Matrix Inversion Using Parallel Processing

MARSHALL C. PEASE

Stanford Research Institute, Menlo Park, California*

ABSTRACT. Two general methods of matrix inversion, Gauss's algorithm and the method of bordering, are analyzed from the viewpoint of their adaptability for parallel computation. The analysis is not based on any specific type of parallel processor; its purpose is rather to see if parallel capabilities could be used effectively in matrix inversion.

It is shown that both methods are indeed able to make effective use of parallel capability. With reasonable assumptions on the parallelism that is available, the speeds of the two methods are roughly comparable. The two methods, however, make use of different kinds of parallelism.

To implement Gauss's algorithm we would like to have (a) parallel transfer capability for n numbers, if the matrix is $n \times n$, (b) the capability for parallel multiplication of the accessed numbers by a common multiplier, and (c) parallel additive read-in capability. For the method of bordering, we need, primarily, the capability of forming the Euclidean inner product of two n-dimensional real vectors. The latter seems somewhat harder to implement, but, because it is an operation that is fundamental to linear algebra in general, it is one that might be made available for other purposes. If so, then the method of bordering becomes of interest.

1. *Introduction*

It is our purpose here to consider two methods for the inversion of nonsingular real square matrices [1, 2], and to discuss their applicability for parallel computation.

Previous studies of matrix inversion by machine computation have largely been confined to methods suitable for sequential operation. This has been a reasonable restriction, since existing computers all operate in an essentially sequential manner. Recently, however, there has been considerable interest in the development of computers and subunits of computers that have been adapted to parallel operation. The feasibility of devices of this sort has been demonstrated and some—such as associative, or content-addressed memories—are either in production or close to it. It seems evident that in the near future parallel processors will become increasingly available. It is becoming important, therefore, to consider how such capability can be exploited in the various classes of computational problems commonly encountered.

The problem of inverting matrices is one that occurs in many problems of practical importance. Also, it is a problem that can easily grow out of bounds as large matrices are considered. It is a problem for which there is great advantage in improving computational speed. It has seemed useful to consider, in general terms, how parallel operation could be exploited in the inversion of matrices.

Some work has been done in this direction by other workers. For example, Crane and Githens [3] have considered the programming of a content-addressed memory for various problems, including, as one example, matrix inversion. However, these studies usually have been concerned with particular types of parallel processing,

* Computer Techniques Laboratory. This work was sponsored by the Office of Naval Research, Information Systems Branch, under Contract Nonr 4833(00).

whereas our concern is with the problem itself and with parallel processing in general.

A related, but not identical, problem is that of solving a set of linear equations, i.e., solving the vector equation[1]

$$\mathbf{Ax} = \mathbf{y} \tag{1}$$

for $\mathbf{x}$, given $\mathbf{y}$.

We chose to study the inversion problem, rather than that of solving eq. (1), since we felt that, by so doing, we would be able to reach a better understanding of what could be accomplished, and how, with parallel techniques. However, the reader should keep in mind that the two problems are not identical, so that statements regarding the relative merits of inversion procedures may not carry over into the solution of eq. (1).

We consider two general methods of matrix inversion. The first, using Gauss's algorithm, is important since most methods in current use are of this type.

The second is the *method of bordering*. It is discussed to some extent by Faddeeva [4], but has not generally been used, since it is not, in general, as efficient as Gauss's algorithm in serial processes.

2. *Gauss's Algorithm*

The basis of most of the methods that are currently used is Gauss's algorithm. This is usually stated as a procedure for solving eq. (1). However, it can be readily adapted to obtaining the inverse of $\mathbf{A}$, providing one exists.

