STORAGE SCHEMES FOR PARALLEL EIGENVALUE ALGORITHMS

Robert A. van de Geijn

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-34 September 1988

Abstract

In this paper, we examine storage schemes for parallel implementation on distributed memory MIMD multiprocessors of algorithms for solving the algebraic eigenvalue problem. We show that a novel storage scheme, which we call block hankel-wrapped storage, allows better utilization of the processors than column-wrapped storage when implementing Jacobi's Method or the nonsymmetric QR algorithm.

1 Introduction

With the advent of parallel processors, much research has concentrated on the parallel implementation of algorithms in numerical linear algebra. Recent papers on parallel eigenvalue algorithms include [4,10] for symmetric, tridiagonal matrices. For full symmetric matrices, Jacobi methods are again of interest [2,3,8,11,12,16]. Papers examining the implementation of QR methods for nonsymmetric matrices include [1,5,14,15].

In this paper, we examine storage schemes for implementation of eigenvalue algorithms on distributed memory multiprocessors. The *column-wrapped* storage scheme, which is useful for parallel implementation of algorithms for solving linear systems [9], does not seem to allow efficient parallel implementation of eigenvalue algorithms. An alternate storage scheme, which we call *block hankel-wrapped* storage, shows promise for parallel implementation of a wide range of algorithms for solving the eigenvalue and singular value problems [16]. In §2, the column-wrapped and block hankel-wrapped storage schemes are defined. Next, we examine the usefulness of these schemes for efficient parallel implementation of Jacobi's method in §3 and the nonsymmetric QR algorithm in §4. Concluding remarks are given in §5.

Throughout this note, we assume all matrices, vectors, scalars, and arithmetic to be real. We further assume that the parallel processor has p processors, $\mathbf{P}_0, \ldots, \mathbf{P}_{p-1}$, and $\mathbf{P}_i$ is adjacent to $\mathbf{P}_j$ if $|i-j|\%p = 1$, where .%. indicates the modulo operator. In other words, we assume it is possible to embed a ring within the network of processors.

2 Parallel Storage Schemes

A popular storage scheme for distributed memory parallel computers is the column-wrapped storage scheme. It assigns the ith column of given matrix A to processor $\mathbf{P}_{(i-1)\%p}$.

As an alternative, we define the class of *block hankel-wrapped* storage schemes. Given m, for simplicity we assume the dimension of matrix A is $n = mhp$ for some integer h. Partition the matrix so that $A = (A_{ij})$, $1 \le i,j \le n$, where $A_{ij} \in \mathbf{R}^{h\times h}$. For general m, the block hankel-wrapped storage scheme (BHW_m), assigns A_{ij} to $\mathbf{P}_{\lfloor (i+j-2)/m \rfloor \%p}$. The superscripts in the following figure indicate the processor to which the blocks are assigned when $m = 1$:

$$\begin{array}{ccccc} A_{11}^{(0)} & A_{12}^{(1)} & A_{13}^{(2)} & \cdots & A_{1p}^{(p-1)} \\ A_{21}^{(1)} & A_{22}^{(2)} & A_{23}^{(3)} & \cdots & A_{2p}^{(0)} \\ A_{31}^{(2)} & A_{32}^{(3)} & A_{33}^{(4)} & \cdots & A_{3p}^{(1)} \\ \vdots & \vdots & \vdots & & \vdots \\ A_{p1}^{(p-1)} & A_{p2}^{(0)} & A_{p3}^{(1)} & \cdots & A_{pp}^{(p-2)} \end{array}$$

It should be noted that this storage scheme is an example of a *skewed storage scheme* [7].

