Solving dense linear systems by Gauss-Huard's method on a distributed memory system

by
Walter Hoffmann, Kitty Potma and Gera Pronk
Department of Computer Systems, University of Amsterdam

Introduction

In this paper we present a modification of Gauss-Huard's method for solving dense linear systems that allows an efficient implementation on machines with a hierarchical memory structure. Gauss-Huard's method resembles Gauss-Jordan's method in the fact that it reduces the given system by elementary transformations to a diagonal system and it resembles regular Gaussian elimination in the fact that it only uses $\frac{2}{3} n^3 + O(n^2)$ floating-point operations to calculate the solution, where Gauss-Jordan's method uses $n^3 + O(n^2)$ floating-point operations to do so.

The method of Gauss-Huard was introduced in 1979 in a version that not included a stabilizing pivoting strategy [6]. After the stability of Gauss-Jordan's method had been properly established in 1989 [1], a stabilizing pivoting strategy for Gauss-Huard's method could be introduced. In 1992 it was proven that Gauss-Huard's method in combination with this pivoting strategy is numerically stable [3].

Our new formulation of Gauss-Huard's method allows an implementation with high data locality; that is to say, this variant is very efficient on machines with a hierarchical memory structure where transport of data to processors may take non-negligible time. The possibility of a better data utilization permits a better performance in comparison with Gaussian elimination on this type of machines. On distributed memory systems with a fast local memory, the performance of our algorithm approaches the performance of Gaussian elimination for large matrices.

Description of the basic algorithm

The Gauss-Huard elimination algorithm is used for solving linear systems of the form A x = b, for a given non-singular matrix A and right-hand side vector b.

With an appropriate pivoting strategy, it can be proven to be numerically stable. In practice, the results are equally satisfactory as the results attained with LU-factorization (Gaussian elimination). [2, 4, 5, 6]

The algorithm is explained by describing step k.

Suppose that at the beginning of step k, the first k-1 rows of A have been transformed such that the square upper left hand corner of the matrix equals the identity matrix and that the first k-1 elements of the right hand side vector b have been transformed accordingly.

1. Row elimination

The first k-1 elements of row k are eliminated by the first k-1 rows of the matrix. As a consequence, the elements numbered k up to n in the k-th row are modified by multiples of the elements in the upper right-hand part of the matrix.

2. Pivot selection and interchanging

An element of maximal size among the elements from position k up to n in the k-th row is selected. If such an element is found in column position p, columns p and k are interchanged (this is equivalent to renumbering the variables).

3. Scaling

Row k is divided by its current diagonal element (thus introducing the next value 1 at the diagonal).

4. Column elimination

The first k-1 elements in column k are eliminated by suitable multiples of row k, thus changing the upper right-hand part of the matrix.

The resulting matrix is an n-th order identity matrix and the right-hand vector is replaced by (a permutation of) the solution vector.

The algorithm is more formally described as follows (for ease of description we have not included the transformations on the right-hand side):

```
 1  for k = 1 to n do
 2     for i = 1 to k-1 do
 3        a_k. := a_k. - α_ki × a_i.
 4     enddo
 5     if |α_kq| = max k≤j≤n |α_kj|then pivot_index:= q
 6     if (pivot_index > k) then swap(a._q , a._k )
 7     a_k. := a_k. / α_kk
 8     for i = 1 to k-1 do
 9        a_i. := a_i. - α_ik × a_k.
10     enddo
11  enddo
```

Fig. 1 Stabilized Gauss-Huard algorithm

The algorithm as described above is rich in BLAS-2 constructions; lines 2-4 can be replaced by:

`A[k,k:n] -:= A[k,1:k-1] × A[1:k-1,k:n]`

("row vector becomes original value minus row vector times matrix"); lines 8-10 can be replaced by

`A[1:k-1,k+1:n] -:= A[1:k-1,k] × A[k,k+1:n]`

("matrix becomes original value minus rank-one matrix", generally called a "rank-one update").

The total number of floating-point operations equals:

$(\frac{1}{3}n^3 - \frac{1}{3}n)$ multiplications, $(\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n)$ additions and n divisions.