We observe, first, that any row operation on $\mathbf{A}$ can be described as a sequence of premultiplications of $\mathbf{A}$ by the following nonsingular matrices, taken as elementary:

i. $\mathbf{C}_i(\lambda)$, which multiplies row i by a scalar $\lambda \neq 0$. $\mathbf{C}_i(\lambda)$ is diagonal with 1's on the main diagonal except in the ith column, where there is λ.

ii. $\mathbf{C}_{ij}$, which interchanges row i and row j, has coefficients c_{rs} as follows:

$$c_{rr} = 1 \quad \text{if} \quad r \neq i \text{ or } j; \qquad c_{ij} = c_{ji} = 1; \qquad c_{rs} = 0 \quad \text{otherwise.}$$

iii. $\mathbf{D}_{ij}(\lambda)$, which adds λ times row j to row i, has coefficients d_{rs} as follows:

$$d_{rr} = 1 \quad \text{for all } r; \qquad d_{ij} = \lambda; \qquad d_{rs} = 0 \quad \text{otherwise.}$$

For the solution of eq. (1), Gauss's algorithm may be described as a procedure for finding a matrix $\mathbf{C}$ which, when premultiplying $\mathbf{A}$, reduces it to either upper or lower triangular form. The matrix $\mathbf{C}$ is developed as a product of the elementary matrices given above. Applying $\mathbf{C}$ to eq. (1), we obtain

$$(\mathbf{CA})\mathbf{x} = \mathbf{Cy}. \tag{2}$$

Since $(\mathbf{CA})$ is triangular, eq. (2) is easily solved.

We can handle inversion in a similar fashion. Conceptually, we set up the supermatrix $\mathbf{B} = (\mathbf{A}\ \mathbf{I})$. (We say "conceptually" since procedures can be devised which avoid the necessity of actually setting up the supermatrix, and which therefore use memory more efficiently than indicated here. See below.) We first develop a $\mathbf{C}$

[1] We use boldface capitals to indicate matrices and boldface lower-case letters to indicate vectors.

that triangularizes **A**, and apply it to **B**. We then continue the process to obtain a **D** that reduces (**CA**) first to diagonal, and then to the unit matrix. That is, we find a **D** such that

$$\mathbf{DC}(\mathbf{A}\ \mathbf{I}) = (\mathbf{DCA}\ \mathbf{DC}) = (\mathbf{I}\ \mathbf{DC}).$$

Since $\mathbf{DCA} = \mathbf{I}$, $(\mathbf{DC}) = \mathbf{A}^{-1}$.

The basic process for determining **C** first makes certain that $a_{11} \neq 0$. If $a_{11} = 0$, we first permute rows by a suitable $\mathbf{C}_{i,j}$ to bring a nonzero element into this position. (There must be at least one nonzero entry in the first column, since **A** is nonsingular.) We then use $n-1$ elementary matrices $\mathbf{D}_{i1}(\lambda)$ to reduce the rest of this column to zero by adding to each row the appropriate multiple of the first row.

Next, we make certain that the new $a_{22} \neq 0$. If $a_{22} = 0$, we permute row 2 with some row for which $i > 1$, to bring a nonzero element into this position. (There must be at least one $a_{i2} \neq 0$, $i > 1$, since otherwise column 2 would be a multiple of column 1 and **A** would be singular.) We use several $\mathbf{D}_{i2}(\lambda)$ to reduce the rest of column 2 to zero.

We continue until **A** is triangular. The process cannot terminate prematurely if **A** is nonsingular.

To find **D**, we now repeat the process, going backward and upward. We use a_{nn} to eliminate all other elements from the nth column. Then we use $a_{n-1,n-1}$ to eliminate all elements above it from the $(n-1)$-st column, and so on, until **A** is diagonal. Finally, using $\mathbf{C}_i(\lambda)$, we divide each row by a_{ii}, and so obtain the identity.

Having modified, by row manipulations, (**A I**) to obtain the identity on the left, we read $\mathbf{A}^{-1}$ on the right.

