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 and Information Sciences, The University of Delaware
Newark, Delaware 19711-1501, USA; email: alobo@cis.udel.edu

Abstract

We describe a coarse-grain parallel software system for the homogeneous solution of linear systems. Our solutions are symbolic, *i.e.*, exact rather than numerical approximations. Our implementation can be run on a network cluster of SPARC-20 computers and on an SP-2 multiprocessor. 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.

1 Introduction

The problem of solving large, unstructured, sparse linear systems using exact arithmetic arises in symbolic linear algebra and computational number theory. For example the sieve-based factoring of large integers can lead to systems containing over 569,000 equations and variables and over 26.5 million nonzero entries, that need to be solved over the Galois field of two elements. Problems of such a large size can be tackled by structured Gaussian elimination which heuristically inhibits fill-in and results in smaller, denser systems, than the original. It is not clear whether the denser system can be guaranteed to fit in the primary memory of a computer system.

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 context of symbolic computation, of the conjugate gradient algorithm (LaMacchia and Odlyzko, 1991), Lanczos algorithm (Coppersmith, 1991) and Wiedemann's (1986) coordinate recurrence algorithm which finds linear relations in Krylov subspaces.

*This material is based on work supported in part by the National Science Foundation under Grant No. CCR-9319776. Authors' previous address: Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180-3590.

Appears in *High Performance Computing '96*, Proc. 1996 Simulation Multiconference, A. M. Tentner (ed.), Simulations Councils Inc., pp. 244–247 (1996).

The iterative methods do not depend upon the structural properties of the matrix, in contrast to structured Gaussian elimination, and the property of sparsity is maintained. The methods are clearly open to parallelization when considering the steps inside the outer loop,*i.e.*, within the matrix times vector product.

A far more difficult task is to parallelize 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. One 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. The key property 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 (1969) algorithm. Evaluation of the minimum polynomial yields the solution, if one exists. Wiedemann's original algorithm is entirely sequential.

Coppersmith (1994) simultaneously used m vectors $\mathbf{x}$ for u and n vectors $\mathbf{y}$ for v. The sequence becomes

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

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 to $N/m + N/n + 2n/m + 1$. He also gave an analysis of the running time complexity. 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.

2 Algorithm Description

We will give just the salient features of the algorithm here and refer the interested reader to Coppersmith (1994) and Kaltofen (1995) for full details.

The object is to find more than one non-trivial solution to a homogeneous linear system with coefficient matrix B and dimension N, over the finite field of 2 elements. We set $m = n$ and define this quantity as the blocking factor. There are three steps to the algorithm.

Step BW1: *Sequence Generation.* Pick random vectors $\mathbf{x} = [x_1|x_2|\cdots|x_m]$, and $\mathbf{z} = [z_1|z_2|\cdots|z_n]$, $x_j, z_i \in \mathsf{K}^N$ for all $0 < i \le m$, $0 < j \le n$. Compute $\mathbf{y} = B\mathbf{z}$ and

$$\mathbf{a}^{(i)} = (\mathbf{x}^{\mathrm{tr}} B^i \mathbf{y})^{\mathrm{tr}}, \quad \text{for all } 0 \le i \le \frac{N}{m} + \frac{N}{n} + \frac{2n}{m} + 1. \quad (1)$$

The task requires not more than

$$\left(1 + \frac{n}{m}\right) N + \frac{2n^2}{m} + 2n \quad (2)$$

multiplications of a vector in K^N by B, which is represented by a black box. Actually, the κ_ν rows of $\mathbf{a}^{(i)}$ can be computed using κ_ν columns of the vector $\mathbf{y}$ as a coarse-grain parallel operation, shown in figure (1). The ν^{th} processor gets a copy of the black box for B, and the entire vector $\mathbf{x}$. Another way is to perform the computation $B \cdot (B^{i-1} y_\nu)$ in parallel, as Coppersmith does, for each i. The grain is much finer than before but synchronization might be needed.

