On the Design of a Tridiagonalization Routine for Banded Matrices*

Christian Bischof[†] Xiaobai Sun[‡]

Abstract

This paper discusses scalability and data layout issues arising in the development of a parallel algorithm for reducing a banded matrix to tridiagonal form. As it turns out, balancing the memory and computational complexity of the reduction of the matrix and the accumulation of the associated orthogonal matrix is the key to scalability and sustained performance.

1 Introduction

Reduction to tridiagonal form is a major step in eigenvalue computations for symmetric matrices. If the matrix is full, the conventional Householder tridiagonalization approach [7, p. 276] or block variants thereof [6] is the usual method of choice. Another approach is the successive bandreduction (SBR) approach [4], which in its simplest form first reduces a matrix to narrow banded form and from there to tridiagonal form. As is shown in [2], such an approach can also advantageously used for a "rank-revealing tridiagonalization" as it is, for example, required, in the invariant subspace decomposition approach (ISDA) [10] for the solution of the symmetric eigenproblem.

In addition to the SBR approach, the reduction of a banded matrix to tridiagonal form arises in the banded eigenvalue problem, and as the final step in block Lanczos methods for the solution of large sparse eigenvalue problems. A banded variant of the ISDA approach [5] also requires the ability to reduce the bandwidth of banded matrices, with tridiagonalization being one particular approach to doing so.

The reduction of a full $n \times n$ matrix requires roughly $\frac{4}{3}n^3$ flops for the reduction of A and $2n^3$ flops for the accumulation of the orthogonal matrix Q on the fly, or alternatively, the application of these transformations to another matrix. In contrast, employing the algorithm developed independently by Muraka and Horikoshi [11] and Lang [9], the reduction of an $n \times n$ matrix of bandwidth b takes $6bn(n-1)$ flops, and the accumulation of the associated transformation matrix Q on the fly requires $2n^2(n-1)$ flops. Lang's work was motivated by the search for a parallel algorithm for reducing a banded matrix A to banded form, but it did not address the issue of accumulating Q. In this paper, we discuss issues arising in the development of a parallel algorithm to reduce A and accumulate Q.

*This work was supported by the Applied and Computational Mathematics Program, Advanced Research Projects Agency, under contract DM28E04120, and by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, bischof@mcs.anl.gov

‡Department of Computer Science, Duke University, Durham, NC 27708-0129, xiaobai@cs.duke.edu

