

In this subsection, we describe the basic idea of factorisation algorithms that have been developed to find arbitrary large prime factors, such as the NFS. The sieving step in the factorisation of a large number n tries to find many pairs (a_j, b_j), $j = 0, \ldots, n_2 - 1$, of integers such that $a_j \equiv b_j \pmod{n}$ and a_j and b_j are the product of squares and small primes. Let p_i be the ith prime, i.e., $p_1 = 2, p_2 = 3, p_3 = 5$, etc. and let $p_0 = -1$. Then we can write each a_j uniquely as a finite product

$$a_j = \prod_i p_i^{m_{ij}}, \tag{1}$$

where m_{ij} is a nonnegative integer. Note that a_j is square if and only if all exponents m_{ij} are even. Define a matrix A by $a_{ij} = m_{ij}$ mod 2. The matrix A is sparse because integers have only a limited number of prime factors. Define a similar matrix B for the integers b_j.

The matrix step tries to construct a subset of pairs $(a_j, b_j),\ j \in S$, such that $\prod_{j\in S} a_j$ and $\prod_{j\in S} b_j$ are both square. Assume this succeeds, and write $\alpha^2 = \prod_{j\in S} a_j$ and $\beta^2 = \prod_{j\in S} b_j$. We have $(\alpha-\beta)(\alpha+\beta) = \alpha^2 - \beta^2 \equiv 0 \,(\text{mod}\, n)$. Therefore, $\alpha-\beta$ may contain a factor of n. If $\gcd(\alpha\beta, n) = 1$, then $\gcd(\alpha - \beta, n)$, which is a factor of n, has a good chance of being a nontrivial factor, in which case we are done. For an extensive discussion of sieving algorithms, see e.g. the book by Crandall and Pomerance [7].

If we write $S = \{j :\ 0 \le j < n_2 \text{ and } x_j = 1\}$, where $\mathbf{x}$ is an integer vector of length n_2 with 0/1 components x_j, we see that the two products $\prod_{j\in S} a_j$ and $\prod_{j\in S} b_j$ are square if and only if $A\mathbf{x} = 0$ and $B\mathbf{x} = 0$, where all computations are carried out modulo 2, i.e., in the finite field GF(2). Let the $n_1 \times n_2$ matrix C represent the two simultaneous linear systems,

$$C = \begin{bmatrix} A \\ B \end{bmatrix}.$$

Thus, we need to solve $C\mathbf{x} = 0$, or in other words, find the nullspace $\mathcal{N}(C)$. The following example illustrates the whole procedure.

EXAMPLE 1 Let us take the small composite number $n = 33$. We would like to factor 33 and find its nontrivial prime factors 3 and 11. Table 1 gives the pairs (a_j, b_j) and their prime factors found by a certain sieving step. Since we are only interested in the products being square or not, the exponents are simplified by taking them modulo 2. The resulting 0/1-elements are entered into a matrix C, with the top three rows representing the prime factorisations for the a_j, and the bottom four rows those for the b_j,

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 1
The prime factorisations of a set of congruent pairs (a_j, b_j). An entry in row i and column j represents the exponent m_{ij} of the corresponding prime factor p_i in a_j (top) or b_j (bottom).

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}. \tag{2}$$

The vector $\mathbf{x} = [0,0,0,1,0,0,0,0,0,1]^T$ is a solution of $C\mathbf{x} = 0$; the product $C\mathbf{x}$ is the sum of column 3 and column 9, taken mod 2. (Note that the column numbering starts at 0.) These columns represent the pairs $(a_3, b_3) = (28, -5)$ and $(a_9, b_9) = (343, -20)$. Thus $S = \{3, 9\}$, $\alpha = 98$, and $\beta = 10$. Note that gcd$(\alpha\beta, n) =$ gcd$(980, 33) = 1$. This solution is successful because gcd$(\alpha - \beta, n) =$ gcd$(88, 33) = 11$, which is a nontrivial factor of the composite number 33.

It is also possible, especially for large numbers, that there are several solution vectors $\mathbf{x}$. Here, another solution is $\mathbf{x} = [1,1,1,0,0,0,0,1,0,0]^T$. This solution has $S = \{0, 1, 2, 7\}$, $\alpha^2 = 25 \cdot 32 \cdot 1 \cdot 128 = 102400$ and hence $\alpha = 320$, $\beta = 32$, so that gcd$(\alpha\beta, n) =$ gcd$(10240, 33) = 1$, and gcd$(\alpha - \beta, n) =$ gcd$(288, 33) = 3$, resulting in the factor 3.

2.3. Finding the nullspace of a sparse matrix

In the matrix step of the NFS algorithm, the linear system $C\mathbf{x} = 0$ has to be solved, where C is of size $n_1 \times n_2$. Here, n_1 is the total number of prime factors considered, each prime factor occurring at most twice (once for the a_j and once for the b_j), and also considering -1 as a prime factor (needed at most once). Furthermore, n_2 is the total number of pairs found during the sieving step. We usually have $n_1 < n_2$, because each prime can be a prime factor of several a_j and b_j. Because there are many pairs, the matrix C has many columns, and often it is huge. It is a sparse matrix, i.e., many of its elements are zero, which is due to the fact that most primes are not divisors of a given pair (a_j, b_j).

In linear algebra, the classic method to solve $C\mathbf{x} = 0$ is by direct Gaussian elimination (GE), which has a runtime $O(n^3)$ for an $n \times n$ system. GE is efficient for small matrices, but here it would be too slow because of the size of C. Therefore, we rather apply iterative methods, which have the advantage that they involve the matrix only in a multiplicative way, so that it does not change and hence remains sparse. In direct methods such as GE, the matrix fills with new nonzeros during the computation. Note that we consider the iterative methods as exact solvers, computing the full n iterations, instead of as approximate solvers, which is common in linear system solving for floating-point numbers.

Suitable iterative solvers are e.g. the Conjugate Gradient (CG) and the Lanczos methods, which solve the linear system $A\mathbf{x} = \mathbf{b}$ for a symmetric and positive definite matrix A; see [11] for a detailed discussion. In our case, we have $\mathbf{b} = 0$. 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. We choose the Lanczos algorithm, because of the availability of the block version by Montgomery [16].

3. Sequential Lanczos algorithm

3.1. Lanczos algorithm for solving real symmetric linear systems

In this section, we describe how the Lanczos algorithm determines the solution for a linear system $A\mathbf{x} = \mathbf{b}$, where A is a symmetric and positive definite matrix of size $n \times n$. We closely follow the the exposition of Montgomery [16]. The Lanczos algorithm constructs an A-orthogonal and linearly independent set of vectors $\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{m-1}$, where the space spanned by these vectors contains the solution $\mathbf{x}$ for $A\mathbf{x} = \mathbf{b}$. As we will see, the Lanczos algorithm has the advantage that the computation of these vectors at each iteration step can be simplified to a three-term recurrent formula providing an opportunity to save computing time and memory.

The Lanczos iteration can simply be written down in the following general form

$$\begin{aligned} \mathbf{w}_0 &= \mathbf{b}, \\ \mathbf{w}_i &= A\mathbf{w}_{i-1} - \sum_{j=0}^{i-1} c_{ij}\mathbf{w}_j \ \ (i > 0), \ \text{ where } c_{ij} = \frac{\mathbf{w}_j^T A^2 \mathbf{w}_{i-1}}{\mathbf{w}_j^T A \mathbf{w}_j} \ . \end{aligned} \tag{3}$$

Notice that c_{ij} is well-defined because A is positive definite. The iteration will break down at some step m when $\mathbf{w}_m = 0$.

