A NOVEL STORAGE SCHEME FOR PARALLEL JACOBI METHODS

Robert A. van de Geijn

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

TR-88-26 July 1988

Abstract

A novel storage scheme is presented that allows efficient parallel implementation of the Jacobi method for symmetric matrices by taking advantage of symmetry. For a fixed number of processors, efficiency approaches 100% as the size of the problem is increased.

1. Introduction

In this paper, we present a novel scheme for distributing matrices among the memories of a parallel multiprocessor computer that allows efficient parallel implementation of Jacobi's method for solving the symmetric algebraic eigenvalue problem. For distributed memory parallel computers, the storage method allows the computation to be distributed among the processors while requiring a limited amount of communication. For shared memory multiprocessors, the storage scheme can be used to reduce memory bank conflicts.

Much progress has been made in parallelizing eigenvalue algorithms for symmetric matrices: For tridiagonal matrices, see [DoSo87],[LoPS87]; For full symmetric matrices, Jacobi methods are now very competitive ([BeSa88], [BrLu85], [IpSa86], [LuPa88], [Same71]). Typically, parallel implementations of the Jacobi method either do not take advantage of symmetry (resulting in a doubling of the number of arithmetic operations), or do not result in optimal speedup due to load balancing problems. In §2, we propose the storage scheme. An efficient parallel Jacobi method that does take advantage of symmetry is presented in §3. Complexity analyses indicate the efficiency of the resulting scheme quickly approaches 100% if the number of processors is held constant and the problem size is increased. Practical experience, presented in §4, supports this claim. Further uses for the storage scheme are briefly discussed in the conclusion.

Throughout this note, we assume all matrices, vectors, scalars, and arithmetic to be real. All results can be easily extended to complex matrices and complex arithmetic. We will assume the multiprocessor has p processors, $\mathbf{P}_0, \ldots, \mathbf{P}_{p-1}$, which are connected in a ring so that $\mathbf{P}_i$ is adjacent to $\mathbf{P}_j$ if $|i-j| \,\%p = 1$, where .%. indicates the modulo operator. We further assume that communicating m floating point numbers between neighboring processors requires time $\alpha + m\beta$, while a floating point operation requires time γ.

2. The Storage Scheme

For simplicity, we assume the dimension of matrix A is $n = mhp$ for some integer h and m. Partition matrix A as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1(mp)} \\ A_{21} & A_{22} & \cdots & A_{2(mp)} \\ A_{31} & A_{32} & \cdots & A_{3(mp)} \\ & & & \\ A_{(mp)1} & A_{(mp)2} & \cdots & A_{(mp)(mp)} \end{bmatrix},$$

where $A_{ij} \in \mathbf{R}^{h \times h}$. For $m = 1$, the proposed storage scheme, which we refer to as the *Block Hankel-wrapped* storage scheme, assigns submatrix A_{ij} to processor $\mathbf{P}_{(i+j-2)\%p}$, as illustrated by the superscripts below:

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

It should be noted that this storage scheme is an example of a *skewed storage scheme* for shared memory multiprocessors [HoJe81].

In general, the Block Hankel-wrapped storage scheme for a given m, BHW_m, assigns A_{ij} to $\mathbf{P}_{\lfloor (i+j-2)/m \rfloor \% p}$. In the next section, the case m=2 is used, which assigns submatrix A_{ij} to processor $\mathbf{P}_{\lfloor (i+j-2)/2 \rfloor \% p}$:

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

Justification for using m=2 will be given at the end of the next section.

3. The Jacobi Method

A plane rotation in the (i,j)-plane $(i<j)$ is of the form

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

where $c^2 + s^2 = 1$ and I_k equals the identity matrix of dimension k. Given symmetric matrix A, one can compute a rotation P_{ij} in the (i,j)-plane such that $P_{ij}AP_{ij}^{\mathrm{T}}$ has a zero (i,j) entry. Jacobi methods for finding the eigenvalues of a symmetric matrix successively compute rotations to annihilate off-diagonal elements of A. If the order in which off-diagonal elements are annihilated is chosen properly, the off-diagonal elements of A will converge to zero. One such ordering (known as the column-serial Jacobi method), is given by the ordering (sweep) of plane rotations

$$(1,2), (1,3), \ldots, (1,n), (2,3), \ldots, (2,n), \ldots, (n-1,n) \tag{3.1}$$

Detailed discussions can be found in [GoVL82] and [FoHe60].

