A NEW PARALLEL CHASING ALGORITHM FOR TRANSFORMING ARROWHEAD MATRICES TO TRIDIAGONAL FORM[1]

SUELY OLIVEIRA[2]

ABSTRACT. Rutishauser, Gragg and Harrod and finally H.Y. Zha used the same class of chasing algorithms for transforming arrowhead matrices to tridiagonal form. Using a graphical theoretical approach, we propose a new chasing algorithm. Although this algorithm has the same sequential computational complexity and backward error properties as the old algorithms, it is better suited for a pipelined approach. The parallel algorithm for this new chasing method is described, with performance results on the Paragon and nCUBE. Comparison results between the old and the new algorithms are also presented.

Keywords: arrowhead matrices, chasing algorithms, pipeline algorithms
Primary: 65F15; Secondary: 68R10, 65F50.

1. Introduction

Chasing algorithms are commonly used to find eigenvalues of tridiagonal matrices [4, §8.2]. They can also be used to transform matrices to tridiagonal form, such as Rutishauser's algorithm for tridiagonalizing banded matrices [6], Gragg and Harrod's improved version of it, and recently Zha's algorithm for tridiagonalizing arrowhead matrices [9]. All of these methods use the *standard chasing step* of [4, §8.2]. We will refer to them as the standard or Zha's algorithm. Using the graph representation of matrices we will show that they correspond to chasing edges of the triangles representing the current entries of the matrix. In this paper, we develop a new algorithm for tridiagonalizing arrowhead matrices based on a *new chasing step.* Unlike the standard chasing step, the new one uses two additional entries outside the tridiagonal band, and chases nodes instead of edges. The new algorithm can be implemented on parallel architectures using a pipeline approach more efficiently than the ones based on the standard chasing step.

In the remainder of the paper, a graphical representation of the structure of symmetric matrices is used. The graph of a symmetric $n \times n$ matrix A consists of nodes 1, 2, ... , n with an edge between nodes i and j if and only if $a_{ij} \neq 0$.

The structure of the paper is as follows. Section 2 describes chasing algorithms in general. Section 3 describes the new chasing step. Section 4 describes the algorithm to transform arrowhead matrices to tridiagonal form. Section 5 describes the parallel algorithm and optimal data partitioning for this algorithm.

[1]This research is supported by NSF grant ASC 9528912 and a Texas A&M University Interdisciplinary Research Initiative Award.

[2]Department of Computer Science, Texas A&M University, College Station, TX 77843.
email: suely@cs.tamu.edu.

2. Standard Chasing algorithms

Chasing algorithms are standard ways of implementing the QR algorithm for eigenvalue computations [4, §7.4.2, §8.2.3]. These are done using Givens rotations, which are matrices which are equal to the identity matrix except for a 2×2 submatrix. Givens rotations have the form

$$\begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c & \dots & s & & & \\ & & & \vdots & \ddots & \vdots & & & \\ & & & -s & \dots & c & & & \\ & & & & & & 1 & & \\ & & & & & & & \ddots & \\ & & & & & & & & 1 \end{bmatrix}.$$

where $c^2 + s^2 = 1$. Givens matrices G are orthogonal ($G^T G = I$), and so algorithms based on Givens matrices usually have very good numerical stability properties. A Givens operation on a matrix A is the operation of computing $A' \leftarrow G^T A G$ for a suitable Givens matrix G. Note that the Givens operation preserves the symmetry and the eigenvalues of A. To compute the eigenvalues of a general symmetric matrix economically, one first transforms it to tridiagonal form. Then it is easier to calculate the eigenvalue of the tridiagonal matrix by using by using the QR algorithm, for example. The QR algorithm is an algorithm which can be implemented using chasing approaches. If the matrix is an arrowhead matrix, we may first use chasing algorithms to transform it into tridiagonal form.

In [6] Rutishauser described a algorithms for changing pentadiagonal matrices into tridiagonal forms which uses an edge chasing approach. In [8] Huffel and Park discusses the parallel implementation of four transformation algorithms for reducing a bidiagonal matrix to tridiagonal form. They use the same approach as Zha's [9], namely edge chasing. The existence of our new chasing algorithm was discovered by analyzing the graph structure of the old chasing step. This graph approach for analyzing Givens rotations was first used in [7].