For reasons explained in the next section, we prefer BHW_2, which assigns submatrices to processors as illustrated by

$$\begin{array}{ccccc} A_{11}^{(0)} & A_{12}^{(0)} & A_{13}^{(1)} & \cdots & A_{1(2p)}^{(p-1)} \\ A_{21}^{(0)} & A_{22}^{(1)} & A_{23}^{(1)} & \cdots & A_{2(2p)}^{(0)} \\ A_{31}^{(1)} & A_{32}^{(1)} & A_{33}^{(2)} & \cdots & A_{3(2p)}^{(0)} \\ \vdots & \vdots & \vdots & & \vdots \\ A_{(2p)1}^{(p-1)} & A_{(2p)2}^{(0)} & A_{(2p)3}^{(0)} & \cdots & A_{(2p)(2p)}^{(p-1)} \end{array}$$

3 Jacobi's Method

Define a plane rotation in the (i,j)-plane by

$$P_{ij} = \begin{pmatrix} I_{i-1} & & & & \\ & c & & s & \\ & & I_{j-i-1} & & \\ & -s & & c & \\ & & & & I_{n-j} \end{pmatrix}.$$

Given symmetric matrix A, one can choose plane rotation P_{ij} so that $P_{ij}AP_{ij}^{\mathrm{T}}$ has a zero (i,j) element. Jacobi methods for finding the eigenvalues of A successively compute rotations to annihilate off-diagonal elements of A. The following sequential algorithm is known as the *column-serial* Jacobi method [6]:

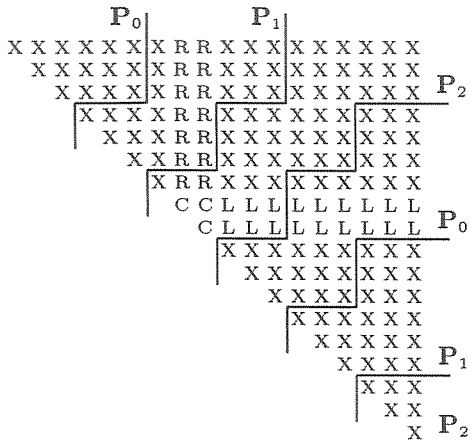

Figure 1: Jacobi's method: annihilating $(i, i+1)$

Algorithm 1

```
do until convergence
   for i=1,...,n-1
      for j=i+1,...,n
         compute P_ij
         update A = P_ij A P_ij^T
```

One iteration of the outer loop annihilates each off-diagonal element exactly once and is called a *sweep*.

To enhance data locality, Algorithm 1 can be reformulated as follows [11,13]:

Algorithm 2

```
do until convergence
   for j=1,...,n/2
      for i=1,...,n-1
         compute P_{i(i+1)}
         update A = P_{i(i+1)} A P_{i(i+1)}^T
         exchange rows and cols i and i+1
```

Notice that only elements of the first super-diagonal are annihilated and as a result only adjacent rows and columns are involved in the application of a plane rotation.

If A is distributed using column-wrapped storage, the application of $P_{i(i+1)}$ from the left results in the work being distributed among the processors. However, applying $P_{i(i+1)}^{\mathrm{T}}$ from the right requires columns i and $i+1$ to be brought together, e.g. on the processor that holds column i. Next, the computation is performed by one processor, while the other processors are idle and hence the work is not balanced among the processors. Moreover, the time complexities of communication and computation are of the same order. The former problem can be overcome by annihilating many super-diagonal elements simultaneously [11]. In this case, communication overhead still stands in the way of efficient utilization of the processors.

Assume the upper triangular part of A is instead distributed using the BHW_2 storage scheme, as illustrated in Figure 1 for $p = 3$ and $h = 3$. This figure also indicates the computation required to annihilate a typical super-diagonal element $(i, i+1)$. Here, Xs are nonzero elements; Cs indicate the elements from which $P_{i(i+1)}$ is computed; and Ls and Rs indicate elements affected by applying $P_{i(i+1)}$ from the left and right, respectively. Once the appropriate processor has computed the rotation and distributed it to all processors, the computation required to update the matrix is distributed among the processors. Whenever a boundary is encountered, a limited amount of additional communication is required.

We chose BHW_2 over BHW_1 since on distributed memory multiprocessors it is important to store data that is involved in a given computation on neighboring processors. The Jacobi method computes plane rotations from submatrices

$$\begin{pmatrix} a_{ii} & a_{i(i+1)} \\ a_{(i+1)i} & a_{(i+1)(i+1)} \end{pmatrix}.$$

Notice that $m = 1$ assigns these elements to $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$ when $i\%h = 0$. If $m = 2$, elements involved in the computation or application of a rotation are always on neighboring processors.