For the total number of floating-point operations in the Gauss-Huard algorithm this yields:

$\frac{2}{3}n^3 + O(n^2)$ flops.

For Gaussian elimination (LU -factorization) we find the same number of floating-point operations, also $\frac{2}{3}$ n^3+ $O(n^2)$ flops; for Gauss-Jordan elimination we find n^3+ $O(n^2)$ flops, which is 1.5 times as many operations.

A comparison of time needed for data access yields the following. The total number of fetches and stores (matrix elements that must be read from and written to memory) in the Gauss-Huard algorithm equals $2 \times \sum_{k=1}^{n} k \times (n-k) = \frac{1}{3} n^3$, stemming from the vector matrix multiplication and the rank-one matrix update.

For Gaussian elimination we find exactly the same number of fetches and stores, $\frac{1}{3} n^3$, where for Gauss-Jordan elimination we find $\frac{1}{2} n^3$+ $O(n^2)$ memory contacts.

Improved performance of the Gauss-Huard algorithm

Two ideas to improve the performance of the Gauss-Huard algorithm are suggested.

A. "Twining"

In step k of the algorithm, the elements that are created in the rank-one update (lines 8-10, Fig. 1), are used again in step k+1 for the vector matrix multiplication (lines 2-4, Fig. 1). If the last part of step k can be combined with the first part of step (k+1), a part of the matrix that is updated in step k can remain in fast memory to be used for its contribution in the vector-matrix multiplication in step (k+1). This can be achieved in various ways:

i) The relevant part of the matrix in the upper right-hand corner is updated row-wise according to the description in line 9, Fig. 1. Before being stored, each modified row is used for the calculation of its contribution in the linear combination that is needed for the row update in step k+1 (line 3, Fig. 1).

ii) The vector matrix product which is needed in the first part of the algorithm (lines 2-4, Fig. 1) can be calculated by taking innerproducts of the given row vector with columns of the upper right-hand part of the matrix. This requires a simple modification of the original algorithm. The matrix update can be calculated column-wise, which is realized by a modification of lines 8-10 in Fig.1. Before being stored, each modified column is used for the calculation of its innerproduct with row k+1 to calculate an element in the row update in step k+1.

B. "Blocking"

The algorithm can be formulated in a block version that allows the use of BLAS-3 type linear algebra operations. Suppose that the matrix is divided in blocks of size b. For simplicity, suppose that n, the order of the matrix, equals $b \times s$. In s steps the matrix is transformed into the identity matrix as follows.

Assume that (k-1) steps in the block algorithm have been performed such that an identity matrix of size $p = (k-1) \times b$ has been created.

The k-th step in the algorithm is described by:

1. Block-Row elimination

The k-th row-block, a rectangular block of size b × p, is eliminated by the leading order p identity matrix. The effect of this elimination is described by updating the remaining part of the k-th row-block by the appropriate matrix - matrix product (see Fig. 2).

2. Block diagonalization

Apply the scalar algorithm to the next strip of b rows. The diagonal block of size b × b is transformed into an identity matrix by the Gauss-Huard algorithm applied to the appropriate submatrix of size b × {n - p}(see Fig. 3).

3. Block-column elimination.

The submatrix that is denoted by V (Fig. 4) is eliminated by subtracting a suitable multiple of the b × b identity-matrix at the diagonal. The effect on all of the matrix is the subtraction of the matrix product V× W from U (cf. Fig. 4).