First, we remark that the vectors $\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{m-1}$ are A-**orthogonal**: $\mathbf{w}_i^T A \mathbf{w}_j = 0$ for all $i \neq j$, which can be proven by induction. From the A-orthogonality and positive definiteness of A it follows that $\mathbf{w}_0, \ldots, \mathbf{w}_{m-1}$ are linearly independent. Since the whole space has dimension n, the iteration above stops in at most n steps, i.e., $m \leq n$.

Second, if $i > j + 2$ then by construction and the symmetry of A it holds that $c_{ij} = 0$. Therefore, the definition of $\mathbf{w}_i$ simplifies to the three-term recurrence

$$\mathbf{w}_i = A\mathbf{w}_{i-1} - c_{i,i-1}\mathbf{w}_{i-1} - c_{i,i-2}\mathbf{w}_{i-2} \ \ (i > 1), \tag{4}$$

which requires much less computational work.

Define a vector $\mathbf{x}$ by

$$\mathbf{x} = \sum_{j=0}^{m-1} \frac{\mathbf{w}_j^T \mathbf{b}}{\mathbf{w}_j^T A \mathbf{w}_j} \mathbf{w}_j \ . \tag{5}$$

As $\mathbf{w}_j^T \mathbf{b} / (\mathbf{w}_j^T A \mathbf{w}_j)$ is a scalar for each j, we can write $A\mathbf{x}$ as

$$A\mathbf{x} = \sum_{j=0}^{m-1} \frac{\mathbf{w}_j^T \mathbf{b}}{\mathbf{w}_j^T A \mathbf{w}_j} A\mathbf{w}_j \ . \tag{6}$$

Therefore, $A\mathbf{x} - \mathbf{b} \in \mathrm{span}\{A\mathbf{w}_0, A\mathbf{w}_1, \ldots, A\mathbf{w}_{m-1}, \mathbf{b}\}$. Since $A\mathbf{w}_i \in \mathrm{span}\{\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{i+1}\}$ for $i = 0, \ldots, m-1$ by Equation (3), and $\mathbf{b} = \mathbf{w}_0$, we have $A\mathbf{x} - \mathbf{b} \in \mathrm{span}\{\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_m\}$. Because $\mathbf{w}_m = 0$, this gives

$$A\mathbf{x} - \mathbf{b} \in \mathrm{span}\{\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{m-1}\}. \tag{7}$$

Since $\mathbf{w}_j^T(A\mathbf{x} - \mathbf{b}) = 0$, for $j = 0, \ldots, m-1$, Equation (7) implies that $(A\mathbf{x} - \mathbf{b})^T(A\mathbf{x} - \mathbf{b}) = 0$, meaning that $\mathbf{x}$ is the exact solution of $A\mathbf{x} = \mathbf{b}$.

3.2. Block Lanczos algorithm of Montgomery for solving symmetric linear systems over GF(2)

In this subsection, we briefly summarise the block Lanczos algorithm of Montgomery [16] for solving symmetric linear systems over GF(2). This algorithm generalises the Lanczos recursion of the previous subsection to work with subspaces instead of vectors. For this purpose, we need the following definitions.

Definition 4 *Let $\mathcal{V}$ and $\mathcal{W}$ be subspaces of GF$(2)^n$. Then, $\mathcal{V}$ and $\mathcal{W}$ are called A-**orthogonal** subspaces if $\mathbf{v}^T A\mathbf{w} = 0$ for all $\mathbf{v} \in \mathcal{V},\ \mathbf{w} \in \mathcal{W}$. This is written as $\mathcal{V}^T A\mathcal{W} = 0$.*

Definition 5 *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 $W^T AW$ 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.

Working with subspaces in the Lanczos algorithm means that instead of constructing a sequence of vectors $\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{m-1}$, we construct a sequence of subspaces $\mathcal{W}_0, \mathcal{W}_1, \ldots, \mathcal{W}_{m-1}$ with the following properties:

$$\begin{array}{rl} \mathcal{W}_i & \text{is } A\text{-invertible} \ , \\ \mathcal{W}_i^T A\mathcal{W}_j = 0 & (i \neq j) \ , \\ A\mathcal{W} \subseteq \mathcal{W}, & \text{where } \mathcal{W} = \mathcal{W}_0 + \mathcal{W}_1 + \cdots + \mathcal{W}_{m-1} \ . \end{array} \tag{8}$$

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

$$\mathbf{x} = \sum_{j=0}^{m-1} W_j (W_j^T A W_j)^{-1} W_j^T \mathbf{b} \tag{9}$$

solves $A\mathbf{x} = \mathbf{b}$ over GF(2). Note that the solution $\mathbf{x}$ of the block Lanczos algorithm corresponds to the solution of the Lanczos algorithm for subspaces of dimension 1, see Equation (5).

The following procedure guarantees that the generated subspaces $\mathcal{W}_i$ satisfy the three conditions of Equation (8). We obtain subspace $\mathcal{W}_i$ from an auxiliary matrix V_i. 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. Then, $\mathcal{W}_i$ is the subspace with basis W_i.

The selection of the columns from V_i is expressed by an $N \times N_i$ matrix S_i, such that $W_i = V_i S_i$. Here, N_i is the number of selected columns, $N_i \leq N$. Each column of S_i contains exactly one entry 1 and all other entries in the column are zeros: $(S_i)_{jk} = 1$ if we wish to select column j from V_i as column k in W_i. We select columns at most once, meaning that each row of S_i has at most one entry 1. It can be shown that $S_i^T S_i = I_N$, which is the $N \times N$ identity matrix.

Now we describe how to construct V_{i+1} at step i, given the basis V_i which is A-orthogonal to $W_0, W_1, \ldots, W_{i-1}$ and the basis W_i of selected columns from V_i. Let

$$\begin{aligned} V_{i+1} &= AW_iS_i^T + V_i - \sum_{j=0}^{i} W_j C_{i+1,j} \;\; (i \geq 0), \;\; \text{with} \\ C_{i+1,j} &= (W_j^T A W_j)^{-1} W_j^T A (AW_iS_i^T + V_i) \; . \end{aligned} \tag{10}$$

This is a generalisation of Equation (3); for more details and proofs, see [16]. We stop with the iteration when at some step m we get $V_m^T A V_m = 0$. By construction, $\mathcal{W}_i$ satisfies the first two properties of Equation (8); a proof of the third property can be found in [16].

