Distributed Matrix-Free Solution of Large Sparse Linear Systems over Finite Fields*

E. KALTOFEN[1] and A. LOBO[2]

[1]Department of Mathematics,
North Carolina State University
Raleigh, North Carolina 27695-8205, (USA)
email: kaltofen@math.ncsu.edu

[2]Department of Computer & Information Sciences,
The University of Delaware
Newark, Delaware 19716-2586,(USA)
email: alobo@cis.udel.edu

May 30, 1997

Abstract

We describe a coarse-grain parallel approach for the homogeneous solution of of linear systems. Our solutions are symbolic, *i.e.*, exact rather than numerical approximations. We have performed an outer loop parallelization that works well in conjunction with a black box abstraction for the coefficient matrix. Our implementation can be run on a network cluster of UNIX workstations as well as on an SP-2 multiprocessor. Task distribution and management are effected through MPI and other packages. Fault tolerance, checkpointing and recovery are incorporated. Detailed timings are presented for experiments with systems that arise in RSA challenge integer factoring efforts. For example, we can solve a $252,222 \times 252,222$ system with about 11.04 million non-zero entries over the Galois field with 2 elements using 4 processors of an SP-2 multiprocessor, in about 26.5 hours CPU time.

*This material is based on work supported in part by the National Science Foundation under Grant No. CCR-9319776. An extended abstract appeared in the proceedings of *High Performance Computing 1996*, a conference sponsored by the Society for Computer Simulation, in April 1996

1 Introduction

The problem of solving large, unstructured, linear systems over finite fields has important implications for the security of public-key encryption algorithms. The RSA scheme [18] is presently impregnable to attack due to the difficulty of finding the prime factors of the public key which is an integer of more than 150 decimal digits. Modern sieve-based integer factoring algorithms generate linear systems containing over 500,000 equations and variables over the finite field of two elements. Usually no more than 0.1 percent of the entries in the coefficient matrix are non-zero.

The probability of successfully breaking the integer key is $1 - 1/2^r$ where r is the number of distinct solutions to a system of linear equations. A candidate solution is useless if it is not exact and speed may be sacrificed for exactitude. Thus the equations are solved in the context of symbolic, rather than numerical, computation.

While coarse-grain parallelism has proven successful [13, 6] for generating the equations, solving those systems has been difficult partly due to the sequential nature of the workhorse Structured Gaussian Elimination [16] algorithm which requires cubic time (in the dimension of the matrix) and does not preserve sparsity. Indeed, the row and column operations performed in the course of executing the algorithm can trigger a dramatic increase in the number of non-zero entries of the matrix, which in turn can overwhelm the storage capacity of most computers.

An alternative approach is to employ iterative matrix-free methods that use the black box model for the coefficient matrix and require a linear number of matrix times vector products plus a quadratic amount of extra arithmetic in the coefficient field, both quantities measured in the dimension of the matrix. Examples of methods in this category are the counterparts, in the exact-arithmetic context of symbolic computation, of the conjugate gradient algorithm [12], Lanczos algorithm [4] and Wiedemann's (1986) coordinate recurrence algorithm [21] which finds linear relations in Krylov subspaces [11].

The black box methods do not depend upon the structural properties of the matrix, in contrast to Structured Gaussian Elimination, and the property of sparsity is maintained. Sparsity is important only in the physical context of a computer implementation. The methods are clearly parallelizable when considering the steps inside the outer loop, *i.e.*, within the matrix times vector product.

A far more difficult task is the parallelization of the outer loop. Wiedemann's original algorithm for solving a system $Bw = 0$ for a system of N linear equations over the finite field K requires no more than $3N$ multiplications of the coefficient matrix B by vectors, plus $O(N^2 \log N)$ arithmetic operations in the field. The method is Las Vegas randomized, *i.e.*, it never gives an incorrect answer upon termination, but it might sometimes fail to provide any answer. Randomization compensates for the loss of speed inherent to the pursuit of a symbolic solution.

An important substep is the computation of the sequence of field elements

$$a^{(i)} = u^{\mathrm{tr}} B^i v \in \mathsf{K} \quad \text{for } 0 \le i \le 2N-1,$$

where u and v are vectors with random entries from K. A key idea is that this sequence is generated by a linear recurrence that, with high probability, corresponds to the minimum polynomial of B and which can be computed by the Berlekamp/Massey algorithm [14]. Evaluation of the minimum polynomial yields the solution, if one exists. Wiedemann's original algorithm is entirely sequential.

Coppersmith simultaneously used m vectors U for u and n vectors V for v. The sequence becomes one of $m \times n$ matrices.

$$\mathbf{a}^{(i)} = U^{\mathrm{tr}} B^i V \in M_{m\times n}(\mathsf{K}).$$

He then generalized the Berlekamp/Massey algorithm to find the linear recurrence with vector coefficients, that generates this sequence of small rectangular matrices. Clearly the $\mathbf{a}^{(i)}$ can be computed independently and in parallel.

Kaltofen (1995) showed that the number of terms $\mathbf{a}^{(i)}$ to be computed reduces from $2N$ down to $N/m + N/n + 2n/m + 1$, and overall there are only $3N/n$ applications of B to a vector (assuming $m \ge n$), instead of $3N$. Kaltofen also gave an analysis of the running time complexity. Thus Coppersmith's block Wiedemann algorithm, when implemented in a parallel setting, exhibits a speedup proportionate to n and performs much faster than its sequential counterpart.

In the this paper we will give an overview of the block Wiedemann method followed by a description of our software package, WLSS2, which runs on a network cluster of SPARC workstations or under the SPMD model on the IBM SP-2 multiprocessor using MPI as a scheduler. Coarse grain parallelism is used for simultaneously finding multiple solution vectors from the kernel of the coefficient matrix. Instead of exploding the matrix and mapping its contents onto individual processors in a mesh, we have kept the matrix whole by treating it as a black box. We have performed an outer loop parallelization that works well in conjunction with this black box approach.

2 Linearly Generated Sequences and Wiedemann's algorithm

Let W be a vector space over an abstract field K. The infinite sequence $\{s_i\}_{i\ge 0}$ with s_i in W is said to be linearly generated over K if there exist scalars $c_0, c_1, \ldots, c_d, \ldots, c_l$ in K with $c_d \ne 0$ for some $0 \le d \le l$, such that for every $j \ge 0$,

$$\sum_{i=0}^{l} c_i s_{j+i} = 0. \tag{1}$$

We say that $c(x) = c_0 + c_1 x + \cdots + c_l x^l$ is a generating polynomial, or annihilating polynomial, for $\{s_i\}_{i\geq 0}$. The set of all generating polynomials together with 0 forms an ideal in $\mathsf{K}[x]$, the ring of univariate polynomials over K, and the generator of this ideal is a unique, monic, non-zero polynomial of minimal degree which we call the minimum polynomial (or minpoly) of $\{s_i\}_{i\geq 0}$.