Turning now to our parallel implementation, assume the upper triangular part of A is distributed using the BHW_2 storage scheme, as illustrated in Fig. 1 for p=3 and h=3. This figure also indicates the computation required to annihilate a typical element $(i,i+1)$. Here, Xs are nonzero elements of the upper triangular part of A; Cs indicate the elements from which a plane rotation is computed; Ls indicate elements affected by the application of the plane rotation from the left and Rs indicate those affected by the application of the plane rotation from the right. Notice that each processor must update $2h$ pairs of elements of

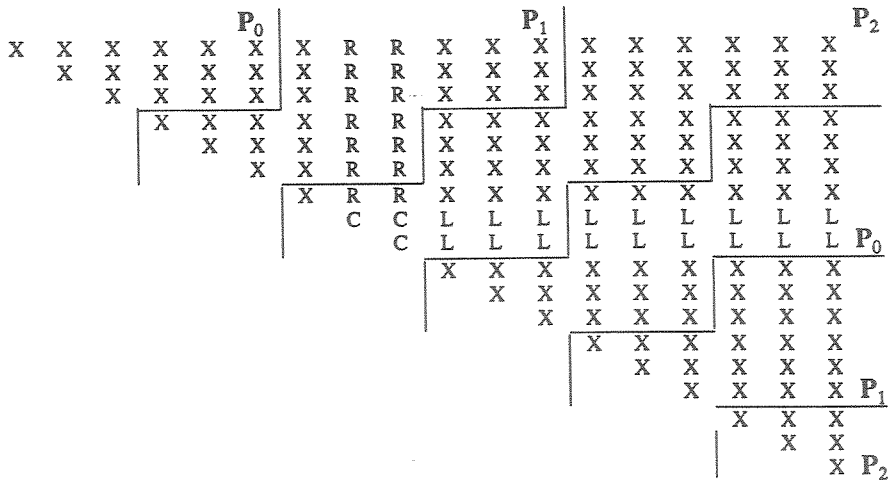

Figure 1: Annihilating $(i,i+1)$

rows and/or columns i and $i+1$. Hence, once the appropriate processor has computed and broadcast the rotation, the processors perform approximately an equal amount of computation.

[Stew85] shows how a sweep of rotations can be implemented while only annihilating elements $(i,i+1)$ by exchanging rows and columns i and $i+1$ once the rotation in the $(i,i+1)$-plane has been computed and applied. The interchange can be combined with P_{ij} by applying $\bar{P}_{ij}$ instead of P_{ij}, where

$$\bar{P}_{ij} = \begin{bmatrix} I_{i-1} & & & & \\ & s & & c & \\ & & I_{j-i-1} & & \\ & c & & -s & \\ & & & & I_{n-j} \end{bmatrix}.$$

The above scheme allows p processors to be used by taking advantage of the inherent parallelism that exists during the application of a particular plane rotation. Pseudo-code driving the processors is given by the following:

Algorithm 1:

Do until convergence

for $j=1,\ldots,n/2$

for $i=1,\ldots,n-1$

if $i\,\%h=0$ send boundary data left

compute $P_{i(i+1)}$ from $\begin{bmatrix} a_{ii} & a_{i(i+1)} \\ * & a_{(i+1)(i+1)} \end{bmatrix}$ on $\mathbb{P}_{(i-1)\%h}$

broadcast $P_{i(i+1)}$

in parallel update A according to $\bar{P}_{i(i+1)}$

if $i\,\%h=0$ send boundary data right

Note that one iteration of the outer loop constitutes a sweep. Whenever i is a multiple of h, all processors must send data to their right neighbor since the data required to compute and apply the rotation do not exist on the same processor. The above parallel scheme implements the column-serial Jacobi method.

Communication overhead can be greatly reduced by also computing several rotations simultaneously, thereby keeping all processors busy during the computation of rotations and increasing the amount of work done between communications. For example, while $\mathbf{P}_0$ computes a rotation to annihilate a_{12}, $\mathbf{P}_1$ can compute a rotation to annihilate $a_{(h+1)(h+2)}$ and $\mathbf{P}_k$ can compute a rotation to annihilate $a_{(kh+1)(kh+2)}$. The task of updating the matrix according to all these rotations can also be performed in parallel. This leads to the following algorithm:

Algorithm 2:

Do until convergence

for $j=1, \ldots, n/2$

Part 1:

for $k=0, \ldots, 2p-1$ simultaneously (on $\mathbf{P}_{k\%p}$)

for $i=1, \ldots, h-1$

compute $P_{(kh+i)(kh+i+1)}$

update $A_{(k+1)(k+1)}$ according to $\bar{P}_{(kh+i)(kh+i+1)}$

distribute all rotations to all processors

in parallel apply all rotations to the remainder of the matrix

Part 2:

send boundary data left

in parallel compute $P_{(kh+h)(kh+h+1)}$ on $\mathbf{P}_{(k-1)\%p}$

distribute all rotations to all processors

in parallel apply all rotations to the remainder of the matrix

send boundary data right

All rotations in Part 1 are computed simultaneously, requiring the submatrices on the diagonal to be updated, after which the rotations are applied to the remaining submatrices. Part 2 deals with the boundaries of the storage scheme, which requires each processor to pass data to the left neighbor before rotations can be computed and applied. After the matrix has been updated, the data is returned. It should be noted that the order in which rotations are computed and applied is in some sense equivalent to the column-serial Jacobi

method [LuPa88].