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.

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

$$\Psi_\nu(\lambda, \mathbf{y}) = \lambda^D \mathbf{y}\mathbf{c}_{D,\nu} + \cdots + \lambda^{\delta+1}\mathbf{y}\mathbf{c}_{\delta+1,\nu} + \lambda^\delta \mathbf{y}\mathbf{c}_{\delta,\nu}$$

with $\mathbf{c}_{k,\nu} = \mathbf{0}$ for all $0 \le k < \delta \le D$. Each coefficient $\mathbf{c}_{j,\nu}$ is the ν^{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. Setting $\widehat{\Psi}_\nu(\lambda) = \lambda^{-\delta}\Psi_\nu(\lambda)$ compute

$$\widehat{w}_\nu = \widehat{\Psi}_\nu(B, \mathbf{z}) \quad (3)$$

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

With high probability $\widehat{w}_\nu \neq \mathbf{0}$. The product $B^d\widehat{w}$ is computed for the smallest integer $1 \le d \le \delta + 1$ that yields the zero vector. Finally $w_\nu = B^{d-1}\widehat{w}_\nu$ is returned. With parallelization, the block step of evaluation can yield as many as n individual candidate solution vectors w_ν which may not all be different or nonzero.

3 Distributed Implementation

We implemented the algorithm in the C programming language. 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 $\mathbf{x}$ and $\mathbf{y}$, 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 $\mathbf{x}$ and κ_ν columns of $\mathbf{z}$. Each processor then computes $\mathbf{a}_\nu^{(i)} = (\mathbf{x}^{\mathrm{tr}} B^{i+1}\mathbf{z})^{\mathrm{tr}}$ 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.

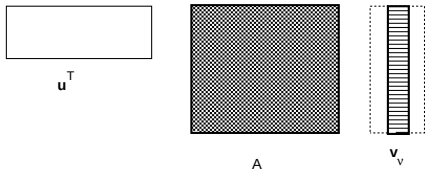

Figure 1: Distributed Sequence Generation.

The evaluation step, too, is executed in a distributed setting. Each processor receives a copy of the black box, the entire vector $\mathbf{z}$ 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 (2). 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.

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, 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 a 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.

We linked to the MPI library (`mpich`) to schedule the tasks of the solver on the IBM SP-2 parallel computer. 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.

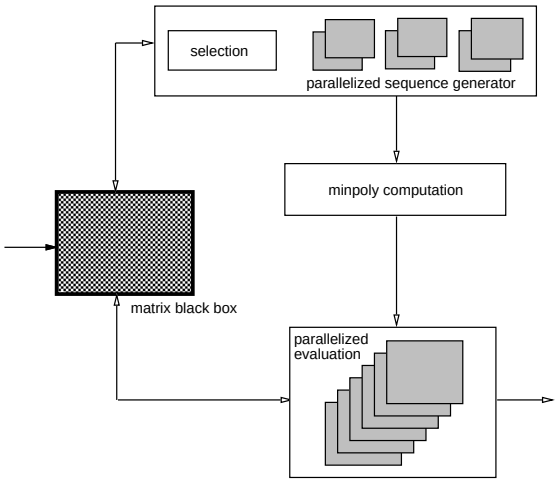

Figure 2: Software Architecture of the black box Block Wiedemann Algorithm .

3.1 Alternative Partitioning Schemes

Our partitioning scheme was chosen to avoid elaborate synchronization and to minimize the cost of data communication across a network. We also considered distributing the black box across processors. Two schemes are possible. In the first, the ν^{th} processor is given the entire vectors $\mathbf{z}$ and $\mathbf{x}$ and κ_ν rows of the matrix, thus leading to ν smaller black boxes. The action for sequence generation is shown in figure (3). The net result is that the $\nu \times n$ blocks generated by the subtasks have to be assembled into the $N \times n$ vector $\mathbf{y}^{(i)} = B^{i+1}\mathbf{z}_\nu$. This involves the broadcast of moderate amounts of data. One synchronization step is required. The scheme has merit in the context of an MIMD machine with a fast communication network.