Similar to the standard Lanczos algorithm, the computation of V_{i+1} can be reduced to a simpler formula, in this case a four-term recurrence

$$V_{i+1} = AW_iS_i^T + V_i - W_iC_{i+1,i} - W_{i-1}C_{i+1,i-1} - W_{i-2}C_{i+1,i-2} \;\; (i > 1). \tag{11}$$

Write $W_i^{\text{inv}} = S_i(W_i^T A W_i)^{-1} S_i^T = S_i(S_i^T V_i^T A V_i S_i)^{-1} S_i^T$. Then, we can rewrite Equation (11) as

$$\begin{aligned} V_{i+1} &= AV_iS_iS_i^T + V_iD_{i+1} + V_{i-1}E_{i+1} + V_{i-2}F_{i+1}, \text{ with} \\ D_{i+1} &= I_N - W_i^{\text{inv}}(V_i^T A^2 V_i S_i S_i^T + V_i^T A V_i), \\ E_{i+1} &= -W_{i-1}^{\text{inv}} V_i^T A V_i S_i S_i^T, \\ F_{i+1} &= -W_{i-2}^{\text{inv}}(I_N - V_{i-1}^T A V_{i-1} W_{i-1}^{\text{inv}})(V_{i-1}^T A^2 V_{i-1} S_{i-1} S_{i-1}^T + V_{i-1}^T A V_{i-1}) S_i S_i^T \; . \end{aligned} \tag{12}$$

3.3. Block Lanczos algorithm for our application

In this section, we explain, how to use the block Lanczos algorithm for our matrix C. To find (part of) the nullspace $\mathcal{N}(C)$ of C, we can apply the block Lanczos algorithm as proposed by Montgomery [16]. Since this algorithm is only suitable for symmetric matrices, it is applied to $A = C^TC$ in such a way that it finds a nullspace $\mathcal{N}(C^TC)$ that is as large as possible. Here, C^T denotes the transpose of the matrix C. There is no need to form the product C^TC explicitly: multiplication by C^TC is carried out as multiplication by C followed by multiplication by C^T. Furthermore, it suffices to store only C. Since $\mathcal{N}(C) \subset \mathcal{N}(C^TC)$, we hope to be able to find some vector $\mathbf{x} \in \mathcal{N}(C)$; this can be done by a postprocessing procedure [16] after the block Lanczos algorithm, which is outlined in the next subsection.

The block Lanczos algorithm is applied to solve $C^TCX = C^TCY$, where X and Y are $n_2 \times N$ matrices. The solution matrix X contains a set of N columns, each representing a solution $\mathbf{x}$. This way, we obtain N solutions in one run of the algorithm. The value of N is chosen as the word size of an integer on the computer used, say $N = 32$. This choice enables storage of the complete matrix X in a single one-dimensional integer array of length n_2. It also allows the use of efficient bitwise operations in the algorithm. The matrix Y is chosen as a random bit matrix. The aim of this approach is to obtain many independent solutions of $C^TC\mathbf{x} = 0$, given by the columns of the matrix $X - Y$.

Algorithm 1 summarises the steps of the block Lanczos algorithm. At the first occurrence of a matrix in the algorithm, its size is given as a superscript, e.g. $V^{n_2 \times N}$. Matrix subscripts denote the order in a sequence of matrices, e.g. V_i. All multiplication operations are explicitly shown by using an asterisk or a circled asterisk, e.g. $C \circledast V$ on line 15. The result matrix is then written as CV. A circled asterisk denotes a multiplication involving a sparse matrix; this operation is emphasised because the matrix is then usually large. There is no need to store both V and V^T; only one of the two matrices suffices. The matrix $Cond_i$ represents the termination condition; $SS_i^T = S_i S_i^T$ represents a set of $N_i \le N$ columns selected from V_i.

The function generateWS generates new matrices W_i^{inv} and SS_i^T. To do this, it needs to compute the inverse of $W_i^T A W_i = S_i^T V_i^T C^T C V_i S_i = S_i^T Cond_i S_i$, where $Cond_i$ is given as input and S_i is computed at the same time as output. The standard method from linear algebra for computing an inverse of an $N \times N$ matrix B is to perform Gauss–Jordan elimination simultaneously on B and on I_N, retrieving I_N and B^{-1} at the end. This involves row swaps in both matrices.

If a column has no nonzeros available as the next pivot for the elimination, the column is skipped, and this is registered in S_i. At the end, this yields the new matrix S_i. The old matrix S_{i-1} is used as input to compute an initial ordering with the old columns (those included in the previous iteration) numbered last. In an implementation, the matrix S_i is stored instead of the product $SS_i^T = S_i S_i^T$; actually, it can be stored most efficiently as a vector. The function generateWS only involves computations on small $N \times N$ matrices.

Algorithm 1 The sequential block Lanczos algorithm.

```
Input: matrices $C^{n_1 \times n_2}$ and $Y^{n_2 \times N}$.
Output: matrices $X^{n_2 \times N}$ and $V_m^{n_2 \times N}$, such that $C^T C X = C^T C Y$ and $V_m^T C^T C V_m = 0$.
Call: $[X, V_m] = \text{blockLanczos}(C, Y, N, n_1, n_2)$.
1. Initialise:
1a. $W_{-2}^{\text{inv}\ N \times N} = W_{-1}^{\text{inv}\ N \times N} = 0$
1b. $V_{-2}^{n_2 \times N} = V_{-1}^{n_2 \times N} = 0$
1c. $CV_{-1}^{n_1 \times N} = 0$
1d. $K_{-1}^{N \times N} = 0$
1e. $SS^T{}_{-1}^{N \times N} = I_N$
1f. $X^{n_2 \times N} = 0$
 2. $V_0^{n_2 \times N} = C^T \circledast (C \circledast Y)$
 3. $CV_0^{n_1 \times N} = C \circledast V_0$
 4. $Cond_0^{N \times N} = (CV_0)^T * CV_0$
 5. $i = 0$
while $Cond_i \neq 0$ do
    7. $[W_i^{\text{inv}}, SS_i^T] = \text{generateWS}(Cond_i, SS_{i-1}^T, N, i)$
    8. $X = X + V_i * (W_i^{\text{inv}} * (V_i^T * V_0))$
    9. $C^T CV_i^{n_2 \times N} = C^T \circledast CV_i$
   10. $K_i = ((CV_i)^T * (C \circledast (C^T CV_i))) * SS_i^T + Cond_i$
   11. $D_{i+1}^{N \times N} = I_N - W_i^{\text{inv}} * K_i$
   12. $E_{i+1}^{N \times N} = -W_{i-1}^{\text{inv}} * (Cond_i * SS_i^T)$
   13. $F_{i+1}^{N \times N} = -W_{i-2}^{\text{inv}} * (I_N - Cond_{i-1} * W_{i-1}^{\text{inv}}) * K_{i-1} * SS_i^T$
   14. $V_{i+1} = C^T CV_i * SS_i^T + V_i * D_{i+1} + V_{i-1} * E_{i+1} + V_{i-2} * F_{i+1}$
   15. $CV_{i+1} = C \circledast V_{i+1}$
   16. $Cond_{i+1} = (CV_{i+1})^T * CV_{i+1}$
   17. $i = i + 1$
18. Return $X$ and $V_m = V_i$.
```

3.4. Computation of the nullspace

The main algorithm for finding the nullspace of a given matrix C is presented as Algorithm 2. The algorithm starts by selecting a random $n_2 \times N$ bit matrix Y. When generating the columns $\mathbf{y}$ of this matrix, it is checked that they satisfy $\mathbf{y}^T C^T C \mathbf{y} \neq 0$. If not, they are replaced by another randomly chosen column. This increases the probability of success of the whole process. We then use the block Lanczos algorithm to find the solution space X_0 of $C^T C X = C^T C Y$ and a space V_m of N vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{N-1}$ with $\mathbf{v}_i^T C^T C \mathbf{v}_j = 0$ for all i, j. We define the matrix Z of size $n_2 \times 2N$ and compute CZ. We then compute a basis U of the nullspace of CZ. Since $C^T C(X_0 - Y) = 0$, we know that the columns of $X_0 - Y$ are contained in this nullspace, but our hope is to find more solutions because $V_m^T C^T C V_m = 0$. This is the reason for extending $X_0 - Y$ with V_m to Z. Since the size of CZ is $n_1 \times 2N$, its nullspace has dimension at most $2N$, so that U contains at most $2N$ basis vectors, each of length $2N$. This nullspace can easily be obtained by Gaussian elimination, because the matrix CZ only has a small number of columns (in contrast to the original matrix C). Because $C(ZU) = (CZ)U = 0$, the basis of ZU is included in the nullspace of C, and hence gives the desired solution space.