The numerical computations performed in the *standard chasing step* are, for the 4×4 matrix containing the entries that are changed by the step,

$$\begin{bmatrix} \alpha' & \theta' & 0 & 0 \\ \theta' & \beta' & \lambda' & \nu' \\ 0 & \lambda' & \delta' & \eta' \\ 0 & \nu' & \eta' & \gamma' \end{bmatrix} = \begin{bmatrix} 1 & & & \\ & c & s & \\ & -s & c & \\ & & & 1 \end{bmatrix} \begin{bmatrix} \alpha & \theta & \mu & 0 \\ \theta & \beta & \lambda & 0 \\ \mu & \lambda & \delta & \eta \\ 0 & 0 & \eta & \gamma \end{bmatrix} \begin{bmatrix} 1 & & & \\ & c & -s & \\ & s & c & \\ & & & 1 \end{bmatrix}. \tag{1}$$

Chasing algorithms can also be represented graphically in terms of their effect on the graph of a symmetric matrix. In Figure 1, the Greek letters at the nodes of the graph indicate the values of the diagonal entries in the matrix while the Greek letters at the edges of the graph correspond to the off-diagonal entries. The value at node i represents a_{ii}, and the value associated with edge $i \sim j$ is $a_{ij} = a_{ji}$. A dark edge between nodes i and j indicate that the Givens operation is applied to rows and columns i and j. A dashed line between nodes i and j indicates that before the Givens operation, $a_{ij} = 0$, but after the operation $a'_{ij} \neq 0$. If the edge between nodes i and j is marked with a cross ($\times$) then it indicates that $a_{ij} \neq 0$ before the Givens operation, but $a'_{ij} = 0$ after the Givens operation. The standard chasing step is illustrated in Figure 1(a).

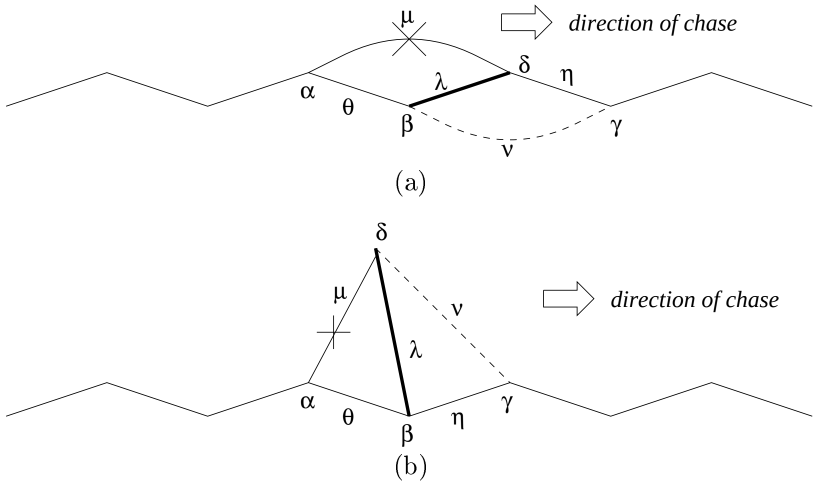

FIGURE 1. Chasing algorithms: (a) standard (b) new

In matrix notation, for transforming an arrowhead matrix into tridiagonal form, chasing steps are applied to the original matrix, creating and annihilating entries in the way illustrated by the 6×6 example in Figure 2. In Figure 2, a "$*$" indicates a nonzero entry, "+" indicates a newly created nonzero entry, and "0" indicates an entry that has just been made zero. Note that at the end of Figure 2, the matrix is made tridiagonal by swapping rows and columns 1 and 2. This method of tridiagonalizing an arrowhead matrix is Zha's chasing algorithm [9]. It is also illustrated graphically in Figure 3.

3. New Chasing Step

Here a new chasing step is used to transform an arrowhead matrix to a tridiagonal matrix. This chasing step is illustrated graphically in Figure 1(b). The triangle shown here will be referred later as a "bulge". In terms of the 4×4 matrix of entries α to ν, the effect of the new chasing step is to compute the new values of α' to ν':

$$\begin{bmatrix} \alpha' & \theta' & 0 & 0 \\ \theta' & \beta' & \eta' & \lambda' \\ 0 & \eta' & \gamma' & \nu' \\ 0 & \lambda' & \nu' & \delta' \end{bmatrix} = \begin{bmatrix} 1 & & & \\ & c & & s \\ & & 1 & \\ & -s & & c \end{bmatrix} \begin{bmatrix} \alpha & \theta & 0 & \mu \\ \theta & \beta & \eta & \lambda \\ 0 & \eta & \gamma & 0 \\ \mu & \lambda & 0 & \delta \end{bmatrix} \begin{bmatrix} 1 & & & \\ & c & & -s \\ & & 1 & \\ & s & & c \end{bmatrix}.$$