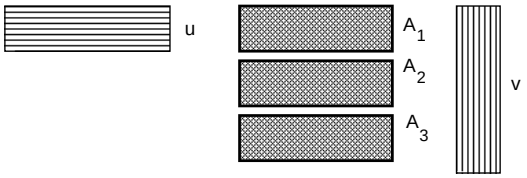

Figure 3: Distributed Black Box Partitioned Transversely.

In the other case, the matrix is partitioned by column, with the ν^{th} processor receiving $\mathbf{x}$, κ_ν rows of the vector $\mathbf{z}$ and a block of κ_ν columns of the matrix. In this case, the intermediate outputs are $N \times n$ vectors which have to be broadcast, added, and then rebroadcast to give $\mathbf{y}^{(i)}$. For large N, this could be prohibitively expensive. Two synchronization steps are needed.

4 Experiments

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 was chosen as an integer multiple of 32 bits, the word size of the machines. The granularity κ_ν was usually 32 except where noted. 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. We give the parallel time to find n solutions on the network of workstations in table (1). The timings for solving the first two test cases on the SP-2 multiprocessor are given in table (2). The number of processors used is given by n/κ_ν.

Table (1) shows the parallel cpu-time for finding n solutions to linear systems over GF(2), with κ_ν greater than 32 on a network of SPARC-20 workstations. The number of processors needed is given by $128/\kappa_\nu$ and is 2 when $\kappa_\nu = 64$ and 4 otherwise. It can be seen that the time for evaluation is approximately one half of the time for generating the sequence. Each of these steps 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 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 for finding 128 solutions is approximately the same when the granularities are 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 any one node of the SP-2 is much faster than a SPARC-20. We took care to see that the output data was always written to local disk space so the difference in timings could 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.

The programs are scalable. We could for example use a larger blocking factor n and more processors to reduce the time for a parallel subtask, which is inversely related to n. However the cost for finding the linear generator is

N	Grain	Sequence	Minpoly	Evaluation	Total	Work
52,250	32	$0^h19'$	$1^h00'$	$0^h10'$	$1^h28'$	$308^\#$
	64	$0^h24'$	$1^h04'$	$0^h09'$	$1^h36'$	$347^\#$
252,222	64	$28^h46'$	$23^h54'$	$11^h52'$	$64^h32'$	$13820^\#$
569,466	64	$167^h39'$	$106^h45'$	$57^h40'$	$332^h04'$	$35630^\#$

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

N	Grain	Sequence	Minpoly	Evaluation	Total
52,250	32	$0^h10'$	$0^h42'$	$0^h04'$	$0^h57'$
252,222	32	$7^h15'$	$15^h24'$	$3^h51'$	$26^h30'$

Table 2: Parallel CPU Time (hours^{h}minutes$'$) for finding 128 solutions with optimized WLSS2 package on 4 nodes of an SP-2 multiprocessor.

directly proportionate to the blocking factor so there is a point when the reduction in the parallel areas is balanced by the increase in the sequential area. Beyond that point, the total time increases.

5 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 are able to solve 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.

We are investigating theoretical issues of the block Wiedemann algorithm, and the parallelization of the linear generator step. We believe that the block algorithm 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 also find the 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. 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 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] Coppersmith, D. (1991). Solving linear systems over GF(2). *Tech. Report* **RC 16997** IBM Thomas J. Watson Research Ctr., Yorktown Heights, New York.

[2] Coppersmith, D. (1994). Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Math. Comput.***62**/205, 333–350.

[3] Kaltofen, E. (1995). Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Math. Comput.* to appear.

[4] LaMacchia, B. A. and Odlyzko, A. M. Solving large sparse linear systems over finite fields *Advances in Cryptology:* CRYPTO '90, Springer. Lect Notes Comput. Sci., **537**,109–133

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