Suppose V is another vector space and let $\alpha : \mathsf{W} \longrightarrow \mathsf{V}, s_i \overset{\alpha}{\longmapsto} t_i$ be a linear map. The sequence $\{t_i\}_{i\geq 0}$ is also linearly generated and its minimum polynomial divides that of $\{s_i\}_{i\geq 0}$. The reader is referred to [8] for further details.

2.1 Wiedemann's Algorithm

Let B be an $N \times N$ matrix over an abstract field K. Typically the entries lie in some finite field $\mathsf{K} = \mathbb{F}_q$ the finite field containing q elements. The problem is to solve the homogeneous linear system

$$Bw = 0 \tag{2}$$

If B is non-singular, the system $By = z$ where z is a vector in the column space of B, can be converted to a homogeneous system by augmenting the columns of B with z and adding a zero row. It is assumed that B is stored in ω memory locations and that the cost of applying B to a row or column vector is at most 2ω arithmetic operations. The actual entries of B are not manipulated. This is the black box model for the coefficient matrix and it is shown in figure (1).

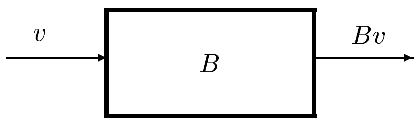

Figure 1: Black Box Model for a Coefficient Matrix

Wiedemann's alogorithm [21] is based on the fact that when a square matrix is repeatedly applied to a vector, the resulting vector sequence is linear recursive. Let v be a column N-vector over K. Since the space $S = \{B^i v\}_{i\geq 0}$, where $B^0 = I$ the identity matrix, is N-dimensional, there is an integer $1 \leq l \leq N$ such that $v, \ldots, B^{l-1}v$ are linearly independant and B^l is a linear combination of these vectors with coefficients in K *i.e.*

$$B^l v = -c_{l-1} B^{l-1} v - \cdots - c_0 v \tag{3}$$

The monic polynomial $f^{B,v}(\lambda) = \lambda^l + c_{l-1}\lambda^{l-1} + \cdots + c_0$ has the least degree among all polynomials for which (2) holds. It is called the minimum polynomial of v, and

$$\begin{aligned} f^{B,v}(B)v &= 0 \\ B^{\delta}\left(B^{l-\delta}v + c_{l-1}B^{l-\delta-1}v + \cdots + c_{\delta}v\right) &= 0 \end{aligned} \tag{4}$$

where $\delta \geq 0$ is the minimum index for which c_δ is non-zero. If B is non-singular, then $\delta = 0$.

Suppose $f^{B,v}$ has been found, let $\widehat{w}$ denote the vector in parenthesis in (4) and suppose $\widehat{w} \neq 0$. Then for some integer t, with $1 \leq t \leq l$, $B^t\widehat{w} = 0$ but

$$w = B^{t-1}\widehat{w} \neq 0 \tag{5}$$

and so $Bw = 0$.

Clearly $\dim S \leq N$ so the solution to (2) can be computed within $2N(\omega + 1)$ operations with a small constant times N space requirement beyond that needed to store B.

Suppose u is a column vector over K, then the sequence $\{u^{\mathrm{tr}}B^iv\}_{i\geq 0}$ is derived from $\{B^iv\}_{i\geq 0}$ by a linear map and satisfies the linear recurrence associated with $f^{B,v}$. The minimum polynomial of $\{u^{\mathrm{tr}}B^iv\}_{i\geq 0}$ denoted by $f_u^{B,v}$, divides $f_u^{B,v}$. Each term of $u^{\mathrm{tr}}B^iv$ is an element of K and the Berlekamp-Massey algorithm [2, 14] can be employed to find $f_u^{B,v}$ from the first $2N$ terms of $\{u^{\mathrm{tr}}B^iv\}_{i\geq 0}$ in time $O(N^2)$ field operations. Wiedemann's algorithm is shown in figure (2.1). The components of vectors u and v are chosen uniformly at random from the elements of K.

2.1.1 Probabilistic Justification of Wiedemann's Algorithm

It may sometimes happen that algSW fails to give a vector in the kernel of B. In such an instance, $f_u^{B,v} \neq f_u^{B,v}$ owing to a "bad" projection and (5) cannot be satisfied. Wiedemann derives (§6 in [21]) an estimate for the probability that the two minimum polynomials are the same.

Theorem 1 *Let $d = \deg(f^{B,v})$ and let W be the linear space of polynomials of degree less than d in $\mathsf{K}[\lambda]$. There exists a surjective linear map $\alpha : M_{N\times 1}(\mathsf{K}) \longrightarrow \mathsf{W}$ such that*

$$\begin{aligned} \forall u \;\in\;& M_{N\times 1}(\mathsf{K}): \\ & f_u^{B,v} = f^{B,v} \iff \mathsf{GCD}(f^{B,v}, \alpha(u)) = 1. \end{aligned}$$

□

Thus the probability that $f_u^{B,v} = f^{B,v}$ for a randomly selected column vector u is the probability that a randomly selected polynomial of degree less than N is relatively prime to $f^{B,v}$. In $\mathbb{F}_q$, the finite field of q elements, Wiedemann shows that

$$\Pr\left(f_u^{B,v} = f^{B,v}\right) \geq \frac{1}{6\max\{\lceil \log_q(\deg f^{B,v})\rceil, 1\}} \tag{6}$$

Kaltofen and Pan ([10] §2) prove the following alternative:

Input		$B \in M_{N\times N}(\mathsf{K})$
Output		$w \in M_{N\times 1}(\mathsf{K})$ such that $Bw = 0$
Body	s0	Select random $u, z \in M_{N,1}(\mathsf{K})$ $L \leftarrow 2N$ $v \leftarrow Bz.$
	s1	Compute $\{u^{\mathrm{tr}} B^i v\}_{i=0,\ldots,L}$
	s2	Apply the Berlekamp-Massey algorithm and find l, $c_0, c_1, \ldots, c_l \neq 0$, $c_i \in \mathsf{K}$, such that for $i = 0, 1, \ldots, L - l - 1$ $u^{\mathrm{tr}} \left(B^i c_0 + \cdots + B^{i+l} c_l\right) v = 0$
	s3	$\delta \leftarrow \min_{i\geq 0}\{c_\delta \neq 0\}$ $\widehat{w} \leftarrow B^{l-\delta} z c_l + \cdots + Bzc_{\delta+1} + zc_\delta$ $t \leftarrow \min_{1\leq j\leq \delta+1}\{B^j \widehat{w} = 0\}$
Return		$w \leftarrow B^{t-1}\widehat{w}$ or failure

Figure 2: Wiedemann's algorithm (algSW)

Theorem 2 *Let $\mathcal{S} \subseteq \mathsf{K}$. Randomly and uniformly select $u \in M_{N\times 1}(\mathcal{S})$. Then*

$$Pr\left(f_u^{B,v} = f^{B,v}\right) \quad \geq \quad 1 - \frac{\deg(f^{B,v})}{\mathrm{card}(\mathcal{S})}$$

$\square$

Of course, theorem (2) is meaningful only when $\mathrm{card}(\mathcal{S})$ is greater than $\deg(f^{B,v})$. The cardinality can always be increased by making $\mathcal{S}$ a finite extension of K of appropriate degree over K.