This process is theoretically valid but may fail because of numerical instability in the computations. To avoid such instability, pivoting is used. By this we mean that, in preparation for the elimination of the subdiagonal elements in each new column, we permute the remaining rows, or rows and columns, so as to place a more appropriate element—perhaps the largest one (in absolute value), although other criteria are sometimes used—on the diagonal position. This coefficient is then the one that is used to eliminate the others in that column. In the case of "full pivoting," in which the largest available element of the remaining matrix is moved into the pivot position, instability has never been observed. See [5].

There are several variations of the basic pivoting procedure that are used, involving something less than full pivoting. For example, we may restrict ourselves to row permutations, or look for a coefficient that is not necessarily the largest available but is, instead, simply greater than some preset limit.

There are also other variations which minimize the amount of additional memory required.

It is beyond the scope of this paper to consider these variations of pivoting. The interested reader can find them discussed in considerable detail in [5]. The present concern is for the extent by which the process itself can be parallelized, assuming pivoting either is not needed or is taken care of along the way.

Considering, now, how parallel capability can be utilized in a procedure using Gauss's algorithm, several applications are evident:

A. The manipulations described by the **C** or **D** matrices do the same thing to all elements of a row (or column). All the elements of the pivot row, for example,

are multiplied by the appropriate number and subtracted from the row being affected. There is evidently great advantage in being able to do all of these operations in parallel.

B. In terms of the basic algorithm using serial processing, we can either (a) diagonalize on a single pass through the matrix, or (b) first triangularize and then diagonalize. For matrix inversion, there is no clear choice. (In solving a set of linear equations (1), it is necessary only to triangularize.) Including the operations on I in the supermatrix, both processes require, asymptotically, n^3 operations, where an operation is defined as the replacement of a_{ij} by $a_{ij} - a_{kj}a_{ik}/a_{kk}$, where a_{kk} is the pivot element. (The computation of $1/a_{kk}$ need be done only once per pivot element, and that of a_{ik}/a_{kk}, only once per row, so that our operation is essentially one multiplication and one subtraction.)

With parallel processing, the total number of operations is not significant. What matters is the number of sets of operations, where each set involves those being done in parallel.

The count of such sets requires some assumption of the parallel capability present. The assumptions we believe to be reasonable are that: (a) all the coefficients of a single row can be accessed and processed in parallel; (b) it is possible, by a single command, to cause the same operations to be performed simultaneously on all the coefficients so accessed.

With these assumptions it becomes better to complete the diagonalization in a single sweep, since then a given row need be accessed only once and then used to complete the diagonalization of a column.

C. To consider the potential for improvement, we suppose we have two auxiliary memories which we label R and S, each capable of holding $(n+1)$ words. The main memory is assumed to contain an $n \times (n+1)$ array. The matrix $\mathbf{A}$ is inserted in the first n columns, the remaining column being filled with 1's to represent the identity. We denote by $\mathbf{B}$ the changing matrix which is originally the identity and ultimately $\mathbf{A}^{-1}$. The program that follows is designed to minimize the main memory required by compressing the supermatrix that is conceptually used. Then, assuming no pivoting is necessary, we can use the following program:

For i stepped from 1 to n:
- (a) Read out the ith row into the R-memory.
- (b) Compute $k_{ii} = 1/a_{ii}$.
- (c) Multiply the R-memory by k_{ii} and leave in R.

For j stepped from 1 to n, skipping $j = i$:
- (d) Read out a_{ji}.
- (e) Read $(-a_{ji})$ times the R-memory into S, without changing R.
- (f) Add the nth coefficient in S to the ith. Set the nth coefficient to zero.
- (g) Additively read S into the jth row of the main memory. This procedure eliminates a_{ji} from $\mathbf{A}$ and moves the corresponding coefficient of $\mathbf{B}$ into this position.
- (h) After completing steps (d)-(g) through $j = n$, clear and add the nth coefficient in R into the ith position in R.
- (i) Clear and add R into the ith row of the main memory. This reduces the ith diagonal entry in $\mathbf{A}$ to unity, and suppresses it, replacing it by the corresponding entry of $\mathbf{B}$.
- (j) Step i as indicated, and repeat from (a) if $i < n$.