That is,

$$\begin{aligned} \alpha' = \alpha, \qquad \gamma' &= \gamma, \\ \beta' &= c^2\beta + 2cs\lambda + s^2\delta, \\ \lambda' &= (c^2 - s^2)\lambda + cs(\delta - \beta), \\ \delta' &= s^2\beta - 2cs\lambda + c^2\delta, \\ \theta' = c\theta + s\mu, \qquad \eta' &= c\eta, \qquad \nu' = -s\eta \end{aligned} \tag{2}$$

$$
\begin{bmatrix} * & * & * & * & * & * \\ * & * & & & & \\ * & & * & & & \\ * & & & * & & \\ * & & & & * & \\ * & & & & & * \end{bmatrix}
\rightarrow
\begin{bmatrix} * & * & * & * & * & 0 \\ * & * & & & & \\ * & & * & & & \\ * & & & * & & \\ * & & & & * & + \\ 0 & & & & + & * \end{bmatrix}
\rightarrow
\begin{bmatrix} * & * & * & * & 0 & \\ * & * & & & & \\ * & & * & & & \\ * & & & * & + & + \\ 0 & & & + & * & * \\ & & & + & * & * \end{bmatrix}
\rightarrow
$$

$$
\begin{bmatrix} * & * & * & * & & \\ * & * & & & & \\ * & & * & & & \\ * & & & * & * & \\ & & & * & * & * \\ & & & & * & * \end{bmatrix}
\rightarrow
\begin{bmatrix} * & * & * & 0 & & \\ * & * & & & & \\ * & & * & + & + & \\ 0 & & + & * & * & \\ & & + & * & * & * \\ & & & & * & * \end{bmatrix}
\rightarrow
\begin{bmatrix} * & * & * & & & \\ * & * & & & & \\ * & & * & * & & \\ & & * & * & * & + \\ & & & * & * & * \\ & & & + & * & * \end{bmatrix}
\rightarrow
$$

$$
\begin{bmatrix} * & * & * & & & \\ * & * & & & & \\ * & & * & * & & \\ & & * & * & * & \\ & & & * & * & * \\ & & & & * & * \end{bmatrix}
$$