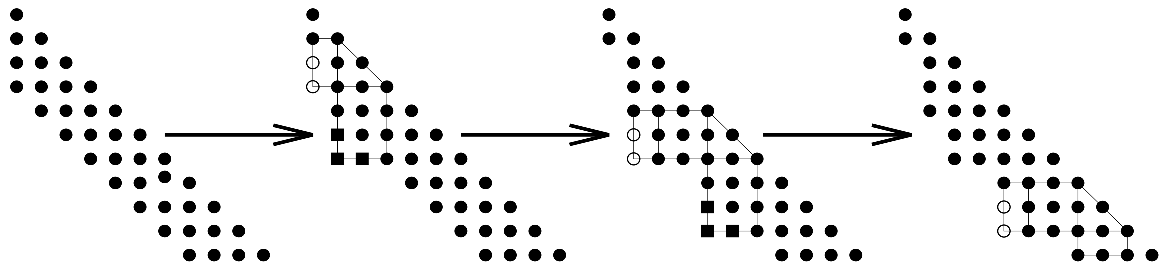

FIG. 1. *First Sweep*

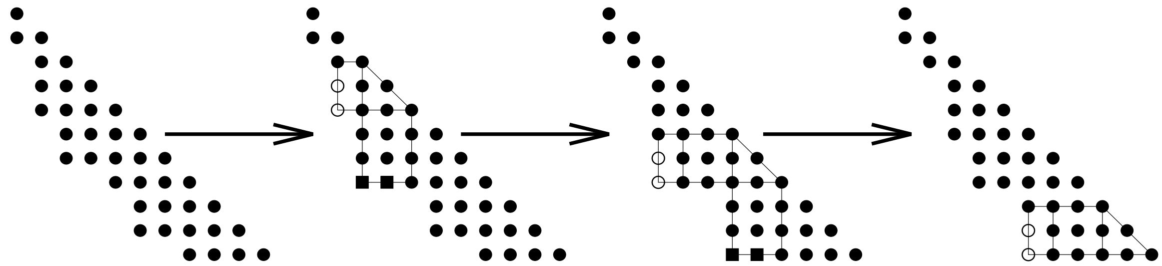

FIG. 2. *Second Sweep*

2 Bandreduction and Bulge Chasing

Let A be an $n \times n$ matrix of semibandwidth b. Employing Lang's [9] and Muraka and Horikoshi's [11] algorithm, the reduction of A to tridiagonal form proceeds in "sweeps", where each sweep consists of the reduction of a column in the original band of A to tridiagonal form, and the chasing of the "bulges' one column down along the diagonal. The first sweep is depicted on a small example in Figure 1. The first bulge is generated by applying on A the Householder transformation to reduce the bandwidth to 1 in the first column. The first column of the bulge is then eliminated by a following transformation, which results in the next bulge, and so on. In the picture, an empty circle denotes a matrix element eliminated by the current Householder transformation, and a filled square denotes a fill-entry generated by the current transformation.

The second sweep, eliminating the elements below the first subdiagonal in column two, is shown in Figure 2.

Note the similarity of the two figures. Apart from the fact that the entries in the second block subdiagonal were entry in the first sweep (and only in this one), the second sweep essentially proceeds like the first one, with all bulges and transformations shifted diagonally to the right by one.

The fundamental operation units in these reduction and chasing steps are shown in Figure 3. There are four tasks to be accomplished:

HH: The generation of a Householder vector to reduce a vector of length b to a multiple of the first canonical unit vector. This step requires roughly $5b$ flops.

PRE: The application of a Householder matrix $H = I - uu^T$ from the left to a $b \times (b-1)$ matrix. This step requires $4b(b-1)$ flops, and is not required in the very first orthogonal reduction in a sweep.

SYM: The symmetric application of H to a symmetric $b \times b$ matrix. This step requires $4b^2$ flops.

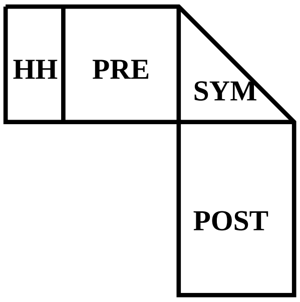

FIG. 3. *Fundamental Kernel in the Tridiagonalization of a Banded Matrix*

POST: The application of H to a $b \times b$ matrix from the right. This step requires $4b^2$ flops.

As there are roughly $(n-j)/b$ "bulges" appearing in the jth sweep, the flop count of the jth sweep is about $12b^2 * (n-j)/b$, and the overall complexity of reducing A to tridiagonal form is $6bn(n-1)$ flops.

When we want to accumulate these orthogonal transformations or apply them to another matrix Q on the fly, each orthogonal reduction in a sweep induces the application of a Householder matrix of order b to a matrix of size $n \times b$, requiring $4nb$ flops. Note that all Householder transformations generated in the jth sweep involve *disjoint sets of columns* of Q. Overall, the complexity of applying the orthogonal transformations on the fly is $2n^2(n-1)$ flops.

3 Parallel Reduction of A to Tridiagonal Form

In most applications, we expect the bandwidth b to be small compared to the problem size n. That is, if we were to choose a data layout that led to a physical partitioning of any of the three $b \times b$ blocks in Figure 3, the PRE, SYM, or POST steps would require communication. This is undesirable as each of these steps is only $4b^2$ flops to start with, and hence the potential for masking communication through computation is very limited.

Therefore, the layout that suggests itself for A is a 1-D block torus wrap, where each block of A is a $2b \times b$ slice, containing the data for the SYM and POST operation of the current Householder reduction, and the HH and PRE operations of the next Householder reduction to be generated in a sweep. The only exception is the first processor which contains one more column. Such a block column of A can be conveniently stored in LAPACK packed format [1], allowing for the use of BLAS-2 kernels. Preliminary performance results of the serial version of the resulting band reduction algorithm are presented in [3]. Assuming that A is distributed over p processors, each processor contains roughly $r := n/(b * p)$ blocks columns.

After the completion of a sweep, the desire to have block columns associated with every bulge chasing step local on a processor necessitates that each process sends the first column of all of its blocks to its left neighbor and receives a set of columns to be added to its blocks from his right neighbor in the processor ring.