The storage scheme allows further parallelism, details of which can be found in [16].

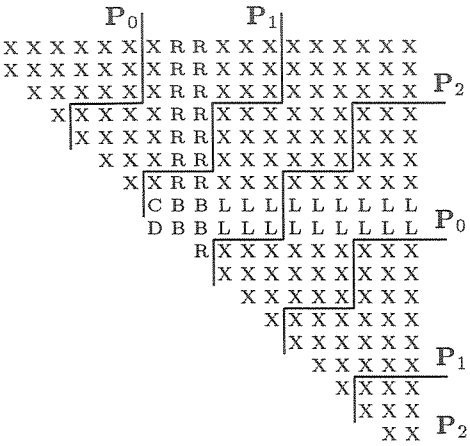

Figure 2: The Implicit QR Algorithm: annihilating $(i+2, i)$

4 The QR Algorithm

The QR decomposition of an $n \times n$ matrix A is given by $A = QR$, where Q is unitary and R is upper triangular. A typical sequential QR algorithm can be described by

Algorithm 3

$A^{(0)} = \hat{Q}^{\mathrm{T}} A \hat{Q}$
`for k=0,1,...`
  $A^{(k)} - s_k I \rightarrow Q^{(k)} R^{(k)}$
  $A^{(k+1)} \leftarrow R^{(k)} Q^{(k)} + s_k I,$

where $\{s_k\}$ is called the shifting sequence, chosen to speed up convergence. The first statement of the algorithm reduces A to upper Hessenberg form, that is $A^{(0)}$ has zeroes below the first subdiagonal. For a detailed description of QR algorithms, see [6].

Each iteration of the loop indexed by k can be implemented by successively computing plane rotations in the $(i+1, 1)$-plane that annihilate the $(i+1, i)$ element of the matrix, as follows:

Algorithm 4

$A^{(k)} = A^{(k)} - s_k I$
`for i=1,...,n-1`
  `compute` $P_{(i+1)i}$ `so that`
    $P_{(i+1)i} A^{(k)}$ `has zero` $(i+1, i)$ `entry`
  `update` $A^{(k)} = P_{(i+1)i} A^{(k)}$
`for i=1,...,n-1`
  `update` $A^{(k)} = A^{(k)} P_{(i+1)i}^{\mathrm{T}}$
$A^{(k+1)} = A^{(k)} + s_k I$

The shift is subtracted explicitly from $A^{(k)}$ and added back to $A^{(k+1)}$. Notice that if $A^{(k)}$ is upper Hesssenberg, so is $A^{(k+1)}$.

An implicitly shifted alternative computes $A^{(k+1)}$ from $A^{(k)}$ as follows:

Algorithm 5

`compute` P_{21} `so that`
  $P_{21}(A^{(k)} - s_k I)$ `has zero` $(2, 1)$ `entry`
$A^{(k)} = P_{21} A^{(k)} P_{21}^{\mathrm{T}}$
`for i=1,...,n-2`
  `compute` $P_{(i+2)(i+1)}$ `so that`
    $P_{(i+2)(i+1)} A^{(k)}$ `has zero` $(i+2, i)$ `entry`
  `update` $A^{(k)} = P_{(i+2)(i+1)} A^{(k)} P_{(i+2)(i+1)}^{\mathrm{T}}$
$A^{(k+1)} = A^{(k)}$

Column-wrapped storage again leads to difficulties in balancing the work during the application of the rotation from the right, as well as excessive communication overhead [5].

If $A^{(k)}$ is stored using BHW_2, the computation required for a given i of the inner loop is illustrated by Figure 2 for $p = 3$ and $h = 3$. In this figure, D is the element to be annihilated; the rotation is computed from C and D; Ls and Rs are elements affected by applying the rotation from the left and right, respectively; and Bs are elements which are affected by applying the rotation from both sides. Once the appropriate processor has computed and distributed the rotation, all processors are involved in updating the matrix. In general, each processor must apply the rotation to $2h$ pairs of elements, except for the processor that computes the rotation, which performs slightly more work. Some additional communication is necessary when a boundary is encountered.