FIGURE 2. Description of Zha's chasing algorithm for a 6×6 matrix

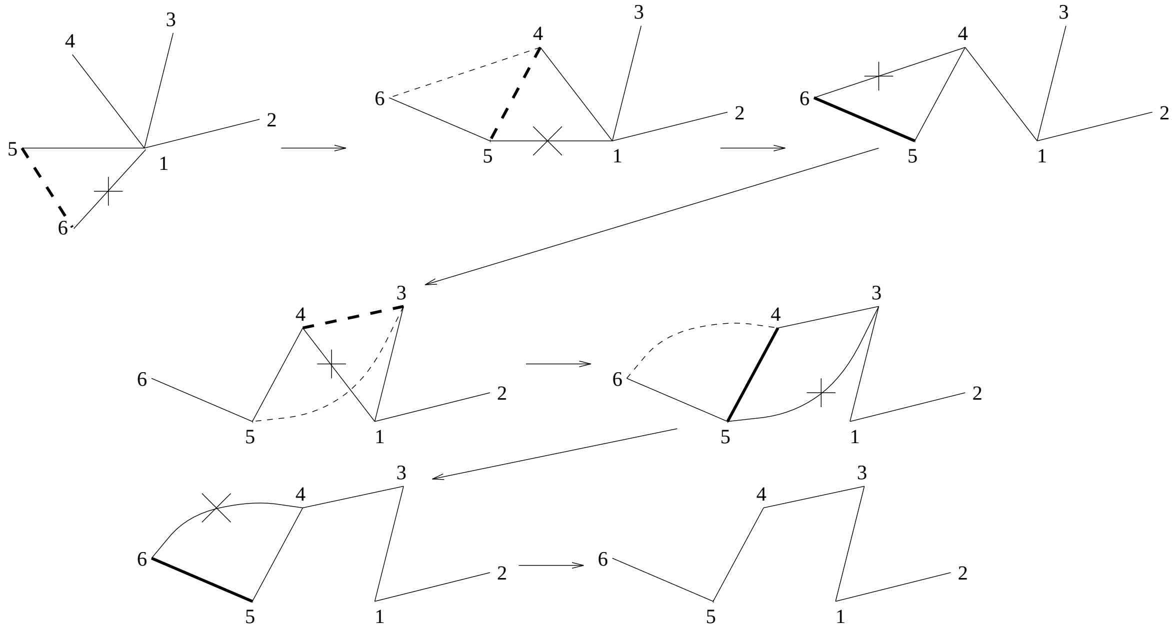

FIGURE 3. Graphical representation of Zha's algorithm for a 6×6 matrix.

for the new chasing step. The variables before and after a chasing step are illustrated in Figure 4.

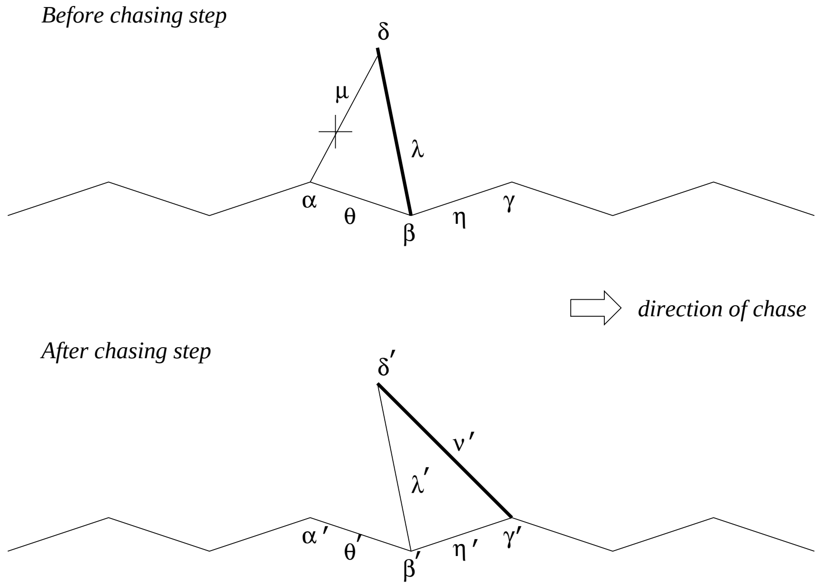

FIGURE 4. Variables before and after a chasing step.

The values c and s satisfy $c^2 + s^2 = 1$ as usual, and also

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} \theta \\ \mu \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}.$$

This operation requires 29 flops plus a square root. This number can be reduced by using fast Givens rotations, at the expense of maintaining diagonal scaling matrices.

While the standard and the new chasing step are similar in terms of operation count and stability properties, the graph representation of matrices clearly shows the basic difference between the two. They are not the same, even under arbitrary permutations. From Figure 1(b) we can see that, for the new chasing step the "bulge" always has (before and after chasing steps) a fixed node as one of its vertices, thus the new algorithm can be thought of as chasing a node along the path, while the standard chasing algorithm chases an edge. Also note that, the edge deleted in the old algorithm does not lie on the longest path, but does so in the new algorithm. Equivalently, there are more edges outside the main path in the new algorithm. This causes the density of bulges with an edge on the main path to be greater for the new algorithm, which is responsible for the better efficiency of our new algorithm for a pipelined implementation.

4. Arrowhead matrices to tridiagonal matrices

The new chasing algorithm can be used for tridiagonalizing arrowhead matrices in a manner which is similar to that used by H. Zha [9]. However, while Zha's algorithm starts with the last two entries and moves backwards through the matrix, this new algorithm starts with the 2nd and 3rd entries and moves forward through the matrix. The arrowhead matrix is assumed to have the point of the arrow at the $(1,1)$ entry. In Figure 5, the steps for tridiagonalizing a 6×6 arrowhead matrix are shown. In Figure 5, a "$*$" indicates a nonzero entry, "$+$" indicates a