3 Coppersmith's block Wiedemann Algorithm

Coppersmith proposed a block version of the Wiedemann algorithm which takes advantage of the ability to perform simultaneous operations on blocks of vectors. His algorithm works with matrices $U \in M_{N\times m}$ and $V \in M_{N\times n}$ in place of u and v. As many as n candidate solutions are found simultaneously on a single invocation. It is not guaranteed that these vectors are non-zero, or different from one another. The cost per solution is decreased in approximate proportion to the amount of blocking *i.e.* the column dimension of V. There are three steps to the algorithm which is shown in figure (3).

Step BW1: *Sequence Generation.* Pick random vectors $U = [u_1|u_2|\cdots|u_m]$, and $Z = [z_1|z_2|\cdots|z_n]$, $u_j, z_i \in M_{N\times 1}$ for all $0 < i \leq m$, $0 < j \leq n$. Compute $V = BZ$ and

$$\mathbf{a}^{(i)} = (U^{\mathrm{tr}} B^i V)^{\mathrm{tr}}, \quad \text{for all } 0 \leq i \leq \frac{N}{m} + \frac{N}{n} + \frac{2n}{m} + 1. \tag{7}$$

The task requires not more than

$$\Big(1 + \frac{n}{m}\Big)N + \frac{2n^2}{m} + 2n \tag{8}$$

black box multiplications $v \longmapsto Bv$ of a column vector in $M_{N\times 1}$ by B as shown in figure (1). Actually, the κ_ν rows of $\mathbf{a}^{(i)}$ can be computed using κ_ν columns of the vector V as a coarse-grain parallel operation, shown in figure (7). The ν^{th} processor gets a copy of the black box for B, and the entire vector U. Another way is to perform the computation $B \cdot (B^{i-1} v_\nu)$ in parallel, as Coppersmith does, for each i. The grain is much finer than before but synchronization might be needed. We will return to this issue in the next section.

Step BW2: *Finding a Linear Generator.* Find a linear generator Ψ of length $D+1$ for the sequence of matrices $\mathbf{a}^{(i)}$.

$$\Psi(\lambda) = \lambda^D + \mathbf{c}_{D-1}\lambda^{D-1} + \ldots + \mathbf{c}_1\lambda + \ldots + \mathbf{c}_0.$$

where $\mathbf{c}_i \in \mathsf{K}^n$ for $i = 0, \ldots, D-1$. All this can be accomplished sequentially with $O(nN^2)$ operations in the coefficient field. A practical parallelization has so far eluded us. We will describe the full details of of this step in a subsequent section.

Step BW3: *Horner-like Evaluation.* This block step involves a Horner-like evaluation of a polynomial Ψ_j for $1 \leq j \leq n$, derived from Ψ whose coefficients are n-dimensional vectors.

$$\Psi_j(\lambda, V) = \lambda^l V\mathbf{c}_{l,j} + \cdots + \lambda^{\delta+1} V\mathbf{c}_{\delta+1,j} + \lambda^\delta V\mathbf{c}_{\delta,j}$$

with $\mathbf{c}_{k,j} = \mathbf{0}$ for all $0 \leq k < \delta \leq l$. Each coefficient $\mathbf{c}_{j,j}$ is the j^{th} column of the coefficient of λ^j in $\Psi(\lambda)$ computed in the previous block step. This step can be performed in a parallel/distributed setting.

With sufficiently high probability, this justified later, Ψ_j yields a valid linear combination of vectors in V_V such that

$$\Psi_j(B, V) = \mathbf{0} \tag{9}$$

within an orthogonality constraint on V_U and V_V. In other words, with high probability, the projections due to U do not introduce any spurious additional linear dependancies.

Setting $\widehat{\Psi}_j(\lambda) = \lambda^{-\delta}\Psi_j(\lambda)$ compute

$$\widehat{w}_j = \widehat{\Psi}_j(B, Z) \tag{10}$$

This task requires no more than $l - \delta$ matrix times vector products plus some additional $O(N^2)$ work to compute the products of the form $Z\mathbf{c}_{j,i}$.

With probability at least $1 - 1/\mathrm{card}(\mathsf{K})$, $\widehat{w}_j \neq \mathbf{0}$. Compute the product $B^t\widehat{w}$ for the smallest integer $1 \leq t \leq \delta + 1$ that yields the zero vector. Finally $w_j = B^{t-1}\widehat{w}_j$ is returned.

With parallelization, the block step of evaluation can yield as many as n individual candidate solution vectors w_j which may not all be different or nonzero.

Input		$B \in M_{N\times N}(\mathsf{K})$
Output		$w \in M_{N\times n}(\mathsf{K})$ such that $Bw = \mathbf{0}$
Body	b0	Choose random $U \in M_{N\times m}(\mathsf{K})$, $Z \in M_{N\times n}(\mathsf{K})$ such that rank $(U^{\mathrm{tr}}AZ) = n$ $L \leftarrow N/m + N/n + 2n/m + 2$ $V_0 \leftarrow BZ$
	b1	Compute $\{(U^{\mathrm{tr}}B^iV)^{\mathrm{tr}}\}_{i=0,\ldots,L}$ by a coarse-grain parallel method
	b2	Compute a linear generator $\Psi^{\mathrm{tr}}(\lambda)$ with $m\times n$ matrix coefficients using algMinp, such that $U^{\mathrm{tr}} \cdot \Psi(B) \cdot V = 0$.
	b3	For each $1 \leq j \leq n$ $\delta_j \leftarrow \max_{i\geq 0}\{\lambda^i \text{ divides } \Psi_j(\lambda)\}$ $\widehat{\Psi}_j(\lambda) \leftarrow \lambda^{-\delta_j}\Psi_j(\lambda)$ $\widehat{w}_j \leftarrow \widehat{\Psi}_j(B) \cdot Z$ $t_j \leftarrow \min_{1\leq i\leq 1+\delta}\{B^i\widehat{w}_j = \mathbf{0}\}$
Return		$w = [w_1 \mid \cdots \mid w_n]$ such that for each $1 \leq j \leq n$, $w_j \leftarrow B^{t-1}\widehat{w}_j$ where $w_j \neq 0$ and $Bw_j = 0$; otherwise set $w_j = 0$

Figure 3: Coppersmith's block Wiedemann algorithm (algBW) Steps b0 and b1 compute a sequence of $n\times m$ matrices. Step b3 computes a generator polynomial with $m \times n$ coefficients. Step b3 evaluates this generator polynomial.

The overall cost for finding n candidate solutions to (2) is within

$$O((1 + \frac{2n}{m})N\omega + (m+n)N^2) \tag{11}$$

field arithmetic operations, where ω, as before, is the cost of applying a black box matrix to a column vector. These figures are supported by the following facts from [4, 9, 20].