When the program is completed through $i = n$, the inverse of $\mathbf{A}$ is contained in the first n columns of the main memory.

Of these steps, (b), which computes a reciprocal, and (c) and (e), which involve parallel multiplication with a common multiplier, are apt to be considerably slower than the other steps. Furthermore, these steps will take the same time each time they occur. In inverting an $n \times n$ matrix, steps (b) and (c) are done n times, and step (c), $n(n-1)$ times. Step (e) will dominate, and the time required will asymptotically equal kn^2, as $n \to \infty$, where k is some constant not less than unity whose exact value depends on the effect of the transfer steps.

D. In the paragraph above, note the comparison between n^3 operations for the serial processor, and kn^2 major steps for the parallel processor. While these figures do not give a comparison of the resultant speeds that is wholly fair, it is valid to infer that parallel processing can be very much faster for large matrices.

E. Other parallel programs can also be considered, which may be useful in particular circumstances or for particular types of matrices, of which the tridiagonal type is a notable example. We can, for example, handle each column serially, but start working on the next column before the first is completed. Working initially on the ith column, we can start on the $(i+1)$-st column as soon as the $(i+1)$-st row is finished, providing the resultant $a_{i+1,i+1}$ is a suitable pivot element. After the next step, when the terms in the ith and $(i+1)$-st columns in the $(i+2)$-nd row have been eliminated, we can start work on the $(i+2)$-nd column, and so on. Providing the operation on each column is kept one step behind that on the previous column, the operations will not interfere.

Such a process in general appears to be more involved, and to require more complex logical and arithmetic capabilities than the purely parallel process described above. However, it might be advantageous in particular circumstances, depending on the particular capabilities of the processor being used, or for specialized types of matrices. For example, if we have available a processor that has the parallel capability required for large matrices, then it has excess capacity for small matrices. By simultaneously operating on two or more columns, we can make use of this excess capacity to obtain a further increase of speed.

3. *Bordering Method*

The second method we discuss is also a practical one. It has not been found to be generally useful for serial processing, but may be well suited for processors having particular parallel capabilities. For example, it appears that it would be of great general utility to be able to form the inner product of two n-dimensional vectors by a single parallel operation. The method of bordering is, then, a way of using this capability in computing the inverse.

The basic method is to invert a $k \times k$ submatrix that consists of the first k rows and columns. We use this submatrix to find the inverse of the $(k+1) \times (k+1)$ submatrix that is the original submatrix bordered by the next row and column. We repeat until the entire matrix is inverted.

We start with the a_{11} element, which, for the moment, we assume to be nonzero. We replace it by its reciprocal. We consider this as a 1×1 matrix bordered by the elements a_{12}, a_{22}, and a_{21}, and obtain the inverse of the 2×2 matrix in the upper left corner. We continue until the entire matrix is inverted.

At any given step, after the $(k-1) \times (k-1)$ submatrix has been inverted, we

consider the matrix $\mathbf{M}_k$, which contains the first k rows and columns of the contents of the memory. This is given by

$$\mathbf{M}_k = \begin{pmatrix} \mathbf{A}_{k-1}^{-1} & \mathbf{u} \\ \mathbf{v}^T & a_{kk} \end{pmatrix}.$$

