A Parallel Algorithm for Householder Tridiagonalization

Christopher Smith* Bruce Hendrickson† Elizabeth Jessup‡

Abstract

We present a parallel algorithm for reducing a dense symmetric matrix to tridiagonal form. The algorithm employs a square torus–wrap mapping of matrix elements to processors to reduce communication and uses level 3 BLAS routines for efficient numerical kernels. We demonstrate the efficiency of this approach with performance results on the Intel Paragon.

1 Introduction

The standard sequential algorithm for finding eigenvectors and eigenvalues of a dense symmetric matrix A proceeds in three steps. First, the matrix is tridiagonalized with a sequence of symmetric orthogonal transformations to yield a tridiagonal matrix T. Second, the eigenvectors and eigenvalues of T are found. And third, a backtransformation generates the eigenvectors or A from those of T.

This paper describes our parallel tridiagonalization algorithm. In section §2, we review the sequential tridiagonalization algorithm, discuss the difficulties associated with its parallelization, and describe our parallel algorithm. In §3 we present data on the performance of the tridiagonalization code on the Intel Paragon, a mesh connected MIMD multiprocessor. A more complete description of our complete symmetric eigensolver and the performance of its components will be forthcoming in [7].

2 Householder tridiagonalization

The best sequential algorithm for tridiagonalizing a symmetric matrix uses a sequence of Householder transformations. Let A by an $n \times n$ symmetric matrix, and let H_1^T be the Householder transformation that zeros elements in the first column of A below the subdiagonal by pivoting on the subdiagonal element. It is easy to see that $H_1^T A H_1$ also zeros the elements in the first row of A to the right of the superdiagonal. We can then construct H_2 to zero the subdiagonal elements of the second column, and so on. After $n-2$ steps of this process, the matrix A is in tridiagonal form. An efficient sequential algorithm for this calculation follows.

ALGORITHM 2.1 (SEQUENTIAL HOUSEHOLDER TRIDIAGONALIZATION).

For $i = 1 : n-2$

$v =$ **House**$(A(i+1:n, i))$

*Dept. Mech. Eng., Univ. Wisconsin, Madison, WI 53706. smithc@cae.wisc.edu. Smith and Hendrickson were funded by the Applied Mathematical Sciences program, U.S. DOE, Office of Energy Research, and performed research at Sandia National Lab, operated for DOE under contract No. DE-AC04-76DP00789. Appeared in *Proc. 5th SIAM Conf. Appl. Linear Alg.* pp. 361–365, 1994.

†MS 1110, Sandia National Labs, Albuquerque, NM 87185. bahendr@cs.sandia.gov.

‡Dept. Computer Science, Univ. Colorado, Boulder, CO 80309-0430. jessup@cs.colorado.edu. Jessup was funded by DOE contract DE-FG02-92ER25122 and by an NSF National Young Investigator Award.

$v = \sqrt{\frac{2}{v^T v}}v$
$p = A(i+1:n, i+1:n)v$
$w = p - \frac{1}{2}(p^T v)v$
$A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - vw^T - wv^T$

End

The two computationally expensive steps in this algorithm are the matrix–vector multiplication in the generation of p and the rank two update of A. Exploiting the symmetry of A allows us to nearly halve the operations required in the rank 2 update (although it complicates the calculation of Av). The resulting algorithm requires $4n^3/3 + O(n^2)$ flops [4].

Algorithm 2.1 employs only matrix–vector operations, known as level 2 BLAS. On processors with a memory hierarchy, it is generally more efficient to recast algorithms to allow for greater memory reuse. Dongarra, Sorensen and Hammarling [3] have shown how to modify algorithm 2.1 to use matrix–matrix operations or level 3 BLAS. The basic idea is to save the v and w vectors from several steps and use them to update A simultaneously. If we save b pairs of vectors, then we can perform a rank–$2b$ update of A. Postponing updates complicates the generation of p, since the values in the matrix are not always up–to–date. But it allows for higher performance in the outer–product update of A. We note, however, that the generation of p still requires level 2 BLAS.