Every time of the forward shifting of A by one column can be done by data moving among processors, $2b$ for each b-columns block, and followed by data moving with each processor, $2b(b-1)$ for each b-columns block. To reduce the local memory traffic, we associate a buffer of size $2b \times h$, say, with every block column. Thus, we move data locally only every h steps. In most cases, the extra memory required for the buffer will be well worth the savings in memory moves, in particular on cache-based architectures.

Let us now consider the issue of starting up the reduction of A. The obvious approach would proceed as follows:

Processor i: Generate Householder vector, send message containing Householder vector to processor $i+1(\text{mod} p)$, complete PRE step.

Processor $i+1(\textbf{mod}p)$: Receive Householder vector, complete POST step, generate next Householder vector, send message containing Householder vector to processor $i+2(\text{mod} p)$, complete SYM defined by Householder vector from processor i and PRE step defined by the locally generated Householder vector.

Note that PRE and SYMM steps are delayed until all of the desired Householder vectors are generated. The delay between receiving a Householder vector and sending out the next one is $4b^2$ flops. In fact, the start-up time can be further reduced by half. To generate a Householder vector, we only need to update the *first* column in the POST block, we can effect this in $2b^2+2b$ flops, and the rest of the POST update can be delayed as PRE and SYMM. With this modification, the startup time until all processor are busy is roughly

$$p*(\text{time}(2(b^2+b)\text{flops})+\text{time}(\text{message of length } b))$$

Also note that, once a Householder vector has been sent, processors can complete local PRE, POST, and SYM operations completely independently. The data dependency is thus significantly improved.

By "computing-ahead" with Householder vectors as fast as we can, we are both likely to overlap communication and computation after the first round, and, at the end of a sweep, we are likely to be in the situation where all nodes are still busy completing previous updates. If we give high priority to the first column of a SYM update, and all "first columns" have been updated, we can use the capability of MPI [8] to send "scattered" vectors with equal stride to shift all leading columns one to the left while the nodes are still busy computing, thus masking this computational cost. Here we assume that a processor can send and receive messages while computing, which is true on recent multiprocessors computers.

To summarize, the key ingredients of the reduction for A are:

- a compute-ahead strategy to generate orthogonal transformations within one sweep as quickly as possible,
- a left-shift of the data after every step, to maintain the local nature for next sweep,
- the use of an internal buffer to save on internal memory moves, and
- the use of the LAPACK packed storage scheme to get good per-node computational performance.

Note that this approach also scales well if we consider the bandwidth b and the number of local blocks r fixed, and n grows. This situation is typical, as the bandwidth is usually dictated by "local" considerations in many matrices arising in numerical PDE solutions, or by blocking considerations in the SBR approach [3].

4 Accumulating the Orthogonal Transformations

We allocate separate processors for A and Q for reasons to be seen. The total operation count for the reduction of A is $6bn(n-1)$ flops, whereas the application of the orthogonal transformations to a matrix Q invoves $2n^2(n-1)$ flops. Storing A requires $2(b+h)*n$

words of memory, where h is the buffer size, whereas Q requires n^2 words of memory. These numbers clearly suggest that we have to dedicate many more processors to the application of the orthogonal transformations to Q than to the tridiagonalization of A. Since all updates of Q assorted with one particular band reduction sweep can be done in parallel, and most of the communication in the reduction of A can be hidden, we can use flop counts as a sensible approximation of time. Then the fact that the reduction of A should generate Householder vectors fast enough to keep the processor devoted to Q busy can be expressed as

$$\frac{\text{flops\_for\_A}}{\text{nodes\_for\_A}} \leq \frac{\text{flops\_for\_Q}}{\text{nodes\_for\_Q}}.$$

If A is distributed over p processors, and Q over q other processors, this implies

$$q \leq \frac{n}{3b}p. \tag{1}$$

To also balance memory requirements, that is, to ensure that each processor devoted to A and to Q allocates the same amount of memory to "its" share of the problem, requires

$$\frac{2(b+h)*n}{p} = \frac{n^2}{q}. \tag{2}$$

Assuming that work between nodes is balanced, i.e., equation (1) holds with equality, (2) simplifies to $h = b/2$, that is, one-third of the memory devoted to solving A is used as buffer space.

Since we have to update block columns of width b of Q with Householder transformations from the right, it would be natural to think of using 1-D row wrap of Q in order to totally avoid communications among phsical column blocks. This appealing idea is unfortunately not scalable. With such a data layout, the number of Q's rows in one processor would be $n/q = 3b/p$ according to (1). This becomes impossible when b remains small and n and p grow.