Algorithm 2 Algorithm for finding part of the nullspace.

```
Input: matrix $C^{n_1 \times n_2}$.
Output: matrix $X^{n_2 \times r}$ with $CX = 0$.
1. Select a random bit matrix $Y^{n_2 \times N}$
2. $[X_0, V_m]$ = blockLanczos$(C, Y, N, n_1, n_2)$
3. $Z^{n_2 \times 2N} = [X_0 - Y, V_m]$
4. $CZ^{n_1 \times 2N} = [C \circledast (X_0 - Y), C \circledast V_m]$
5. $U^{2N \times k} = \mathcal{N}(CZ)$
6. $ZU^{n_2 \times k} = Z * U$
7. Return $X^{n_2 \times r}$ = basis of matrix $ZU$.
```

4. High-level components for parallel computation

The block Lanczos algorithm requires the following major high-level components:

1. sparse matrix–vector multiplication;
2. sparse matrix partitioning;
3. vector partitioning;
4. dense vector inner-product computation;
5. AXPY operation (‘A times X Plus Y’, with X, Y vectors and A a scalar);
6. global-local indexing mechanism.

These components can be viewed as building blocks that occur in many different applications; for instance, they occur in both the block Lanczos algorithm and the block Wiedemann algorithm for the matrix step, but also in most iterative linear system solvers.

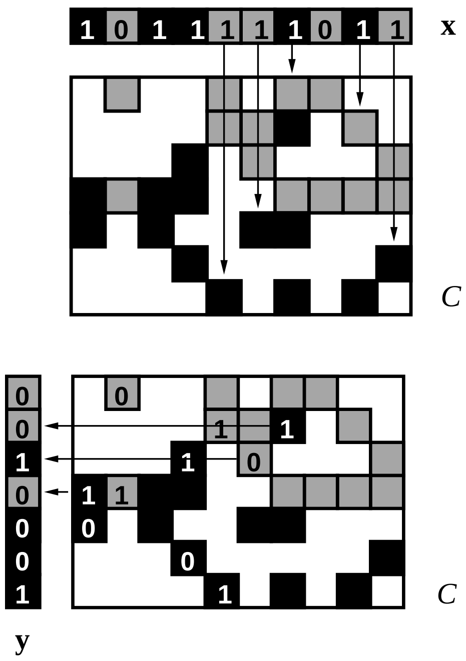

Figure 1. Parallel sparse matrix–vector multiplication $\mathbf{y} = C\mathbf{x}$ for the matrix C from Example 1. The matrix is partitioned for two processors, shown in grey and black. The input and output vectors are treated as dense; their component values are shown. (It is convenient to depict the vector $\mathbf{x}$ above the matrix, as a row vector, even though it is a column vector.) The matrix is treated as sparse; only its nonzeros (i.e., values 1) are owned by processors. Top: vertical arrows denote communication of input values x_i in phase (i). Bottom: each processor in a row obtains a local result by adding products $C_{ij}x_j$ in phase (ii). This result is depicted in the leftmost matrix element that the processor owns in a row. These results are communicated to the owner of y_i in phase (iii), and added to give the result y_i in phase (iv).

4.1. Sparse matrix–vector multiplication

The sparse matrix C or its transpose is multiplied by a vector on lines 2, 3, 9, 10, 15 of Algorithm 1. Here, the sparse bit matrix C is multiplied by an $n_2 \times N$ bit matrix, which can be viewed as N multiplications by a bit vector of length n_2, or the transpose matrix C^T is multiplied by an $n_1 \times N$ bit matrix. The parallel component required is a four-phase algorithm, consisting of: (i) communication of the components of the input vectors to exactly those processors that need them; (ii) local matrix–vector multiplication; (iii) communication of local results to the owner of the corresponding output vector component; and (iv) finally addition of these results. Figure 1 illustrates the parallel sparse matrix–vector multiplication in the simple case of a single vector, with $N = 1$. For more details, see [2, Chap. 4]. This parallel algorithm for sparse matrix–vector multiplication improves upon that used by Montgomery [17] in his parallel version of the block Lanczos algorithm because in our approach the communication exploits the sparsity of the matrix C; in [17], however, the amount of communication is as large as for a dense matrix. The algorithm should work for every possible distribution of the matrix and the input and output vectors. Note that the algorithm is a generalisation of the regular sparse matrix–vector multiplication to the multi-vector case.

4.2. Sparse matrix partitioning

A sparse matrix can be partitioned by any of the available sparse matrix partitioners based on multilevel hypergraph partitioners, such as hMetis [12], Mondriaan [22], Par$k$way [19], PaToH [4], or Zoltan [8,9]. Par$k$way and Zoltan are able to perform the partitioning itself in parallel. In the present work, we use the sequential partitioner Mondriaan.

The Mondriaan sparse matrix partitioner works by recursive bisection, splitting the current submatrix into two submatrices, either by rows or by columns. Both directions are tried, and the direction causing the least communication volume (total number of data words sent) is chosen. As a result, a subset of the rows (or columns) is assigned to one part and the remainder to the other part. This procedure is repeated until the desired number p of submatrices is obtained.

The Mondriaan partitioner tries to assign an equal number of matrix nonzeros to all processors, with a tolerance ϵ specified as input by the user, with the goal of achieving a good load balance in the sparse matrix–vector multiplication. The partitioner guarantees that the number of nonzeros $nz(C_r)$ of processor r satisfies

$$nz(C_r) \leq (1+\epsilon)\frac{nz(C)}{p}, \tag{13}$$

for $r = 0, \ldots, p-1$.

At each split into two parts during the recursive partitioning procedure, a suitable tolerance δ is determined for that split. Since there is a trade-off between load imbalance and communication cost, δ is chosen as large as possible while leaving sufficient freedom for subsequent splits. For $p = 2^q$, the default choice for a first split is $\delta = \epsilon/q$. This choice lets the number of nonzeros of the largest part grow each time by a factor $1 + \delta$, hence yielding a total growth of $(1+\delta)^q \approx 1 + q\delta = 1 + \epsilon$.

An important feature of Mondriaan that helps to keep the communication volume low is that the value of δ is dynamically adjusted after each split. A matrix part with a larger number of nonzeros needs a stricter growth control and hence a smaller δ in subsequent splits than a part with fewer nonzeros.