The time complexity for a given iteration of the loop indexed by j can be obtained as follows: Computing all rotations for $i \neq h$ and updating the blocks on the diagonal appropriately requires time $2(h-1)(8+4(h-2))\gamma$; distributing all rotations to all processors requires time $(p-1)(\alpha+4(h-1)\beta)$ on a ring of processors; updating the remainder of the matrix blocks requires time $2(2p-1)(4(h-1)h)\gamma$; sending the appropriate data to the neighbor to the left when iterate $i=h$ can start requires time $\alpha+n\beta/2$; computing the rotations for $i=h$ requires $2\times 8\gamma$; distributing the rotations requires time $(p-1)(\alpha+4\beta)$; updating the matrix requires time $(4(2p-1)(2h)-8)\gamma$; and returning the updated data to the original processor requires time $\alpha+n\beta/2$. The total time complexity per sweep therefore is

$$T(n,p)=\frac{n}{2}\left[\frac{4n(n-1)+8p}{p}\gamma+2p\alpha+(3+\frac{2}{p})n\beta\right]$$

on a ring of processors. This compares favorably with the sequential time complexity of $T(n,1)=\frac{n}{2}\left[4n(n-1)\right]\gamma$ [GoVL83]. If p is fixed, $\lim_{n\to\infty}\frac{T(n,1)}{T(n,p)}\to p$, indicating that asymptotically the efficiency is 100%.

We can implemented Alg. 1 using the BHW_1 storage scheme. However, some parallelism is lost in Alg. 2 if p is even since then only even numbered processors have access to submatrices on the diagonal, requiring half the processors to idle during the computation of rotations.

4. Numerical Experiments

To test performance, we simulated a distributed memory parallel processor on a Sequent Balance 21000 multiprocessor. The shared memory feature of the Sequent was only used to pass messages between processors. The following table reports the time per sweep in seconds for a straight-forward implementation of the column-cyclic Jacobi method on a single processor (routine SJAC) and an implementation of Alg. 2 (routine PJAC), both coded in C. The number in parentheses reports the speedup attained.

	SJAC	PJAC					
n	p=1	p=2		p=4		p=8	
32	2.6	1.7	(1.5)	1.1	(2.4)	.9	(2.9)
64	19.3	11.1	(1.7)	6.3	(3.1)	4.5	(4.3)
128	153.6	81.7	(1.9)	43.9	(3.5)	27.2	(5.6)
256	1217.9	623.7	(1.9)	345.3	(3.5)	195.4	(6.2)

5. Conclusion

Both theoretically and through numerical experiments, it has been shown that the Block Hankel-wrapped storage scheme allows efficient implementation of the Jacobi method. Further applications of the storage scheme include the efficient parallel implementation of the QR algorithm for nonsymmetric matrices [Vand88]. We are currently pursuing the possibility of developing a portable EISPACK-like package of parallel routines that utilize the Block Hankel-wrapped storage scheme.

References

[BeSa88] Berry, M. and Sameh, A., "Parallel Algorithms for the Singular Value and Dense Symmetric Eigenvalue Problems," CSRD report No. 761, 1988

[BrLu85] Brent, R. and Luk, F., "The Solution of Singular Value and Symmetric Eigenvalue Problems on Multi-processor Arrays," *SIAM J. Sci. Stat. Comp.*, 6, 69-84, 1985.

[DoSo87] Dongarra, J.J. and Sorensen, D.C., "A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem," *SIAM J. Sci. Stat. Comp.*, Vol. 8, No. 2, March 1987

[FoHe60] Forsythe, G.E. and Henrici, P., "The Cyclic Jacobi Method for Computing the Principal values of a complex matrix", *Trans. Amer. Math. Soc.*, 94, 1-23, 1960

[GoVL83] Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins Press, 1983

[HoJe81] Hockney, R.W. and Jesshope, C.R. *Parallel Computers*, Adam Hilger Ltd., Bristol, 1981

[IpSa86] 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 Sci-

ence Publishers, 1986

[LoPS87] Lo, S.S., Philippe, B., and Sameh, A., "A Multiprocessor Algorithm for the Symmetric Tridiagonal Eigenvalue Problem," *SIAM J. Sci. Stat. Comp.*, Vol. 8, No. 2, March 1987

[LuPa88] 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

[Same71] Sameh, A., "On Jacobi and Jacobi-Like Algorithms for a Parallel Computer," *Math. Comp.*, Vol. 25, pp. 579-590, 1971

[Stew85] 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

[Vand88] Van de Geijn, R. A., "Storage Schemes for Parallel Eigenvalue Algorithms," The University of Texas at Austin, Dept. of Computer Sciences, Technical Report