A 2-D distribution of Q, wrapped at least in the column direction, proves to be a feasible choice. The distribution of rows can be left open for strategies to meet any additional requirement or provide convenience in applications. Taking b columns of Q a block column as we did with A, we can then map Q onto a processor array of size $\frac{n}{3b} \times p$, say. Note that we cannot choose an unwrapped distribution, as the ith column of Q is only involved in sweeps $1, \ldots i$. As a result, successive blocks of Q will be on different processors, so that only in one out of b times the application of a Householder transformation to Q does not require any communication. Shifting forward Q, as we did with A, would result in a reduction in communication but complicate the update algorithm since the size and shape of Q is fixed but the area to update becomes smaller as the reduction proceeds. For A, each time of data moving forward involves putting away only two numbers(a diagonal element and a subdiagonal element).

We summarize:

- To balance the work between the update of A and Q, the number of processors p and q that house A and Q, respectively, should satisfy $q \leq \frac{n}{3b}p$.
- Up to one-third of the memory for A can be devoted to buffer space for message passing, without disturbing the work balance.
- Q has to be distributed with a 2-D block scheme, wrapped in the column direction. The row partition is irrelevant.

We mention that in order to save communication, the reduction of A could generate a few rounds of Householder vectors, at least in later stages of the computation, before sending them to the processors housing Q, which, in turn, would allow the updates of Q to proceed with fewer communication calls.

Lastly, we mention that the scheme discussed here does not incorporate the blocking of the orthogonal updates of Q suggested in [3]. The reason is that the blocking scheme requires us to complete nb full sweeps (where nb is the desired block size) before we can begin with the update of Q. It is not clear whether the resulting increase in performance makes up for the greatly increased startup delay.

5 Conclusions

In this paper, we outlined the rationale behind a parallel algorithm for both reducing a banded matrix to tridiagonal form and for updating the associated transformation matrix on the fly. Driven by a simple timing model for the updates of A and Q, we developed an approach which both balances the work and memory between updating A and Q, and has a relatively simple control logic. An implementation of this approach is in progress.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK User's Guide*, SIAM, Philadelphia, 1992.

[2] C. Bischof, S. Huss-Lederman, X. Sun, and A. Tsao, *The PRISM project: Infrastructure and algorithms for parallel eigensolvers*, in Proceedings of the Scalable Parallel Libraries Conference, Washington, DC, 1994, IEEE Computer Society, pp. 123–131.

[3] C. Bischof, X. Sun, and B. Lang, *Parallel tridiagonalization through two-step band reduction*, in Proceedings of Scalable High Performance Computing Conference, IEEE Computer Society Press, May 1994, pp. 23–27.

[4] C. H. Bischof and X. Sun, *A framework for band reduction and tridiagonalization of symmetric matrices*, Tech. Rep. MCS–P298–0392, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[5] C. H. Bischof, X. Sun, A. Tsao, and T. Turnbull, *A study of the invariant subspace decomposition algorithm for banded symmetric matrices*, in Proceedings of the Fifth SIAM Conference on Applied Linear Algebra, J. G. Lewis, ed., 1994, pp. 321–325.

[6] J. J. Dongarra, S. J. Hammarling, and D. C. Sorensen, *Block reduction of matrices to condensed form for eigenvalue computations*, Tech. Rep. MCS–TM–99, Mathematics and Computer Science Division, Argonne National Laboratory, September 1987.

[7] G. H. Golub and C. F. V. Loan, *Matrix Computations*, The Johns Hopkins University Press, 1983.

[8] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI – Portable Parallel Programming with the Message Passing Interface*, MIT Press, 1994.

[9] B. Lang, *A parallel algorithm for reducing symmetric matrices banded matrices to tridiagonal form*, SIAM Journal on Scientific Computing, 14 (1993), pp. 1320–1338.

[10] S. Lederman, A. Tsao, and T. Turnbull, *A parallelizable eigensolver for real diagonalizable matrices with real eigenvalues*, Tech. Rep. TR-91-042, Supercomputing Research Center, Institute for Defense Analysis, Bowie, MD, 1991.

[11] K. Muraka and K. Horikoshi, *A new method for the tridiagonalization of the symmetric band matrix*, Information Processing in Japan, 15 (1975), pp. 108–112.