Besides its intrinsic complexity, there are several features of the tridiagonalization algorithm that make it challenging to parallelize. First, the matrix–vector multiplication in the generation of p makes for a much more complex code than the simple updates required in LU or QR factorization. Second, exploiting symmetry both increases the communication requirements of the algorithm and complicates the matrix–vector multiplication. Finally, the outer–product update required in our parallel algorithm is *not* included in the standard suite of level 3 BLAS. As we will show below, the portion of A owned by a processor may not be symmetric, and there is no level 3 BLAS routine to perform a non–symmetric, outer–product update of a lower triangular matrix.

A critical first step in devising a parallel linear algebra algorithm is deciding which matrix elements are assigned to which processors. We choose to use a *square torus–wrap mapping*, a special case of the torus–wrap mappings which have been successfully applied to a variety of dense linear algebra problems. A detailed discussion of the properties of this mapping can be found in [6] and references therein. One way to describe a square torus mapping is to consider a square tile of size $\sqrt{P} \times \sqrt{P}$, where P is the number of processors in the parallel machine. Each element of the tile is assigned a unique value between 0 and $P-1$. The matrix is now covered with these tiles, and a matrix element is assigned to the processor whose tile number overlays that element. In this way each row and each column of the matrix is owned by $\sqrt{P}$ processors. Also, for a symmetric matrix, the array of elements owned by a processor is identical to the transpose of the array owned by the processor in the transpose location on the tile.

Our reasons for using a square torus–wrap mapping are threefold. First, compared to a column mapping a torus mapping reduces the communication requirements by a factor of $\sqrt{P}$. Second, the square torus allows for an efficient implementation of the matrix–vector multiplication step [5]. And third, a square torus simplifies the exploitation of matrix symmetry. On the other side of the ledger, insisting on a square torus restricts our algorithm to run on a number of (virtual) processors that is a perfect square.

Our parallel algorithm involves several well known communication patterns. First is point–to–point communication in which a processor sends a message to the processor in the transpose location on the tile. Second is broadcast where a processor sends the same message to several other processors. Third is global summation, in which each processor begins with a value and ends with the sum of all the processor's values. The fourth pattern is recursive halving, which is used for element–wise summation of vectors. The Q processors each begin with a vector of length K and upon completion each processor owns a distinct subvector of length K/Q of the summed values. Recursive halving when Q is not a power of two is described in [7]. The final communication pattern is recursive doubling, which is the opposite of recursive halving. Each processor begins with a vector of length K/Q and ends with all the K values.

Our parallel algorithm for Householder tridiagonalization for processor q can now be sketched below. Other approaches to this problem are described in [1, 2, 6, 8].

ALGORITHM 2.2 (PARALLEL BLOCK HOUSEHOLDER TRIDIAGONALIZATION).

(∗ Processor q owns row set α and column set β of symmetric $N \times N$ matrix A. ∗)
(∗ The number of columns in a block is b. ∗)
$B = \lceil (N-2)/b \rceil$ (∗ Compute number of blocks to loop over. ∗)
For $i = 1 : B$
  $V_\alpha = [\emptyset]$; $W_\alpha = [\emptyset]$; $V_\beta = [\emptyset]$; $W_\beta = [\emptyset]$
  $s = (i-1)b + 1$ (∗ First column in block. ∗)
  $t = \min(N-2, s+b-1)$ (∗ Last column in block. ∗)
  For $c = s : t$
    If $c \in \alpha$ **Then** $\alpha = \alpha \setminus \{c\}$ (∗ Remove c from active rows. ∗)
    (∗ BROADCAST AND GENERATE NEXT HOUSEHOLDER VECTOR. ∗)
    If $c \in \beta$ **Then**
      $v_\alpha = A(\alpha, c)$
      Broadcast v_α to processors sharing rows α
    Else Receive v_α
    Compute my contribution to $v^T v$
    Sum contributions to form $\gamma = v^T v$, and append $v(c+1)$ to communication
    $\tau = 2(\gamma + |v(c+1)|\sqrt{\gamma})$
    If $c+1 \in \alpha$ **Then** (∗ Update top of c to form v. ∗)
      $v(c+1) = v(c+1) + \mathrm{sign}(v(c+1))\sqrt{\gamma}$
      If $c \in \beta$ **Then** (∗ Compute subdiagonal element of T ∗)
        $A(c+1, c) = A(c+1, c) + \mathrm{sign}(A(c+1, c))\sqrt{\gamma}$
    If $c \in \beta$ **Then** $\beta = \beta \setminus \{c\}$ (∗ Remove c from active columns. ∗)
    (∗ GET ELEMENTS OF v^T TO THE CORRECT PROCESSORS. ∗)
    Send v_α to transpose processor
    Receive v_β from transpose processor
    $V_\alpha = [V_\alpha \;\; v_\alpha]$; $V_\beta = [V_\beta \;\; v_\beta]$
    (∗ COMPUTE $p = Av$. ∗)
    $L(\alpha, \beta) =$ Lower triangular portion of $A(\alpha, \beta)$
    $x_\alpha^q = L(\alpha, \beta) v_\beta$ (∗ Without diagonal of L. ∗)
    $y_\beta^q = v_\alpha^T L(\alpha, \beta)$ (∗ With diagonal of L. ∗)
    Send y_β to transpose processor
    Receive y_α from transpose processor
    $p_\alpha = x_\alpha + y_\alpha$
    Recursive halve p_α within rows yielding p_η