$$
\begin{bmatrix} \ast & \ast & \ast & \ast & \ast & \ast \\ \ast & \ast & & & & \\ \ast & & \ast & & & \\ \ast & & & \ast & & \\ \ast & & & & \ast & \\ \ast & & & & & \ast \end{bmatrix}
\to
\begin{bmatrix} \ast & \ast & 0 & \ast & \ast & \ast \\ \ast & \ast & + & & & \\ 0 & + & \ast & & & \\ \ast & & & \ast & & \\ \ast & & & & \ast & \\ \ast & & & & & \ast \end{bmatrix}
\to
\begin{bmatrix} \ast & \ast & & 0 & \ast & \ast \\ \ast & \ast & \ast & + & & \\ & \ast & \ast & + & & \\ 0 & + & + & \ast & & \\ \ast & & & & \ast & \\ \ast & & & & & \ast \end{bmatrix}
\to
$$

$$
\begin{bmatrix} \ast & \ast & & & \ast & \ast \\ \ast & \ast & \ast & 0 & & \\ & \ast & \ast & \ast & & \\ & 0 & \ast & \ast & & \\ \ast & & & & \ast & \\ \ast & & & & & \ast \end{bmatrix}
\to
\begin{bmatrix} \ast & \ast & & & 0 & \ast \\ \ast & \ast & \ast & & + & \\ & \ast & \ast & \ast & + & \\ & & \ast & \ast & & \\ 0 & + & + & & \ast & \\ \ast & & & & & \ast \end{bmatrix}
\to
\begin{bmatrix} \ast & \ast & & & & \ast \\ \ast & \ast & \ast & & 0 & \\ & \ast & \ast & \ast & \ast & \\ & & \ast & \ast & + & \\ & 0 & \ast & + & \ast & \\ \ast & & & & & \ast \end{bmatrix}
\to
$$

$$
\begin{bmatrix} \ast & \ast & & & & \ast \\ \ast & \ast & \ast & & & \\ & \ast & \ast & \ast & 0 & \\ & & \ast & \ast & \ast & \\ & & 0 & \ast & \ast & \\ \ast & & & & & \ast \end{bmatrix}.
$$

FIGURE 5. Description of new chasing algorithm for a 6×6 matrix

newly created nonzero entry, and "0" indicates an entry that has just been made zero. The first step is to perform a Givens operation on rows/columns 2 and 3, zeroing the (1,3) (and thus, also the (3,1)) entry. This step also creates a new nonzero entry at (2,3) (and its symmetric pair). This new nonzero can be left as it is part of the tridiagonal matrix being created. Now apply a Givens operation to rows/columns 2 and 4 to zero the (1,4) entry, which creates the (2,4) and (3,4) entries (and their symmetric pairs). Apply a Givens operation to rows/columns 3 and 4 to zero the (2,4) entry and its symmetric pair. Apply a Givens operation to rows/columns 2 and 5 to zero the (1,5) entry, which also creates the (2,5) and (3,5) entries. Applying a Givens rotation to rows/columns 3 and 5 can zero the (2,5) entry, but creates another nonzero at the (4,5) entry. Then applying a Givens rotation to rows/columns 4 and 5 can zero (3,5) entry, leaving a tridiagonal matrix except for the (1,6) entry and its symmetric pair. The process can be continued to push this entry to the (5,6) position, or the matrix can be permuted using the cyclic permutation $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 1$. The corresponding graphical representation of this example is shown in Figure 6.

Furthermore, the 2-way modification of Zha [9] can also be used here to roughly halve the number of operations needed to convert an arrowhead matrix to tridiagonal form. This is done by creating two branches along which chasing is performed; before the chasing actually begins, the elements should be re-ordered to make the middle entry (entry $n/2$ or $(n \pm 1)/2$) the center of the *star*. This is illustrated in Figure 7. The effect of this permutation is a matrix in the form shown in Figure 8.