Another important feature is that the partitioning detects automatically in which direction to perform a split. This cannot always be predicted a priori, although for the first split of integer-factorisation matrices one may argue that the vertical direction is preferred (due to the logarithmically decreasing density of primes). This is illustrated in Fig. 2, where a vertical split of the initial matrix C with $n_1 = 179$ and $n_2 = 210$ into two parts causes fewer communications since the nonzeros are concentrated in the few dense rows at the top (which will inevitably be spread over both parts), leaving many possibilities to avoid communication for the other rows. In contrast, a horizontal split would cause communications for many columns. The matrix is then split again, until $p = 4$ parts are obtained. All splits of Fig. 2 are in the vertical direction. A detailed look at the picture confirms that all nonzeros in one matrix column have indeed the same colour/grey shade, meaning that they are owned by the same processor. After repeated splits, the situation for the current submatrix may be different so that the other direction is preferred; the software detects this.

A single split of a submatrix is carried out by translating the submatrix into a hypergraph. If the split is by column, then each submatrix column j becomes a vertex j in the hypergraph, and row i becomes a hyperedge containing all the vertices j with $a_{ij} \neq 0$. (A *hyperedge* in a hypergraph is a subset of the vertices, of any cardinality; this is a generalisation of an *edge* in a graph, which is a subset of exactly two vertices.) The matrix partitioning problem thus becomes a hypergraph partitioning problem, which is solved efficiently using a multilevel scheme.

The multilevel method first merges similar vertices during a *coarsening phase*, to reduce the problem to a much smaller size. All vertices are merged pairwise, and this procedure is repeated at several

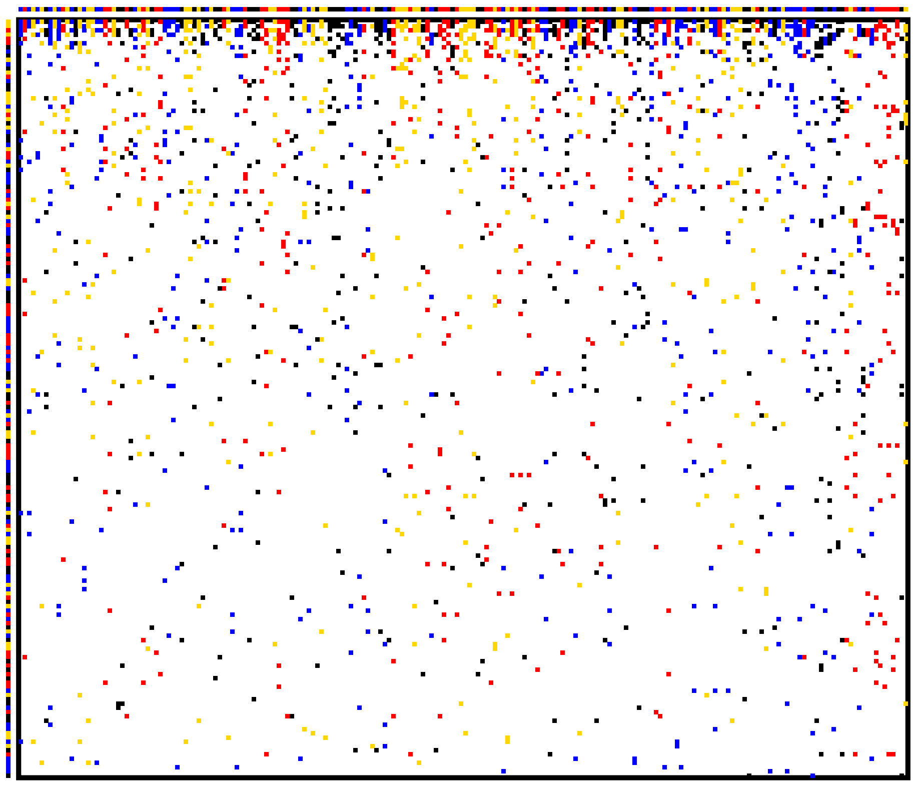

Figure 2. Matrix `mpqs30` corresponding to a sparse linear system $C\mathbf{x} = 0 \pmod 2$ obtained by the Multi-Polynomial Quadratic Sieve (MPQS) method for a 30-decimal integer. The 179×210 matrix C has $nz(C) = 1916$ nonzero elements. Each matrix row represents a prime; each column a pair of integers (a_j, b_j). The primes are sorted in increasing order, where each prime may occur at most twice. The matrix has been partitioned for $p = 4$ processors of a parallel computer, shown in different colours/grey shades, by using the Mondriaan package [22]. The imbalance allowed in the number of nonzeros is $\epsilon = 3\%$. Also shown is a partitioning of the input vector (above the matrix) and output vector (left) for a sparse matrix–vector multiplication $\mathbf{y} = C\mathbf{x}$. Matrix source: courtesy of Richard Brent.

levels. Two vertices are similar if they occur together in many hyperedges. The resulting smaller hypergraph is then partitioned using either a greedy method or more sophisticated methods, such as the Kernighan–Lin [13] and Fiduccia–Mattheyses [10] algorithms, and then the merged vertices are taken apart in a *refinement phase*, also consisting of several levels. The partitioning is further improved during this phase by small adjustments at each level.

Figure 2 shows a global view for $p = 4$ of the partitioned example matrix C. Because submatrices need not be contiguous, but can be scattered, the resulting matrix partitioning may seem highly irregular in this global view, and even more so if both dimensions are split, as happens for $p = 8$. Still, the matrix can be viewed as comprising disjoint local submatrices that together fit exactly in the space of the original matrix; this is called the Mondriaan structure. Figure 3 illustrates this structure by showing a local view of a partitioning for $p = 8$. Here, the first split is vertical, the following split of the left part is also vertical, and that of the right part horizontal. The direction of the remaining splits can be read from the picture. In each split, the current submatrix is bipartitioned by permuting some rows or columns to one side, and the remainder to the other side. If a row or column is empty,

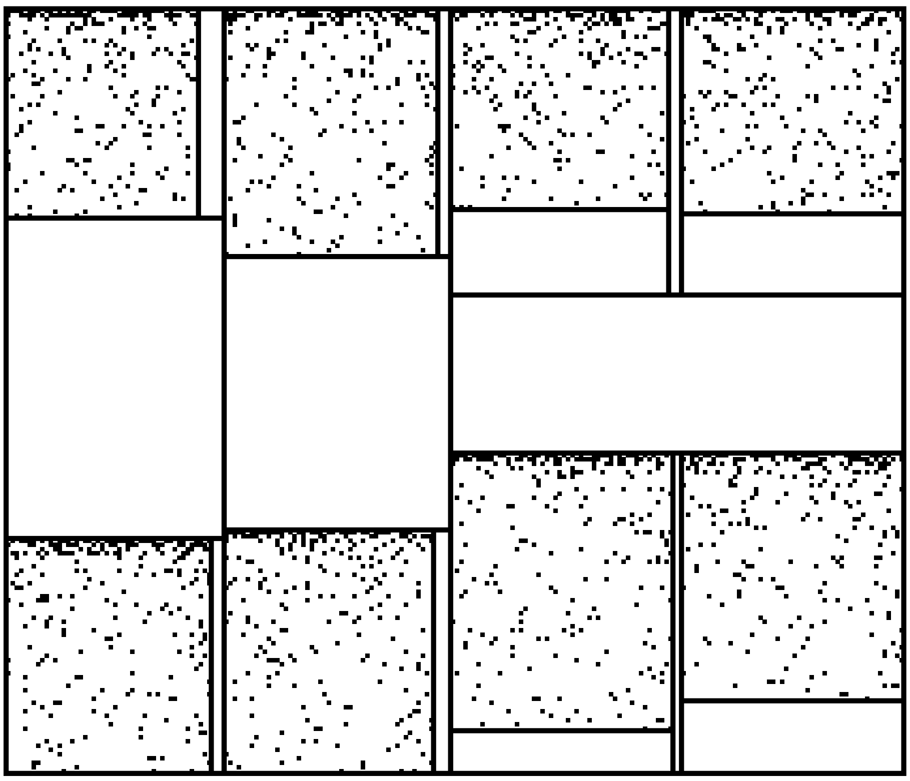

Figure 3. Matrix `mpqs30` from Fig. 2 now partitioned for $p = 8$ processors by using the Mondriaan package. The imbalance allowed is $\epsilon = 3\%$. The matrix is shown in local view, with each nonempty block representing the nonzeros assigned to one processor. The empty blocks are created by the partitioning; they represent row parts or column parts without nonzeros, and these are removed before splitting further.