(* UPDATE p WITH OUTER PRODUCTS NOT YET APPLIED TO A. *)
$\delta^q = V_\eta^T v_\eta$
$\epsilon^q = W_\eta^T v_\eta$
Sum δ^q and ϵ^q among all processors yielding δ and ϵ
$\zeta = \eta \cap (t+1 : N)$ (* Just indices outside block. *)
$p_\zeta = p_\zeta - \epsilon V_\zeta - \delta W_\zeta$
(* COMPUTE w. *)
$(p^T v)^q = p_\eta^T v_\eta$
Sum $(p^T v)^q$ among all processors yielding $p^T v$
$w_\eta = \frac{2}{\tau}[p_\eta - \frac{p^T v}{\tau} v_\eta]$
Recursive double w_η within rows yielding w_α
Send w_α to transpose processor
Receive w_β from transpose processor
$W_\alpha = [W_\alpha \;\; w_\alpha]$; $W_\beta = [W_\beta \;\; w_\beta]$
(* UPDATE REMAINDER OF BLOCK. *)
$\mu = \beta \cap (c+1 : t)$ (* Columns within current block. *)
$A_{\alpha,\mu} = A_{\alpha,\mu} - v_\alpha w_\mu^T - w_\alpha v_\mu^T$
End
(* UPDATE REMAINDER OF MATRIX. *)
$L(\alpha,\beta) = L(\alpha,\beta) - V_\alpha W_\beta^T - W_\alpha V_\beta^T$
End

Several aspects of this algorithm are worth noting and receive further attention in [7]. First, as mentioned above, the outer–product update is nonsymmetric since the set of row indices owned by a processor will in general be different from its column indices. We perform the update by covering the lower triangular matrix with a sequence of rectangular panels, and then using a standard rectangular outer product update on each panel. The full rank–$2b$ update of a panel can be performed with a single call to the level 3 BLAS routine dgemm. Second, there is only $O(PN)$ redundant numerical computation and since the computation is well balanced, the numerical operations should scale as $\frac{4}{3}N^3/P + O(N^2)$. Third, a careful analysis reveals that the total volume of all the messages per processor is $3N^2/\sqrt{P} + O(N)$. Thus, by the simple metric of computation to communication volume, the algorithm should scale well as long as N^2 is proportional to P; that is, the number of matrix elements per processor remains constant.

3 Results

We implemented our algorithm on the Intel Paragon, a mesh connected MIMD computer at Sandia's Massively Parallel Computing Research Laboratory. Our results were obtained under the SUNMOS operating system.

We first ran a sequence of factorizations of a 4000×4000 matrix on 64 processors in which we varied the block size in the outer–product update operation. The run times are shown in the left plot of Fig. 1. As expected, moving from a level 2 to a level 3 formulation improves the performance of the code. However, this improvement is limited by the necessity of a level 2 operation in the generation of p, and by the lack of the appropriate level 3 update routine. The curve flattens out beyond a block size of 4, and further increase has only a small impact on performance.