Here $\mathbf{A}_{k-1}$ is the submatrix consisting of the first $k-1$ rows and columns of $\mathbf{A}$. In $\mathbf{M}_k$, we assume that we have already inverted $\mathbf{A}_{k-1}$. In $\mathbf{M}_k$, then, $\mathbf{A}_{k-1}^{-1}$ is bordered by $\mathbf{u}$, $\mathbf{v}^T$, and a, where $\mathbf{u}$ is the vector consisting of the first $(k-1)$ rows of the kth column of $\mathbf{A}$ and $\mathbf{v}^T$ is the transposed vector consisting of the first $(k-1)$ columns of the kth row. $\mathbf{A}_k$ is given by

$$\mathbf{A}_k = \begin{pmatrix} \mathbf{A}_{k-1} & \mathbf{u} \\ \mathbf{v}^T & a_{kk} \end{pmatrix},$$

and its inverse is easily found to be

$$\mathbf{A}_k^{-1} = \begin{pmatrix} \mathbf{A}_{k-1}^{-1} + p(\mathbf{A}_{k-1}^{-1}\,\mathbf{u})\,(\mathbf{v}^T\mathbf{A}_{k-1}^{-1}) & -p\mathbf{A}_{k-1}^{-1}\,\mathbf{u} \\ -p\mathbf{v}^T\mathbf{A}_{k-1}^{-1} & p \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{A}_{k-1}^{-1} & \mathbf{0} \\ \mathbf{0} & 0 \end{pmatrix} + p \begin{pmatrix} \mathbf{A}_{k-1}^{-1}\,\mathbf{u} \\ -1 \end{pmatrix} (\mathbf{v}^T\mathbf{A}_{k-1}^{-1} \quad -1),$$

where

$$p = p_k = (a_{kk} - \mathbf{v}^T\mathbf{A}_{k-1}^{-1}\mathbf{u})^{-1}.$$

We assume that $1/p_k \neq 0$. If this fails, then pivoting must be used. We have not studied the stability problem for this method, but would expect it to be necessary to avoid letting p_k get too large by comparison with the remaining elements of the matrix. Again, such problems are beyond the scope of this paper.

Assuming, then, that p_k is well behaved, we need to pull out of the array the subvectors $\mathbf{u}$ and $\mathbf{v}^T$, and the coefficient a_{kk}. We compute p_k and the k-dimensional vector col $\{(\mathbf{A}_{k-1}^{-1}\mathbf{u}), -1\}$ and row $\{(\mathbf{v}^T\mathbf{A}_{k-1}^{-1}), -1\}$. The correction dyad is then read in as it is computed.

Assuming approximately the same amount of parallel computation facilities as we assumed for Gauss's algorithm, but organized so as to form Euclidean inner products, we can break up the operation into the following substeps:

(a) Calculate $\mathbf{A}_{k-1}^{-1}\mathbf{u}$ and form the vector col $\{(\mathbf{A}_{k-1}^{-1}\mathbf{u}), -1\}$. This requires the computation of $(k-1)$ components. Clear the coefficients in $\mathbf{u}$ as $\mathbf{u}$ is read out of the main memory. As a reasonable specification of what parallel capability we might expect to have, we assume the capability of calculating only one coefficient at a time. This amounts to assuming that we can form the Euclidean inner product of two $(k-1)$-dimensional vectors ($\mathbf{u}$ and a row of $\mathbf{A}_{k-1}^{-1}$) as a simultaneous computation. There are, then, $k-1$ such computations.

(b) Calculate $\mathbf{v}^T\mathbf{A}_{k-1}^{-1}$ and form the vector row $\{(\mathbf{v}^T\mathbf{A}_{k-1}^{-1}), -1\}$. Clear the coefficients in $\mathbf{v}^T$ from the main memory. On the same assumptions as in (a), this requires $(k-1)$ computations.

(c) Calculate $\mathbf{v}^T\mathbf{A}_{k-1}^{-1}\mathbf{u}$. This is a single computation by the previous assumption.

(d) Calculate p and clear a_{kk} from the main memory. This requires one subtraction and one reciprocation; we call this one computation.
(e) Calculate p times the vector obtained in (a). This is one computation.
(f) Taking the coefficients of the vector formed in (b) one at a time, form the product of this coefficient with the vector found in (a) and additively read it into the corresponding column of the main memory. This requires k computations.