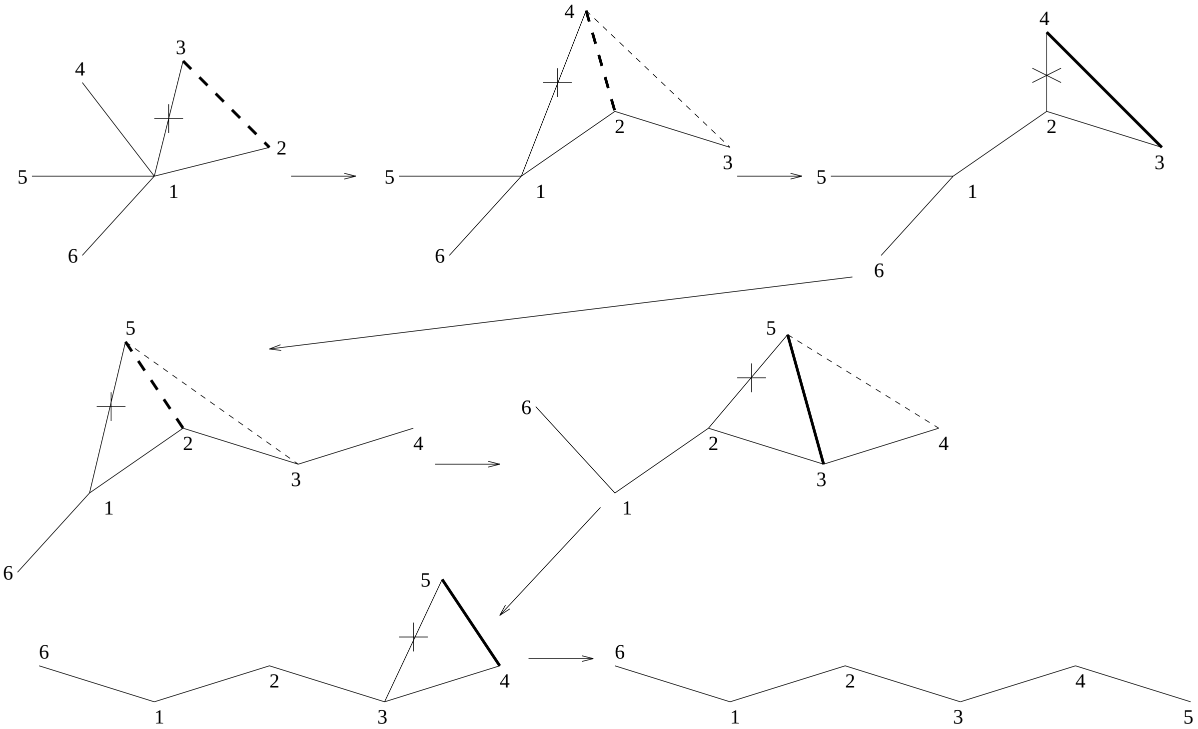

FIGURE 6. Graphical representation of the new chasing algorithm for a 6×6 matrix.

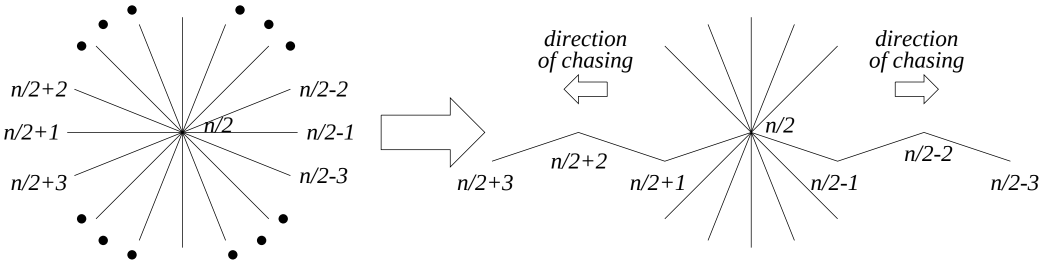

FIGURE 7. Two-way modification of chasing algorithm.

$$\begin{bmatrix} * & & & * & & & \\ & * & & * & & & \\ & & * & * & & & \\ * & * & * & * & * & * & * \\ & & & * & * & & \\ & & & * & & * & \\ & & & * & & & * \end{bmatrix}$$