it is removed and put in a block in the middle. Large empty blocks in the result mean successful partitioning with low communication volume. An arbitrary partitioning of the nonzeros of the matrix will in general not have a Mondriaan structure. For an extensive discussion, see [2, Ch. 4].

4.3. Vector partitioning

A vector can be partitioned by algorithms that try to balance the communication load of the sparse matrix–vector multiplication. Such a partitioning is incorporated in the Mondriaan package, and an improved version is described in [3]. (An alternative method for vector partitioning, which tries to minimise the total number of messages sent, is presented in [20]; this metric is important for a message-passing model of communication.) Note that in the block Lanczos algorithm we have two types of vectors: those of length n_1 and those of length n_2. The two types can be partitioned independently, taking the result of the preceding matrix partitioning into account. This independence allows the improvements of [3] to be applied. For instance, if the number of processors owning a matrix column is at most two, the input vector of length n_2 can be partitioned optimally. This will yield the best possible communication balance given the matrix partitioning. (Note that this balance may still not be perfect.) Figure 2 shows the partitioned input vector of length 210 above the matrix and the partitioned output vector of length 179 to the left of the matrix.

4.4. Dense vector inner-product computation

An inner product of two dense vectors is computed on lines 4, 8, 10, 16. It is easiest to do this if all vectors of the same length have the same distribution. The vector partitioning in Mondriaan does not take the number of components assigned to each processor into account, although in practice this number is not too badly balanced among the processors. A possible extension would be to perform the vector partitioning for multiple objectives, including balancing the inner-product computation.

4.5. AXPY operation

The AXPY is a well-known level 1 operation [14] from the Basic Linear Algebra Subprograms (BLAS). In iterative linear system solvers, it has the form $\mathbf{y} := \alpha\mathbf{x} + \mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors and α is a scalar. Its double-precision version is called DAXPY. In Algorithm 1, the AXPY operation occurs on lines 8 and 14. For instance, on line 8, we can view the multiplication of the $n_2 \times N$ bit matrix V_i by an $N \times N$ matrix as the multiplication of an integer vector of length n_2 by a small object, analogous to a scalar. An AXPY is carried out in parallel by replicating the scalar so that every processor has a copy and letting each processor multiply the local part of the vector by the scalar.

4.6. Global-local indexing mechanism

A mechanism to connect global and local indices is needed at the start of the block Lanczos algorithm. After the matrix and vectors have been distributed, the processors know which matrix elements and vector components they own, but they do not know where to obtain the components of the input vector they need in the matrix–vector multiplication, or where to send contributions for output vectors. The solution to this problem is that the global address of every vector component is first stored at a location that can be inspected by all processors. This location is called the *notice board* in BSPedupack [2], or the *data directory* in Zoltan [9], which provides many additional services besides partitioning. For example, the address of component x_j with global index j of a vector $\mathbf{x}$ can be found at processor $j \bmod p$ at local index $\lfloor j/p \rfloor$, where p is the number of processors; thus, the addresses are distributed cyclically. After the address has been retrieved at the start of the block Lanczos algorithm, the current value of vector component x_j can be obtained from that address each time it is needed.

4.7. Remainder

All remaining parts of Algorithm 1 are less important. Matrices of size $N \times N$ are small (only N integers) and can easily be communicated and replicated. For instance, the computations of lines 7, 11, 12, 13 are carried out redundantly by every processor, i.e., in a replicated fashion, and the same holds for the multiplications of $N \times N$ matrices occurring on lines 8 and 10.

Algorithm 2 uses the same high-level components as Algorithm 1 and some additional computations. On line 3, the matrix Y is subtracted from the matrix X_0. This operation is purely local, since these bit matrices are stored in the same way, namely as an integer vector of length n_2. On line 4, two sparse matrix–vector multiplications are carried out. On line 5, the Gaussian elimination for finding the nullspace is performed in parallel by converting the system to row echelon form, using an algorithm with complete pivoting based on a parallel algorithm for LU decomposition with partial pivoting and a row distribution (see [2, Ch. 2]). Here, each matrix row is actually stored in two integers. On line 6, the dense $n_2 \times 2N$ matrix Z is multiplied by the $2N \times k$ matrix U, with $k \leq 2N$. This can be done by viewing U as a $2N \times 2N$ matrix (padded with $2N - k$ zero columns), splitting it into four submatrices of size $N \times N$ and applying four AXPY operations and two vector additions.

5. Cost analysis

To analyse the cost of our algorithm, we first consider the costs of the separate high-level components. Following the bulk synchronous (BSP) model [21], we can express the cost of a component in the form

$$T = a + bg + cl, \tag{14}$$

where a represents the computation time, b the maximum number of data words (of N bits) a processor has to send or receive, and c the number of global synchronisations required. Usually, computing times are measured in the time of a floating-point operation (flop). Since we are interested in integer operations (iop) here, we take the iop as our unit of time. Furthermore, g and l are machine-dependent parameters representing the communication and synchronisation characteristics of the computer, respectively. The value of g is the average time needed to send one data word into the communication network, or to receive one word, in a situation of continuous network traffic. The value of l is the time of a global synchronisation.

The most expensive component is the sparse matrix-vector multiplication. A very crude approximation (see [2, Section 4.4]) would be

$$T_{\text{Matvec}} \approx \frac{nz(C)}{p} + \frac{n_1 + n_2}{\sqrt{p}} g + 2l, \tag{15}$$

where we assume that the nonzeros of the matrix C are well spread over the processors and the matrix distribution is two-dimensional and square, so that in particular each matrix row and column is assigned to $\sqrt{p}$ processors. Phase (ii) of the matrix-vector multiplication requires an exclusive OR (XOR) operation on a pair of integers for every nonzero in C that is stored locally on a processor, giving a term $nz(C)/p$. This is because in the sequential computation of $C \circledast V$, for every $c_{ij} = 1$, row i of the current matrix CV has to be XOR-ed with row j of V. Phases (i) and (iii) are communication phases which require a global synchronisation at the end, hence the term $2l$. In the worst case, the n_2/p local components of the vector V each have to be sent to the other $\sqrt{p} - 1$ processors containing a matrix column. In phase (iv), the integers received are added into the current vector CV by a bitwise AND operation. The cost of this phase is much less than that of the preceding communication of the data. This cost analysis of the matrix-vector multiplication is just an estimate based on several strong assumptions. In practice, the true cost will be determined by the success of the matrix and vector partitioners. Good partitioners will limit the communication cost bg while balancing the computations. The partitioning shown in Fig. 2 for $p = 4$ has cost $491 + 62g + l$; here, only phases (i) and (ii) are needed. The partitioning shown in Fig. 3 for $p = 8$ has cost $245 + 66g + 2l$. The sequential cost is 1916.