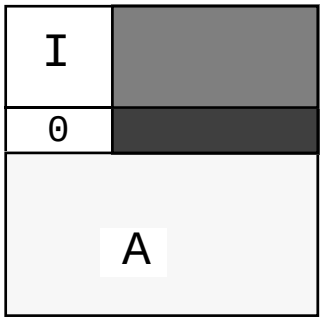

Fig. 2
Block-row elimination

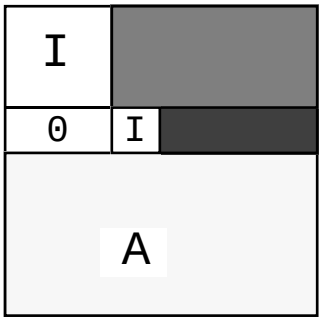

Fig. 3
Block diagonalization

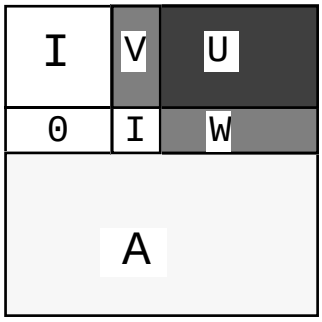

Fig. 4
Block-column elimination

The two ideas for improvement of the algorithm can be combined.

"Block - twining"

A combination of twining and blocking yields so called Block - twining . The idea is described as follows. In the last step of the block algorithm as described above, the part of the matrix denoted by U is modified according to the expression U = U - V × W. In the beginning of the next step, the part of the matrix denoted by (X|Y) (see Fig. 5) must be eliminated. This is accomplished by changing Z according to $Z := Z - (X \mid Y) \times \left(\frac{U'}{W}\right) = Z - X\,U' - Y\,W$

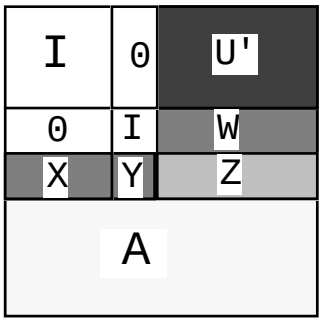

Fig. 5
Block-twining

For a computer with a hierarchical memory structure the amount of data-traffic can be minimized by updating matrix U in blocks, each block consisting of (multiples of) b rows and columns. Matrices W and Z (or parts thereof) must be kept in fast memory.

Each time a new part of U is imported into fast memory, it is updated by the appropriate part of V times W after which the product of the next part of X times this updated part of U is subtracted from Z. Finally the product Y W must be subtracted from Z.

We have implemented the block algorithm on the distributed memory systems that are available at our department, a 64 processor Meiko machine and a 512 processor Parsytec machine. The matrices

are distributed over the local memories of the processors, using a wrapped around distribution of the blocks.
The performance of our version of Gaussian elimination [5,6] on these machines is almost optimal (it approaches the peak performance for large matrices) and can hardly be expected to be 'beaten' by other codes for solving dense linear systems. However, it is expected that for very large matrices our block Gauss-Huard implementation will come close.
The algorithm is expected to show its full potential on computers that have a fast cash memory. We expect that for this type of computers, it can be implemented to be the most efficient direct solver for large dense non-singular matrices.

Literature:

1. T.J. Dekker, W.Hoffmann; Rehabilitation of the Gauss-Jordan algorithm, Numerische Mathematik 54 (1989) 591-599

2. T.J. Dekker, W.Hoffmann, K.Potma; Parallel Algorithms for Solving Large Linear Systems, Dept. of Computer Systems, Univ. of Amsterdam, Technical Report CS-92-12, July 1992

3. T.J. Dekker, W.Hoffmann, K.Potma; Stability of Gauss-Huard Elimination for Solving Linear Systems, Dept. of Computer Systems, Univ. of Amsterdam, Technical Report CS-93-08, August 1993

4. W.Hoffmann; A fast variant of the Gauss-Jordan Algorithm with partial pivoting; in: "Basic Transformations in Linear Algebra for Vector Computing," thesis, University of Amsterdam, 1989

5. W.Hoffmann and K. Potma; Threshold Pivoting in Gaussian Elimination to Reduce Inter-Processor Communication; Dept. of Computer Systems, Univ. of Amsterdam, Technical Report CS-91-05, August 1991

6. W.Hoffmann, K. Potma and Z.W. Zhang; Solving Dense Linear Systems Efficiently on the Parsytec GCel; Dept. of Computer Systems, Univ. of Amsterdam, Technical Report CS-93-16, November 1993

7. P. Huard, La methode du Simplexe sans inverse explicite; bulletin E.D.F. Série C n.2 (1979)