Proposition 1 *A linear generator for the sequence $\{U^{\mathrm{tr}}B^iV\}_{i\geq 0}$ of $m\times n$ matrices, $m\geq n$ can be computed from a consideration of the first L terms of the sequence, provided*

$$L \geq N/m + N/n + 2n/m + 2$$

$\square$

Proposition 2 *Let $\widehat{w}$ be calculated by step 3 in algorithm* algBW. *Then*

$$\mathrm{Prob}(\widehat{w}\;\neq\;0) \geq 1 - \frac{1}{\mid \mathrm{Kernel}(B)\mid}$$

$\square$

Unlike the sequential case, a proof that the block algorithm works and an estimate for the probability of success are available in a restricted setting, in which the rank of the matrix B bears a strict relationship to the degree of the minimum polynomial of B, and the cardinality of K is approximately $O(N^2)$.

Theorem 3 *Let K be a finite field, and let $B\in M_{N\times N}(\mathsf{K})$ be a singular matrix whose minimum polynomial f^B has degree $\deg(f^B) = \mathrm{rank}(B)+1$. Suppose that the vector blocks $U\in M_{N\times m}(\mathsf{K})$ and $V\in M_{N\times n}(\mathsf{K})$ are chosen at random and that $\widehat{w}$ is computed as in algorithm* algBW. *Then with probability no less than $1-(2\,\mathrm{rank}(B)+1)/\mathrm{card}(\mathsf{K}) \geq 1-(2N-1)/\mathrm{card}(\mathsf{K})$ we have $\widehat{w}\neq 0$ and $B^{t+1}\widehat{w}=0$ for some integer $t\leq N/n$*

Proof. See [9]. $\square$

Kaltofen's analysis is valid only for fields where $\mathrm{card}(\mathsf{K})$ is greater than approximately $2N$. Villard[20] has a generalized approach that is valid in any field $\mathbb{F}_q$. Letting β denote the number of companion blocks in the Frobenius form of B restricted to its range space, letting π_B denote the minimum polynomial of B, Villard defines functions $\Phi_n(f_B,\beta)$ and $\Theta_n(f,\beta)$ and an integer parameter Δ such that the length of the sequence computed is $L = N/m + N/n + \Delta$, and gives the following theorem.

Theorem 4 *Let B be an $N\times N$ matrix over $\mathsf{K}=\mathbb{F}_q$. The matrices $U\in M_{m,N}(\mathsf{K})$ and $Z\in M_{N,n}(\mathsf{K})$ are chosen at random. Suppose that $m\geq min\{\beta,n\}$, and $K_e = \mid \mathrm{Kernel}(B)\mid$. If w is computed by the block algorithm with shift parameter Δ then $\mathrm{Prob}\{w\neq 0, Bw=0\}$ is greater than*

$$(\Phi_m(f_B, min\{\beta,n\}) - \Theta_m(f_B, min\{\beta,n\}))\,(1-1/K_e)$$

Proof. [20] $\square$

Villard's result effectively lays to rest all concerns about the conditions under which the block Wiedemann algorithm fails. His theorem does not depend upon randomized preconditioning. For large fields, his result can be specialized to Kaltofen's.

4 The Sequential Component

The biggest obstacle faced in designing a block Wiedemann algorithm was the correct generalization of the Berlekamp/Massey Algorithm [14]. We will describe a version due to Coppersmith in light of our experience with the implementation of this and similar algorithms for rational approximation of linear sequences. The algorithm corresponds to step BW2 and is purely sequential. A parallel version working well in practice has so far eluded us.

Given the first L terms of the formal power series $T(z) = \sum_{i=0}^{\infty} T_i z^{-i}$ of $m \times n$ matrices with $T(z)$ assumed rational, the problem is to find an $m \times n$ polynomial matrix $G(z)$ and and $n \times n$ invertible polynomial matrix $H(z)$, $\deg(G) < \deg(H)$ such that the first L terms of the Laurent expansion of $G(z)H(z)^{-1}$ correspond to $\{T_0, \ldots, T_{L-1}\}$ *i.e.*

$$T(z) \bmod z^{-L} = (G(z)H(z)^{-1}) \bmod z^{-L}. \tag{12}$$

In linear control system theory this is the partial minimal realization problem [19]. In the present context, the T_i are the matrices $\left(U^{\mathrm{tr}} B^i V\right)^{\mathrm{tr}}$. The denominator term $H(z)$, corresponds to the sought-after linear generator. The pair (G, H) is unique only upto right multiplication by an $n \times n$ unimodular matrix.

The problem has been solved many ways in the past, with the oldest version being apparently the work of Rissanen [17]. Rissanen's algorithm finds a linear generator of minimum degree.

Competing algorithms have been given by [9] which uses randomization and FFT-based fast polynomial arithmetic and costs $O((m+n)N^2 \log N \log\log N)$ arithmetic operations; the generalized Levinson/Durbin/Trench algorithm [5] for Toeplitz-like matrices [7]; and the asymptotically faster version by [3] or [15]. Using fast polynomial arithmetic the cost is in $O((m+n)^2 N (\log N)^2 \log\log N)$. Another approach computes matrix Padé approximants [1] in $O(mN^2)$ time.

4.0.2 Rational Approximation of a Linearly Generated Matrix Sequence

Coppersmith's algorithm treats the sequence $\{\left(U^{\mathrm{tr}} B^i V\right)^{\mathrm{tr}}\}_{i \geq 0}$ as a matrix polynomial with $n \times m$ matrix coefficients. His method is iterative, requires no randomization, and costs $O((m+n)N^2)$ with classical arithmetic. The algorithm is shown in figure (3).

The key variables are $\Lambda^{(t)} \in M_{2n\times m}(\mathsf{K})[\lambda], \Delta^{(t)} \in M_{2n\times n}(\mathsf{K}), T_t \in M_{2n\times 2n}(\mathsf{K})$. At each iteration $t = 0, 1, 2, \ldots$ the polynomial

$$\Lambda^{(t)}(\lambda) = \begin{bmatrix} \Psi^{(t)}(\lambda) \\ \Phi^{(t)}(\lambda) \end{bmatrix}$$

is computed. When viewed as a polynomial matrix in $M_{2n\times m}(\mathsf{K}[\lambda])$, the first n rows of $\Lambda^{(t)}$ represent "current" trial generator polynomials and the latter n are the "most recent past" generators, as in the classical Berlekamp/Massey algorithm. An integer upper bound $\mathbf{d}_i^{(t)}$ or nominal degree is assigned to the i^{th} row of $\Lambda(\lambda)$. The actual degree of the row could be less due to cancellation in the course of row operations on $\Lambda(\lambda)$. The matrix coefficient $\Delta^{(t)} = \mathrm{coeff}(\lambda^t, \Lambda^{(t)}\boldsymbol{\alpha})$ represents the discrepancy between the actual value of $\mathbf{a}^{(t)}$ and the value generated using the most recent trial generators. In particular, if the upper n rows of $\Delta^{(t)}$ are all zeroes then the trial generator Ψ could be a valid generator for the entire input sequence