The second largest cost contribution is that of the inner product computations, which are of the form $U = V^T * W$, where V, W are integer vectors of length $n = n_1$ or $n = n_2$, representing bit matrices of size $n \times N$. A sequential algorithm for this operation is given by Algorithm 3. In the parallel version, every processor computes its own $N \times N$ matrix by using Algorithm 3 for those indices k that it owns. This costs nN/p integer XOR operations, if the computation load is balanced. Note that the zero/nonzero pattern of a vector V changes from iteration to iteration in the block Lanczos algorithm, and hence the balance fluctuates. This cannot be solved by better static partitioning of the matrix C or the vectors involved. The resulting p bit matrices are summed to give the final result. Each bit matrix is stored as N integers. These can be sent to all processors, and added redundantly by XOR operations in time $pN + (p-1)Ng + l$. (For large N, this can be further reduced by a two-phase approach to about $N + 2Ng + 2l$, see [2, Ch. 2]). The total time is

$$T_{\text{Inprod}} \leq \frac{nN}{p} + pN + (p-1)Ng + l. \tag{16}$$

Algorithm 3 Dense multiplication of bit matrices.

```
Input: matrices $V^{n\times N}$, $W^{n\times N}$.
Output: matrix $U^{N\times N} = V^T * W$.
$U = 0$
**for** $i = 0, \ldots, N-1$ **do**
  **for** $k = 0, \ldots, n-1$ **do**
    **if** $v_{ki} = 1$ **then**
      **for** $j = 0, \ldots, N-1$ **do**
        $u_{ij} = u_{ij}$ XOR $w_{kj}$
```

The computation cost of an AXPY operation is the same as that of an inner-product computation, and it is carried out in a similar way. No communication is necessary, however, because we assume the $N \times N$ matrix involved is already available locally. (It is produced by other redundant computations.) Thus,

$$T_{\text{AXPY}} \leq \frac{nN}{p}. \tag{17}$$

The remaining computations are small. The cost of a multiplication of two $N \times N$ matrices is

$$T_{\text{Scalar}} = N^2. \tag{18}$$

Note that these N^2 integer operations involve N^3 bit operations. We call this multiplication a scalar operation, because we view the small matrices involved as a scalar object. The cost of the function generateWS is at most $3N^2$, since it involves a complete sweep through an $N \times 2N$ system and a multiplication $S_i * S_i^T$.

The total cost of our computation is mainly determined by the cost of the main loop of the block Lanczos algorithm, i.e. lines 7–17 of Algorithm 1. Thus, the total cost of one iteration is

$$T_{\text{iter}} \approx 3T_{\text{Matvec}} + 3T_{\text{Inprod}} + 5T_{\text{AXPY}} + 12T_{\text{Scalar}}. \tag{19}$$

This cost is dominated by the cost of the three sparse matrix–vector multiplications.

6. Numerical experiments

We performed numerical experiments on up to 16 processors of the Silicon Graphics Origin 3800 at the computing centre SARA in Amsterdam. We used two matrices, `c82` and `c98a`, produced by the MPQS method during the factorisation of composite integers with 82 and 98 decimal digits, respectively. For problems of this size, MPQS is faster than NFS. The matrices originating in MPQS are similar to those of NFS and representative of the wider class of sieving matrices. The properties of these matrices are given in Table 2.

Name	n_1	n_2	$nz(C)$
`c82`	16307	16338	507716
`c98a`	56243	56274	2075889

Table 2
The properties of the two test matrices: size $n_1 \times n_2$ and number of nonzeros $nz(C)$.

We implemented the parallel block Lanczos algorithm using the high-level components described in Section 4. We partitioned the two test matrices and the corresponding vectors using the Mondriaan package [22] version 1. The execution times of the parallel program with $p = 1$ for `c82` and `c98a` are 80 s and 1195 s, respectively. We only have a parallel version of the program available, so we cannot use a sequential version to compare with. Thus, there will be some overhead in our reference version, which is the parallel program run on one processor. The overhead mainly consists of global-local index transformations and unnecessary calls to the synchronisation mechanism. This overhead is expected to be small, because the index transformations are carried out in a preprocessing step, and thus are removed from the main loop of the computation. Furthermore, the main loop contains only a few synchronisations. For $p = 1$, all communications reduce to memory copies.

The relative speedup of the parallel program compared to the $p = 1$ case is given in Figure 4. A speedup of 8 is obtained on 16 processors for `c82`, and a speedup of 7 for `c98a`. Note that we achieve a higher speedup on the smaller problem, which is unusual, and which we find hard to explain. Most likely, cache effects play a role here. (We have partially optimised our implementation to make it cache friendly.) The speedups achieved are reasonable, but not optimal. One reason for this is that the vector partitioning should be improved. The current tests used the vector partitioning of Mondriaan version 1; for version 2, we expect an improvement.

Tables 3 and 4 show that the block Lanczos algorithm is indeed the most time-consuming part of the whole computation, as predicted by the cost analysis in Section 5. The execution time grows considerably with problem size: the larger problem `c98a` has 3.5 times more rows and columns than the smaller problem `c82`, and 4 times more nonzeros, but the computation takes 15 times longer. The time of the input phase is negligible, and it remains constant with an increase in p. This is because the input phase is essentially sequential: the nonzeros and the identity of their owner (as determined by the matrix partitioning) are read by one processor from the input file and then spread over the processors. The postprocessing is also negligible, and its time decreases with an increase in p. The scaling behaviour of the postprocessing is similar to that of the block Lanczos algorithm; both are dominated by sparse matrix–vector multiplications.

We can try to understand the scaling behaviour of the sparse matrix–vector multiplication by relating the theoretical BSP cost with experimental timings. For the smaller problem, `c82`, and for $p = 16$, the BSP cost of a partitioning by the Mondriaan package (version 1.02) run with $\epsilon = 3\%$ is $32,683 + 8298g + 2l$. This value is quite close to the approximation $31,732 + 8161g + 2l$ obtained by (15). This means that partitioning remains a hard problem for this matrix, since it does not perform much better than a fixed two-dimensional, square matrix distribution. This can be improved if we would manage to balance the communication better; here, more than a factor of two can in principle still be gained.