Next we fixed the block size at 7 and ran a sequence of problems on different numbers of processors. We scaled the problem sizes to keep the memory requirements per processor

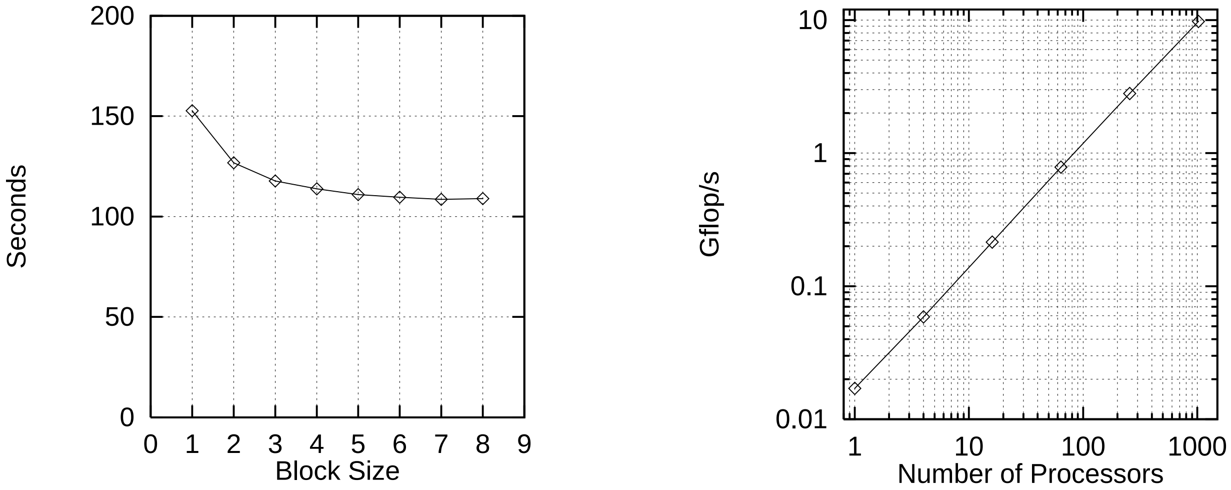

Fig. 1. *Performance of Tridiagonalization code on the Paragon.*

constant. Each processor began the computation with a 500×500 double precision submatrix, so for example, the matrix for 1024 processors was of size $16,000$. In the right plot of Fig. 1 we show Mflop/s performance for these problems, assuming exactly $4n^3/3$ flops are required. The calculation is scaling moderately well; the Mflops/s per processor decrease from 17.08 on a single processor to 9.55 on 1024, for a scaled efficiency of about 56%. Of course, a single processor code could be written without the overhead required in our parallel version and would obtain somewhat better performance. Our performance scales sublinearly due to the high communication volume in the algorithm and the contention for wires that is inevitable with a mesh connected machine. More results and further discussion can be found in [7].

4 Acknowledgements

We are indebted to Xiaobai Sun and Dave Womble for helpful discussions about the tridiagonalization algorithm. We are also indebted to Robert van de Geijn for assistance with the level 3 BLAS and for providing preliminary communication routines on the Paragon.

References

[1] C. Bischof, M. Marques, and X. Sun, *Parallel band reduction and tridiagonalization*, in Proc. 6th SIAM Conf. Parallel Proc. Sci. Comput., Norfolk, Virginia, March 1993, pp. 383–390.

[2] J. Dongarra and R. van de Geijn, *Reduction to condensed form on distributed memory architectures*, Parallel Comput., 18 (1982), pp. 973–982.

[3] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling, *Block reduction of matrices to condensed forms for eigenvalue computations*, J. Comp. Appl. Math, 27 (1989), pp. 215–227.

[4] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 1989. Second edition.

[5] B. Hendrickson, R. Leland, and S. Plimpton, *An efficient parallel algorithm for matrix–vector multiplication*, Int. J. High Speed Comput., (1994). To appear.

[6] B. Hendrickson and D. Womble, *The torus–wrap mapping for dense matrix calculations on massively parallel computers*, SIAM J. Sci. Comput., (1994). To appear.

[7] C. Smith, B. Hendrickson, and E. Jessup, *A parallel eigensolver for dense symmetric matrices.* In preparation.

[8] D. Walker. Personal communication.