Input	$\boldsymbol{\alpha}(\lambda)$ – sequence of $n \times m$ matrices
Output	$\Psi(\lambda)$ – linear generator for $\boldsymbol{\alpha}(\lambda)$ $\mathbf{d}$ – list of nominal degrees of rows of Ψ, Φ
Variables	$\Phi, \Psi \in M_{m\times n}(\mathsf{K}[\lambda])$ – linear generators $T \in M_{2n\times 2n}(\mathsf{K})$ – linear transformation $\Delta \in M_{2n\times m}(\mathsf{K})$ – discrepancy matrix $\mathbf{d} \in \mathsf{K}^{2n}$ – nominal degree list
Body	$\Phi \leftarrow \lambda$ $\Psi \leftarrow 1$ $\mathbf{d}[1, \ldots, n] \leftarrow 1$ for $t = 1, \ldots, \deg(\boldsymbol{\alpha}(\lambda))$ do $\quad \Delta \leftarrow \mathrm{coeff}\left[\lambda^t, \begin{matrix}\Psi(\lambda)\\ \Phi(\lambda)\end{matrix}\ \boldsymbol{\alpha}(\lambda)\right]$ $\quad$ Compute T_t such that $T_t\Delta = \begin{bmatrix} * \\ 0 \end{bmatrix}$ $\begin{bmatrix}\Psi(\lambda)\\ \Phi(\lambda)\end{bmatrix} \leftarrow \begin{bmatrix} I_n & \mathbf{0}_n \\ \mathbf{0}_n & \lambda I_n \end{bmatrix} T_t \begin{bmatrix}\Psi(\lambda)\\ \Phi(\lambda)\end{bmatrix}$ $\quad \mathbf{d}[m+1, \ldots, 2n] \leftarrow 1 + \mathbf{d}[n+1, \ldots, 2n]$ od.
Return	$\Psi, \mathbf{d}$

Figure 4: Algorithm algMinp finds a linear generator with matrix coefficients in step BW2

Two conditions are maintained throughout, namely:

(C1) $\mathrm{coeff}(t, \Lambda_i^{(t)}\alpha) = \mathbf{0}^{\mathrm{tr}}$
$(\forall i, 1 \leq i \leq 2n) \left(\forall j, \mathbf{d}_i^{(t)} \leq j < t\right)$

(C2) $\mathrm{Rank}(\mathrm{Coeff}(t, \Lambda^{(t)}\alpha)) = n$

At start, with $t = 0$ with $\Psi^{(0)}(\lambda) = \Phi^{(0)}(\lambda) = I_n$ and all nominal degrees are initialized to 1, Condition (C1) is vacuous. But clearly (C2) is enforced by the choice for Z and $\mathbf{u}$. Now, by setting $\Phi = \lambda I_n$ and $t = 1$, the remarks on (C1) and (C2) still apply. Of course, in practice there is no need to start at $t = 0$. Proceeding inductively, for $t > 0$ with increments by 1, first $\Delta^{(t)}$ is computed as in step (3) of figure (4). This costs $O(n^2 mt)$ operations. Except for $t = 1$ only the submatrix from the product $\Psi^{(t)}\boldsymbol{\alpha}$ needs to be computed, the remaining submatrix comes from the last n rows of $T_{t-1}\Delta^{(t-1)}$. The next step is to maintain (C2) using the last n rows of $\Delta^{(t)}$ to compute a transform T_t that makes the first n rows of $\Delta^{(t)}$ each zero.

One approach is to triangulate $\Delta^{(t)}$ along the antidiagonal. Briefly, for each $j = 1, 2, \ldots, m$ a pivot row is selected which is non-zero in its j^{th} column. This row is then exchanged with an appropriate row from the lower m rows of $\Delta^{(t)}$ and then used to make zero the entries above it in column j. The pivot row is selected sorting the nominal degrees in ascending order and picking a candidate row of lowest nominal degree. Thus a row of $\Delta^{(t)}$ corresponding to a row of $\Lambda^{(t)}$ is never subtracted from another row of lower nominal degree. We remark that T_t can be computed in no more than $(m + n)^3$ field arithmetic operations. The transformation is applied to all terms of of $\Lambda^{(t)}$ and the nominal degrees of the last n rows of $\Lambda^{(t)}$ are incremented by 1. Thus

$$\Lambda^{(t+1)} = \mathrm{Diag}(\underbrace{1, \ldots, 1}_{n}, \underbrace{\lambda, \ldots, \lambda}_{n}) \cdot T_t \cdot \Lambda^{(t)}.$$

The multiplication using the diagonal matrix can be done implicitly by simply incrementing the last n rows of $\mathbf{d}^{(t)}$. This update step costs $O(n^2 mt)$ operations. Summing the component costs over all $O(N/mn)$ iterations, The entire cost of step BW2 is in $O((m+n)N^2)$ and the coefficients of Ψ are rectangular matrices.

Unfortunately a better, practical, parallel version has eluded us, and so the step of finding a recurrence for the matrix sequence of algBW is purely sequential.

5 Black Box Matrix and Outer-loop Parallelization

Mathematically, a black box matrix is a linear operator that carries a vector into another vector. In the language of object-oriented programming, a black box matrix is an instance of a program class that consists of a private area, containing either an encoded matrix or a private member function that can dynamically generate components of the matrix on demand, and a public area that contains a constructor and an applicator function. The black box matrix implicitly or directly multiplies an input vector by the private matrix using the applicator function. The applicator function is permitted to call the constructor function, if one is supplied, and it has access to private data.

The constructor obtains matrix data from an external source and stores an encoding in the private data area. In our implementation, the data for the matrix is stored in a compressed format within private memory, and the constructor reads that data from a file.

The apply function carries out the matrix times vector product by means of either a private direct access to the static matrix data, or by calls to the private data generator function, if one exists. A schematic black box is depicted in figure (1).

When memory restrictions prohibit the instantiation of the data, or a good way exists to dynamically generate the contents of the matrix, the data generator function approach is preferred. In that case, the constructor is rudimentary and is frequently a vestigeal requirement of the object-oriented language, needed for creating an instance of the black box class.

Figure (5) shows the C language description of the black box for a matrix over the finite field of 2 elements. The data is stored statically as an array of arrays of column indices for those columns containing non-zero data. In this case, there is no need to actually store the data, but if the field contained more than 2 elements, the row lists would be of ordered pairs representing column index and actual data stored. The column addresses are divided into a set of contiguous columns whose addresses are relatively offset by less than 32767 (representable in 15 binary digits), and a second set containing the remaining columns.

The representation conserves physical memory on matrices from integer factoring which, while possessing no special structure, are nevertheless highly dense in the first few hundred rows, and the density gradually falls off to the point where a row might contain as few as three non-zero columns.