Published parameter values of the parallel computer used are $g = 122$ and $l = 93488$, see [2, Table 3.3]. These benchmark values, however, are for nonoptimised communication, whereas our implementation has been optimised. Furthermore, the sequential computing rate is for dense matrix

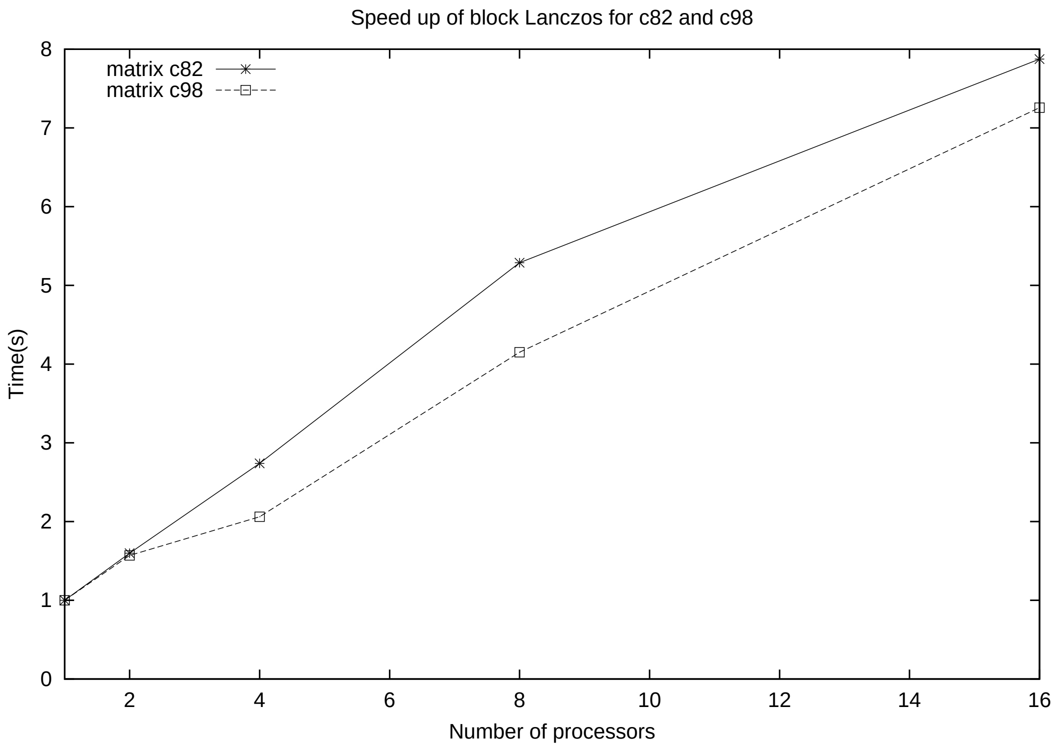

Figure 4. Speedup of parallel block Lanczos algorithm for test matrices `c82` and `c98a`.

computations; for sparse matrix computations the rate will be slower. As a final caveat, the sequential rate has been measured for floating-point operations, whereas here we perform bit-wise operations on integers. Most likely, the measured values are too high. Note that the speedup of about 8 achieved implies that around 50% of the runtime is spent on communication and synchronisation, and 50% on computation (assuming that the load balance is good). For a very precise prediction, the values of g and l must be measured for the application at hand, which we did not do. Still, using our cost model we can get a first indication of parallel runtime and the relative cost of computation, communication, and synchronisation, and their scaling behaviour.

7. Conclusions and future work

We have studied an application in the field of cryptology, the solution of sparse linear systems in the binary field GF(2). We have identified important high-level components for this application and discussed their parallel aspects. This application has some particular characteristics (e.g. the computations modulo 2), but otherwise it stands for a much larger class of applications such as iterative methods for the solution of linear systems and eigensystems. The identified high-level components are important for this whole class.

We have obtained reasonable speedups for our test matrices on our parallel computer. Further improvements should be possible. Here, it is possible to benefit directly from ongoing projects for better matrix and vector partitioners such as Mondriaan [3,22], PaToH [4,20], and the parallel partitioners Par$k$way [19], and Zoltan [8,9].

An issue that also emerged from this application is that we cannot balance the computational load completely by preprocessing to find good matrix and vector partitionings. If we make use of the

p	Input	Lanczos	PP	Total
1	1.15	78.27	0.47	79.90
2	1.12	48.98	0.25	50.36
4	1.13	28.57	0.15	29.85
8	1.15	14.80	0.08	16.02
16	1.30	9.94	0.07	11.31

Table 3
Time (in s) of input phase, block Lanczos algorithm, postprocessing (PP), and total run time for the matrix `c82`. The time is the average over three runs.

p	Input	Lanczos	PP	Total
1	4.1	1186.4	4.0	1194.5
2	4.0	755.8	1.9	761.7
4	3.9	575.5	0.6	580.0
8	4.0	285.8	0.5	290.3
16	4.1	163.5	0.2	167.8

Table 4
Time (in s) of input phase, block Lanczos algorithm, postprocessing (PP), and total run time for the matrix `c98a`. The time is the average over three runs.

current bit pattern in the vectors and in the small $N \times N$ matrices (with $N = 32$) to avoid certain unnecessary operations (cf. the **if**-statement in Algorithm 3), we save work but we also introduce a dependence of the work load on the current state. This may well be the most important source of load imbalance. It is a challenge to find a dynamic procedure to mitigate this effect.

Much research is carried out these days on higher-level tools for parallelisation. The high-level components identified here could provide focus for these efforts. If the tools would help in developing efficient and flexible components for the block Lanczos algorithm, this would have an impact on a wide range of applications.

Acknowledgements

We thank Richard Brent for providing the test matrices used in this paper and for many valuable suggestions. We thank Fatima Abu Salem for interesting and helpful discussions on integer factorisation. We also thank Herman te Riele for useful comments on the initial version of this paper. Furthermore, we thank the anonymous referees whose comments have greatly helped in improving the exposition in this paper. Part of the research has been funded by the Dutch BSIK/BRICKS MSV1-2 project. Computer time has been partially funded by the Dutch National Computer Facilities foundation (NCF). Our computer program is available upon request.

References

[1] Friedrich Bahr, M. Böhm, Jens Franke, and Thorsten Kleinjung. Factorisation of RSA-200. Announcement, http://www.loria.fr/ zimmerma/records/rsa200, May 9, 2005.

[2] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI.* Oxford University Press, Oxford, UK, March 2004.

[3] Rob H. Bisseling and Wouter Meesen. Communication balancing in parallel sparse matrix-vector multiplication. *Electronic Transactions on Numerical Analysis*, 21:47–65, 2005. Special Issue on Combinatorial Scientific Computing.

[4] Ümit V. Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

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

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA and McGraw-Hill, New York, second edition, 2001.

[7] Richard Crandall and Carl Pomerance. *Prime numbers: a computational perspective*. Springer-Verlag, New York, 2001.

[8] K. D. Devine, E. G. Boman, R.T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings IEEE International Parallel and Distributed Processing Symposium 2006*. IEEE Press, 2006.

[9] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, March/April 2002.

[10] Charles M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th IEEE Design Automation Conference*, pages 175–181. IEEE Press, Los Alamitos, CA, 1982.

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

[12] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 343–348. ACM Press, New York, 1999.

[13] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[14] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[15] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard. The number field sieve. In *Proceedings 22nd Annual ACM Symposium on the Theory of Computing*, pages 564–572, 1990.

[16] Peter L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In *Proceedings EUROCRYPT'95*, volume 921 of *Lecture Notes in Computer Science*, pages 106–120. Springer-Verlag, Berlin, 1995.

[17] Peter L. Montgomery. Parallel block Lanczos. In *Proceedings RSA-2000*, 2000.

[18] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[19] A. Trifunovic and W.J. Knottenbelt. Parkway 2.0: A parallel multilevel hypergraph partitioning tool. In *Proceedings 19th International Symposium on Computer and Information Sciences (ISCIS 2004)*, volume 3280 of *Lecture Notes in Computer Science*, pages 789–800. Springer-Verlag, Berlin, October 2004.

[20] Bora Uçar and Cevdet Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.

[21] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[22] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel

sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.