The explicitly and double shifted QR algorithms can be implemented in a similar fashion. Communication overhead can be reduced by pipelining the computation and application of rotations. These issues, as well as how to reduce the original matrix to upper Hessenberg form and how to add deflation schemes, will be discussed in detail in a future report.

5 Conclusion

In the previous sections, we have shown that block hankel-wrapped storage is more appropriate than column-wrapped storage for parallel implementation of the Jacobi method and the nonsymmetric QR algorithm on distributed memory parallel computers. Note that the Jacobi method for symmetric matrices presented in §3 can be easily changed to a Jacobi method for finding the Singular Value Decomposition of an upper triangular matrix. We are currently studying the use of hankel-wrapped storage for other algorithms for finding eigenvalue and singular values, and are pursuing the possibility of developing a portable EISPACK-like package of parallel routines that utilize the storage scheme.

The usefulness of hankel-wrapped storage may not end with eigenvalue algorithms. For example, during Gaussian elimination, storage by elements of rows and columns are distributed among processors, allowing the computation as well as the application of multipliers to occur in parallel.

References

[1] Boley, D., "Solving the Generalized Eigenvalue Problem on a Synchronous Linear Processor Array," *Parallel Computing*, 3, 123-166, 1986

[2] Berry, M. and Sameh, A., "Parallel Algorithms for the Singular Value and Dense Symmetric Eigenvalue Problems," CSRD report No. 761, 1988

[3] Brent, R. and Luk, F., "The Solution of Singular Value and Symmetric Eigenvalue Problems on Multi-processor Arrays," *SIAM J. Sci. Stat. Comput.*, 6, 69-84, 1985.

[4] Dongarra, J.J. and Sorensen, D.C., "A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem," *SIAM J. Sci. Stat. Comput.*, Vol. 8, No. 2, March 1987

[5] Geist, G.A., Ward, R.C., Davis, G.J., and Funderlic, R.E., "Finding Eigenvalues and Eigenvectors of Unsymmetric Matrices using a Hypercube Multiprocessor," unpublished manuscript

[6] Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins Press, 1983

[7] Hockney, R.W. and Jesshope, C.R. *Parallel Computers*, Adam Hilger Ltd., Bristol, 1981

[8] Ipsen, I.C.F and Saad, Y. "The Impact of Parallel Architectures on the Solution of Eigenvalue Problems," *Large Scale Eigenvalue Problems*, J. Cullum and R.A. Willoughby (Ed.), Elsevier Science Publishers, 1986

[9] Ipsen, I.C.F., Saad, Y., and Schultz, M. H., "Complexity of Dense-Linear-System Solution on a Multiprocessor Ring," *Linear Algebra and its Applications*, 77:205-239, 1986

[10] Lo, S.S., Philippe, B., and Sameh, A., " Multiprocessor Algorithm for the Symmetric Tridiagonal Eigenvalue Problem," *SIAM J. Sci. Stat. Comput.*, Vol. 8, No. 2, March 1987

[11] Luk, F.T. and Park, H., "On the Equivalence and Convergence of Parallel Jacobi SVD Algorithms," *Proceedings of SPIE, Advanced Algorithms and Architectures for Signal Processing II*, Vol. 826, 152-159, 1988

[12] Sameh, A., "On Jacobi and Jacobi-Like Algorithms for a Parallel Computer," *Math. Comp.*, vol. 25, pp. 579-590, 1971

[13] Stewart, G.W., "A Jacobi-like Algorithm for Computing the Schur Decomposition of a Non-Hermitian Matrix," *SIAM J. Sci. Stat. Comp.*, B 6, pp. 853-64, 1985

[14] Stewart, G.W., "A Parallel Implementation of the QR Algorithm," Dept. of Comp. Sci., Univ. of MD, TR-1662, May 1986

[15] Van de Geijn, R.A., "Implementing the QR-Algorithm on an Array of Processors," Ph.D. thesis, Dept. of Comp. Sci., Univ. of MD, TR-1897, 1987

[16] Van de Geijn, R. A., "A Novel Storage Scheme for Parallel Jacobi Methods," The University of Texas at Austin, Dept. of Computer Sciences, TR-88-26