The constructor reads the matrix into static memory while the applicator function performs routine matrix times vector multiplication with the important consideration that in $\mathbb{F}_2$, two elements can be added or multiplied using the bitwise *exclusive-or* and *and* functions respectively.

5.1 Parallelizing the outside loop

The use of block vectors and a black box lead naturally to coarse-grain parallelism. The computations for step s1 and s3 in algBW can be distributed over a network of compute nodes. Significantly, the length of sequence needed is reduced, which characterises an outer-loop parallelization of algSW. The expected parallel speedup is roughly proportional to the number of vectors in a block. Moreover the algorithm can be applied to any linear system over any finite field and retains the property of being matrix-free because the black box approach is preferred over the natural tendency to partition the matrix and assign components to processors on a mesh or array, the latter being a form of inner-loop parallelization.

```
/*------------------------------------------- *
 * ----       Class Declaration         ---- *
 *------------------------------------------- */
typedef  struct smatrix
{ short int **sdata;/* offsets within 16 bits */
  long  int **ldata;/* offsets within 32 bits */
  long  int imode;  /* format bits of file    */
  long  *mbase,     /* base address           */
        *scount,    /* count short offset data*/
        *lcount;    /* count of long  absolute*/
  long  Nr, Nc;     /* no. of rows and cols   */
}  SMATRIX;
/*--------------------------------------------*
 * --- Signatures of the member functions --- *
 *------------------------------------------- */
long bbMx_Init(/*void */); /* initialize bbox */
long bbMx_Apply(/* VECT v_out,VECT v_in */);
               /* matrix times vector product */
/-------------------------------------------- *
 * ----        Global Variables         ---- *
 *------------------------------------------- */
char  *bbMx_fname;/* input filename of matrix */
SMATRIX  mbbox;   /* matrix  box  instance    */
long Nsys;        /* dimension = max(Nr,Nc)   */
int  bbMx_initflag;  /* prevents re-init.     */
```

Figure 5: C language definition of Black Box and associated variables for an $N_r \times N_c$ matrix

5.1.1 Preferred Mode of Distribution

Our preferred partitioning scheme was chosen to avoid elaborate synchronization and to minimize the cost of data communication across a network. The entire black box matrix is copied to every active processor, and blocks of vectors are distributed. The overall mechanism is shown in figure (7).

The j^{th} block column of $U^{\text{tr}}B^iV$ is computed by a coarse grained parallel operation in which vector U, and the j^{th} column block v_j of V having width κ_j, is sent along with a copy of the black box to each of several processors on a network. Starting with $v_j^{(0)} = v_j$, the j^{th} processor, computes $v_j^{(k+1)} = Bv_j^{(k)}$, by a black box application, followed by an inner product $U^{\text{tr}}v_j^{(k+1)}$, which yields κ_j columns of $\mathbf{a}^{(k+1)}$. All this is depicted in figure (6)

The output from a processor depends only on its own most recent output, and not on the output of any other processor sibling. Thus there is no need

t_1: Proc(j) computes $v_j^{(k+1)} = By_j^{(k)}$
t_2: Proc (j) computes $U^{\mathrm{tr}} v_j^{(k+1)}$ and writes to file

Figure 6: Black Box Matrix with Distributed Vectors

for interprocessor communication and synchronization is uncomplicated. The processors can be co-ordinated by barrier synchronization or a busy wait, both at minimal cost. Furthermore the cost of sending large volumes of data over a slow link is avoided.

At regular intervals each processor updates a checkpoint of its state, particularly of a current value of $v_j^{(k+1)}$. The writing of the output $U^{\mathrm{tr}} v_j^{(k+1)}$ to a file immediately upon generation means that all past output is saved. So faults as well as recovery are localized.

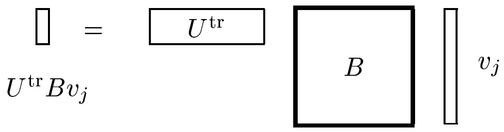

Figure 7: Distributing the vectors: preferred mode for distributed computation of vector sequence

Other strategies exist for multiprocessor architectures with shared memory, but they might require synchronization. The strategies lead to finer grainsize than in our preferred strategy and incorporate a distributed black box, as shown in figure (10) and figure (8). They are are not suited for networks of workstations because of the synchronization issue and the frequent transfer of fairly high volumes of data.

5.1.2 Transversely Segmented Matrix Black Box

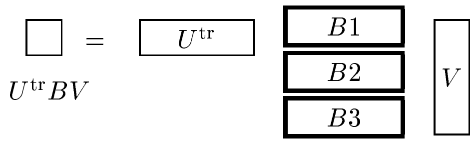

Figure 8: Distributing the black box with transverse partitioning: alternative strategy

With transverse partitioning the j^{th} processor is able to call a black box representing a block of ρ_j rows of B, which denoted by B_j. Applied to sequence generation, the strategy is depicted in figure (8). The j^{th} processor computes $y_j^{(i)} = B_j y^{(i-1)}$ and must have access to $y^{(i-1)}$ which is assembled from all $y_j^{(i-1)}$. The mechanism is depicted in figure (9).

t_1: Proc(j) computes $y_j^{(i+1)} = B_j y^{(i-1)}$
t_2: Synchronization wait
t_3: Proc(j) broadcasts $y_j^{(i+1)}$ to all other processors
t_4: Proc(j) assembles $y^{(i+1)}$ from the received $y_k^{(i)}$
t_5: Designated processor computes $U^{\text{tr}} y^{(i+1)}$ and writes output to file
t_6: Synchronization wait

Figure 9: Black Box Segmented Transversely.

Step (t_1) produces the vector $y_j^{(i+1)}$ that is only $j < N$ long. This portion has to be broadcast to all other processors which then assemble $y^{(i+1)}$. Only then can the inner product be taken. This means that moderate quantities of data have to be transferred between processors and that at least two synchronization steps will be needed. Synchronization and data transfer are expensive when done across a local area network even if there is a shared file system. The partitioning strategy becomes more attractive on a multiprocessor with shared memory and a fast data link.

5.1.3 Longitudinally Segmented Matrix Black Box

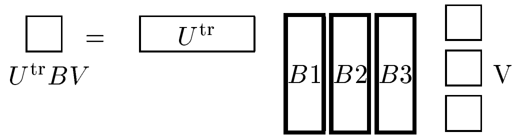

Figure 10: Distributing the black box with longitudinal partitioning : alternative strategy

The black box may also be broken into blocks of columns of B. In this strategy, each processor is given a black box representing κ_j columns of B which we denote by B_j. Each processor initially takes the vector Z as input . An application of the black box generates the vector $y^{(i+1)} = By^{(i)}$ as in figure (11).