The kth major step then requires $3k+1$ computations. For an $n \times n$ matrix, we require $N(n)$ computations, where

$$N(n) = N(n-1) + 3n + 1,$$

$$N(1) = 3 \qquad \text{(steps (d), (e), and (f)).}$$

Hence

$$N(n) = \tfrac{1}{2}(n+2)(3n-1).$$

Since this is asymptotic to $3n^2/2$, the method is comparable to Gauss's method, which required kn^2, where k is at least 1 and may be significantly higher.

In comparing the two methods, we have not included in either case the effect of pivoting. The actual operations necessary for pivoting are the same in either case. The tests for the best pivot are quite different, however. In Gauss's method we need only find the largest of the available elements. Hence the test is a simple search procedure for which certain kinds of parallel processors (e.g., content-addressed memories) are very well adapted.

In the bordering method, the test for the best pivot element requires us to find an element for which $| a_{ij} - \mathbf{v}^T\mathbf{A}_{k-1}^{-1}\mathbf{u} |$ has its maximum value. These coefficients are obtained only after a considerable amount of computation.

It appears, then, that the method of bordering should not be used in cases where extensive pivoting is to be expected. If there is any likelihood that extensive pivoting will be required, Gauss's algorithm has a decisive advantage.

There are important classes of matrices for which pivoting is not required, for example, matrices which are nearly diagonal and matrices which are symmetric and known to be positive definite.[2] For these matrices, the method of bordering may be preferable in terms of speed and size capability, depending on the particular class of matrices involved, and the facility with which the parallel requirements can be implemented.

4. *Conclusions*

In this paper, we have considered how we might program the inversion of nonsingular matrices for a processor having parallel access and computation capability. In particular we have considered two general methods of inversion: Gauss's algorithm and the method of bordering.

We have intentionally avoided assuming any particular set of parallel capabilities. Our purpose, instead, has been to gain some insight into what capabilities would be most advantageous, and to develop a "feel" for the overall benefits that could be obtained from such capabilities. A precise and quantitative evaluation of

[2] That positive definite symmetric matrices do not need pivoting was pointed out to us by the referee.

the relative merits of the two methods must obviously depend on the detailed capabilities that are available. Hence, we can only say that the two methods look roughly comparable in terms of speed.

We do observe a significant difference in the parallel capabilities that could be effectively utilized by the two methods. In the case of Gauss's algorithm we were lead to postulate the availability of (a) parallel readout, (b) parallel multiplication by a common multiplier, and (c) parallel additive readin. In the method of bordering, we were lead to assume primarily the capability of forming the Euclidean inner product of two n-dimensional vectors, as well as parallel access and additive readin. The Euclidean inner product is a more complex operation but, because of its fundamental importance in the theory of linear vector spaces, might be made available for other purposes as well. If it is available, then the method of bordering should be considered as a procedure of considerable interest for the inversion of matrices.

We conclude, then, that both methods remain of interest, and that both can make very effective use of parallel processing capability.

ACKNOWLEDGMENT. The author is indebted to W. H. Kautz of Stanford Research Institute for stimulating his interest in the problem, and for the benefit of his experience in the general field of parallel processing.

REFERENCES

1. PEASE, M. C. *Method of Matrix Algebra.* Academic Press, New York, 1965. (This is a good source for general matrix theory.)
2. GANTMACHER, F. R. *The Theory of Matrices* (2 vols.). Chelsea, New York, 1960. (This is a good source for general matrix theory.)
3. CRANE, B. A., AND GITHENS, J. A. Bulk processing in distributed logic memory. *IEEE Trans. EC-14*, 2 (April 1965), 186–196.
4. FADDEEVA, V. N. *Computational Methods of Linear Algebra.* Dover, New York, 1959.
5. WILKINSON, J. H. *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, 1965. (This gives a detailed discussion of the stability problem in general and of the effect of various pivoting procedures.)

RECEIVED JUNE, 1966; REVISED APRIL, 1967