In step (t_1), proc(j) produces an N-dimensional vector $y_{i+1}^{(j)}$ which it then broadcasts. It also receives a vector from each of the other processors. After that, the processor must sum these vectors so as to have a local copy of y_{i+1}. All this must happen before a designated processor can take the inner product

t_1: Proc(j) computes $y_j^{(i+1)} = B_j y_j^{(i)}$
t_2: Synchronization wait
t_3: Proc(j) broadcasts $y_j^{(i+1)}$ to all other processors
t_4: Synchronization wait
t_5: Each processor assembles $y^{(i+1)} = \sum_k^{n/j} y_k^{(i+1)}$
t_6: Synchronization wait
t_7: Designated processor computes $U^{\mathrm{tr}} y^{(i+1)}$ and writes to file
t_8: Synchronization wait

Figure 11: Black Box Segmented Longitudinally

and write the output to a file. This means that large quantities of data have to be transferred between processors and that at least two, and possibly three, synchronization steps will be needed. This cost of vector addition is not seen in transverse partitioning. This method of partitioning is unattractive for most applications and architectures.

6 Implementation

We implemented the algorithm in the C programming language. As shown in figure (5) our black box is a structure containing statically initialized data and two functions, `init` to do the initialization, and `apply`, which computes the matrix times vector product. No direct access to the static data is permitted to any other function, in keeping with the spirit of a matrix-free algorithm. The functions are passed by pointers to other procedures.

The software system is decomposed into three sections that are counterparts to the steps BW1, BW2, and BW3. Sequence generation was further split into a function that selects the vectors U and Z, and a module to actually compute the $\mathbf{a}^{(i)}$ that can be run in a distributed setting. The load is statically balanced by giving the ν^{th} processor a copy of the blackbox, the vector U and κ_ν columns of Z. Each processor then computes $\mathbf{a}_\nu^{(i)} = (U^{\mathrm{tr}} B^{i+1} Z)^{\mathrm{tr}}$ and puts it out to a file as an append operation, and then closes that file to minimize the effect of any fault. Barrier synchronization is employed before the sequential step of finding a linear generator.

The block size is a multiple of 32 bits, which is the width of an integer word on most computers.

The evaluation step, too, is executed in a distributed setting. Each processor receives a copy of the black box, the entire vector Z and κ_ν columns of each coefficient of the linear generator. As in the sequence generation step, the load is statically balanced by the choice of the grain size, κ_ν and barrier synchronization is employed.

The software architecture is depicted in figure (12). The tasks are of very long duration both in terms of CPU usage and elapsed time and checkpointing strategies are built in to give fault tolerance and error recovery.

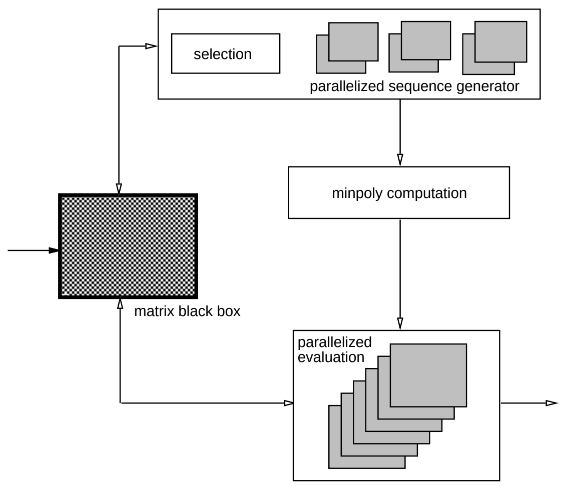

Figure 12: Software Architecture of the black box Block Wiedemann Algorithm.

Our intention is to have a coarse-grain parallel computation over a network of workstations or on an MIMD machine. Though the parallel tasks are of long duration, interprocessor communication is far more expensive than computation. Accordingly, the subtasks do not communicate with one another after they are started. They write their data directly to output files on a shared filesystem.

For the network of SPARC-20 workstations we wrote a UNIX script that took a list of available nodes and used a ready-queue mechanism to match subtasks to available nodes and used a simple busy wait for synchronization. No attempt was made to pick the best compute engine since only a few large workstations were available. The queue was essential when there were fewer available processors than subtasks. The script provided an environment and it was not linked into the solver. The solver in this case, was broken into four stand-alone modules.

On the IBM SP-2 parallel computer, we linked to the MPI library (`mpich`) to schedule the tasks of the solver. While maintaining the same partitioning, balancing and communication strategies, there was only one program written in a straightforward fashion, instead of discrete modules. Barrier synchronization is used for the parallel subtasks.

N	#proc	BW1	BW2	BW3	Total	Work
52,250	4	$0^h19'$	$1^h00'$	$0^h10'$	$1^h28'$	$308^\#$
	2	$0^h24'$	$1^h04'$	$0^h09'$	$1^h36'$	$347^\#$
252,222	2	$28^h46'$	$23^h54'$	$11^h52'$	$64^h32'$	$13820^\#$
569,466	2	$167^h39'$	$106^h45'$	$57^h40'$	$332^h04'$	$35630^\#$

Table 1: Parallel CPU Time (hours^{h}minutes$'$) for finding 128 solutions with optimized WLSS2 package on a network of SPARC-20 workstations, with variable grainsize $\kappa = 128/\#proc$. Each processor is rated at at 107.3 MIPS. Work is measured in units of (MIPS-hours$^\#$).

7 Experiments and Discussion

We conducted our tests on a network of SPARC-20 workstations with nominal ratings of 107 MIPS, and on an SP-2 parallel computer whose native MIPS ratings are not known to us at the present time. The software was compiled with the highest possible optimization flags on both platforms.

The test cases came from RSA challenge integer factoring experiments. The first was supplied by A. Odlyzko, and the other two representing the RSA-120 and RSA-129 efforts, were obtained from A. Lenstra. The systems were made square by padding with zero rows. Their specific details are as follows:

1. A $50,001 \times 52,250$ matrix over GF(2) containing 9 to 34 entries per row and 1.1 million entries totally.

2. A 245,811$\times$ 252,22 matrix over GF(2) containing 10 to 217 non-zero entries per row and 11.04 million entries totally.

3. A 524,339 $\times$ 569,466 matrix over GF(2) containing 26.6 million entries totally.

The blocking factor n, with $m = n$ was chosen to be an integer multiple of 32 bits, the word size of the machines. The granularity κ_ν was usually 32. Vectors, the matrix and the intermediate results were maintained and passed as files. A shared filesystem was in use and hence only filenames needed to be exchanged by tasks.

7.1 Discussion of Timings

Table (1) shows the parallel cpu-time for finding 128 solutions to linear systems over GF(2), with κ_ν a multiple of 32, on a network of SPARC-20 workstations.

It can be seen that the time for evaluation (step BW3) is approximately one half of the time for generating the sequence (step BW2). Each of these steps

N	#proc	BW1	BW2	BW3	Total
52,250	4	$0^h10'$	$0^h42'$	$0^h04'$	$0^h57'$
252,222	4	$7^h15'$	$15^h24'$	$3^h51'$	$26^h30'$

Table 2: Parallel CPU Time (hours^{h}minutes$'$) for finding 128 solutions with optimized WLSS2 package using fixed grainsize $\kappa = 32$, on an SP-2 multiprocessor using MPI scheduler.

involves matrix times vector products plus some additional work, but the dominating cost of these steps appears to be the matrix times vector multiplication. It makes sense, therefore, to optimize this task as much as possible.

The times for finding the linear generator (step BW2) grow quadratically with dimension of the matrix and overall, the total time appears to grow according to a function that is in $O(N^{2+\epsilon})$ where $0 < \epsilon \leq 1$.

Our tables give the total work, which is the product of the total time taken by all parallel and sequential tasks, multiplied by the native MIPS of the machines. This is a measure of the total number of instructions that the processors could have executed while active. Another point to observe is that for the smallest matrix, the time is approximately the same for granularities of 32 and 64.

Table (2) shows the time for solving two of the systems on an SP-2 multiprocessor. Scheduling was done by means of MPI. Comments similar to the ones for table (1) can be made here as well. We presently have no figure for the native MIPS of the individual processors.

Interestingly execution on any one node of the SP-2 is much faster than on a SPARC-20. We took care to see that the output data was always written to local disk space so the difference in timings is most likely to be the result of different processor power, different compilers or different disk write speeds and bandwidth. We did not have sufficient disk space to store the contents of the matrix and the intermediate files, for the RSA-129 matrix.

7.2 Scalability

The programs are scalable and will run on more processors, with appropriate adjustments to the blocking factor. There is a point when adding processors will not materially improve the overall performance.

We could for example use a larger blocking factor n and more processors to reduce the time for a parallel subtask in steps BW1 and BW3, which is inversely related to n. However the time complexity of the block Wiedemann algorithm is $O\left(\omega N/n + nN^2\right)$ finding the linear generator in step BW2 is directly proportionate to the blocking factor. Thus, there is a point in the total-time characteristic when the reduction of time due to parallelization is balanced by the increase in the sequential time. Past that point, the total time increases.

8 Conclusions

We have demonstrated the black box concept and have successfully parallelized a program in the outer loop. Our solutions are symbolic *i.e.* exact, rather than numerical approximations. Parallel subtasks are statically balanced and the computation is scalable. The problems of processor synchronization and the exchange of large quantities of intermediate data over a network, are circumvented by the decomposition of the program into three sets of subtasks that communicate by means of files alone.

We solved a system of 569,466 equations in 332 hours on a workstation cluster, and a system of 252,222 equations in about 26.5 hours on an SP-2. These and other results gave concrete evidence that the block Wiedemann algorithm could be successful in a field of any cardinality, and was a motivating factor in a search for a proof and estimate of success probability which has finally been given by Villard in theorem (4). Coppersmith attempted to give such a theorem, and relied on the notion of *pathological* matrices, for which, he claimed the algorithm would fail. Experiments such as ours failed to find a single pathological matrix. Thus in a certain sense, theory has been encouraged by experiment.

We are investigating theoretical issues of the block Wiedemann algorithm, and the parallelization of the linear generator step. We believe that the block algorithm is a viable alternative to structured Gaussian elimination because it is matrix-free and causes no loss of sparsity during the computations. Were sparsity to be lost, for example, in the case of our largest matrix, nearly 40Gb of storage would be needed to to accommodate the dense matrix even if just one bit was used per entry.

We find the block Wiedemann algorithm to be competitive to the block Lanczos algorithm which is susceptible to problems of of orthogonality of the vectors in its intermediate steps, that problem becoming more visible as the dimension grows.

In conclusion, we have shown how a non trivial problem can be solved on a network of inexpensive workstations as well as on an MIMD machine where each processor has a large amount of local memory and processing power, and where interprocessor communication is minimal.

We anticipate that the techniques described here will keep pace with the larger systems generated in the state of the art of integer factoring and will hence, indirectly, have an impact upon the security of public-key cryptosystems. Our next challenge is to solve a system of approximately 1,500,000 equations and variables generated in a factoring experiment. We plan to use a distributed black box strategy for this purpose

Acknowledgement: The authors thank Dr. Charles Norton for valuable discussions and assistance with the use of the SP-2. Thanks also to David Hollinger

and Nathan Schimke for technical support.

References

[1] Beckerman, B., and Labahn, G. A uniform approach for the fast computation of matrix-type Padé approximants. *SIAM J. Matrix Anal. Appl. 15*, 3 (1994), 804–823.

[2] Berlekamp, E. R. *Algebraic Coding Theory.* McGraw Hill, New York, 1968.

[3] Bitmead, R. R., and Anderson, B. D. O. Asymptotically fast solution of Toeplitz and related systems of linear equations. *Linear Alg. Appl. 34* (1980), 103–116.

[4] Coppersmith, D. Solving linear systems over GF(2). Tech. Rep. RC 16997, IBM Thomas J. Watson Research Ctr., 1991.

[5] Gohberg, I., Kailath, T., and Koltracht, I. Efficient solution of linear systems of equations with recursive structure. *Linear Alg. Appl. 80* (1986), 81–113.

[6] Golliver, R. A., Lenstra, A. K., and McCurley, K. S. Lattice sieving and trial division. In *Proc. ANTS '94* (1994).

[7] Kailath, T., Kung, S. Y., and Morf, M. Displacement ranks of matrices and linear equations. *Linear Alg. Appl. 80* (1979), 395–407.

[8] Kaltofen, E. Efficient solution of sparse linear systems, 1992. Lecture notes, Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, USA.

[9] Kaltofen, E. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. Tech. Rep. 93-22, Rensselaer Polytechnic Institute, Troy, NY, USA, 1993.

[10] Kaltofen, E., and Pan, V. Processor-efficient parallel solution of linear systems over an abstract field. In *Proc. 3rd Annual ACM Symp Parallel Algorithms and Architectures* (1991), ACM Press, pp. 180–191.

[11] Krylov, A. N. On the numerical solution of the equation by which the frequency of small oscillations is determined in technical problems (in Russian). *Izv. Akad. Nauk SSSR, Ser. Fiz.-Mat. 4* (1931), 491–539.

[12] LaMacchia, B. A., and Odlyzko, A. M. Solving large sparse linear systems over finite fields. In *Advances in Cryptology: CRYPTO '90* (1990), pp. 109–133.

[13] Lenstra, A. K., and Manasse, M. S. Factoring by electronic mail. In *Proc. Eurocrypt '89* (1990), pp. 355–371.

[14] Massey, J. L. Shift-register synthesis and BCH decoding. *IEEE Trans. Inf. Theory 15* (1969), 122–127.

[15] Morf, M. Doubling algorithms for Toeplitz and related equations. In *IEEE Intl. Conf. Acoust. Speech Signal Process.* (1980), IEEE, pp. 954–959.

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

[17] Rissanen, J. Realizations of matrix sequences. Tech. Rep. IBM Tech. Report RJ-1032, IBM, Yorktown Heights, NY, 1972.

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

[19] Rosenbrock, H. H. *State-Space and Multivariable Theory.* John Wiley and Sons, 1970.

[20] Villard, G. Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems. In *Proc. of ISSAC'97* (1997), pp. ?–?

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