FIGURE 8. Initial permutation for two-way chasing algorithm

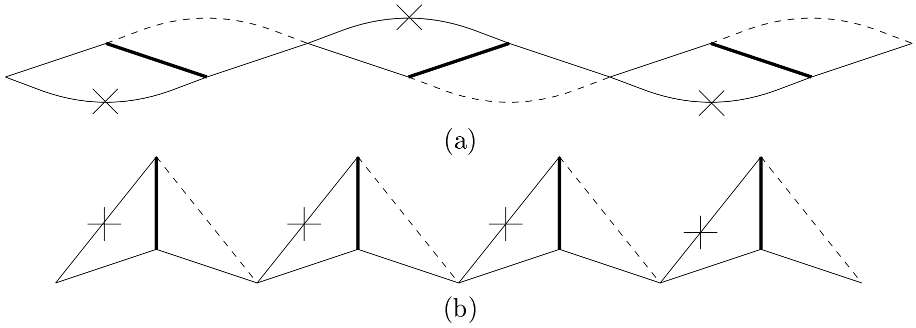

FIGURE 9. Simultaneous chasing: (a) standard algorithm, (b) new algorithm

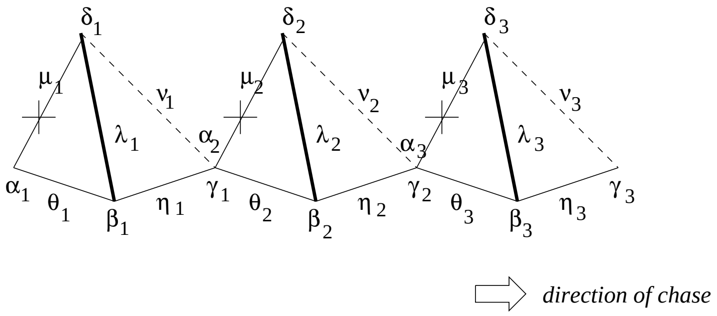

FIGURE 10. Simultaneous new chasing steps

5. PARALLEL ALGORITHM

Pipelined parallel algorithms are developed here for this new chasing algorithm on $O(n)$ processors with $O(n)$ time complexity for tridiagonalizing an $n \times n$ arrowhead matrix. The basis of the pipeline technique is the ability to chase multiple *bulges* along a chain simultaneously, as illustrated in Figure 9. Figure 9(a) shows simultaneous standard chasing, and Figure 9(b) shows the new chasing step performed simultaneously. For Figure 9(b), the only matrix entries that might be affected by more than one of the parallel chasing steps are those associated with the nodes in common with the old and new triangles. These diagonal entries are not changed by either of the adjacent chasing steps (note that $\alpha' = \alpha$ and $\gamma' = \gamma$ in (2)). Thus the chasing steps shown can be done independently. A similar argument can be applied to Figure 9(a) for the conventional chasing step. Note, however, that the density of bulges (triangles) is one and a half times greater for the new chasing step as for the conventional chasing step, but with the same number of floating point operations per chasing step per bulge. This gives a greater level of parallelism for the new chasing step. Figure 10 shows the variables for the new chasing step in parallel.

Practical parallel algorithms need care with problems of overhead in message passing. This is particularly true for pipeline algorithms where the natural message size is quite small, as is the case here. Thus the blocked version of the algorithm should be considered to make effective use of current message-passing technology. It should be noted that many *shared memory* machines are effectively distributed

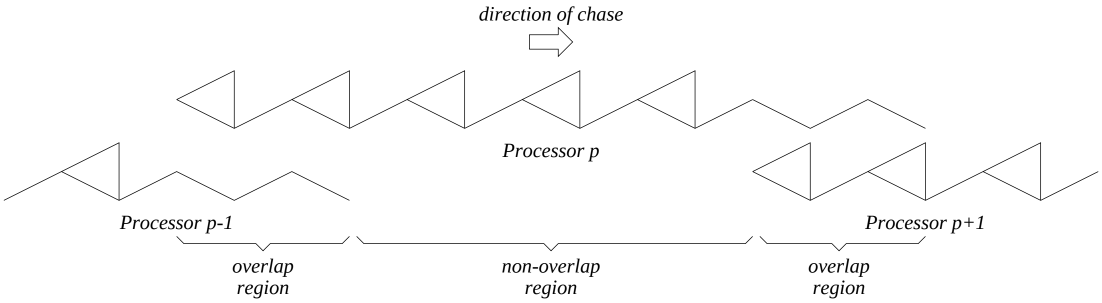

FIGURE 11. Partitioning of data between processors

memory machines as they commonly have a fast local *cache* and a considerably slower global memory which requires access through a large memory switch. To obtain high performance, access to global memory needs to be minimized and done in blocks. Another consideration is the load balance across the processors. Provided the block sizes are not too large, this can be done by using a *block wrap* mapping of the data.

In a blocked algorithm, there need to be buffer zones to allow for asynchronous computation in each processor, and for transmitting blocks of data between processors. In Figure 11 the buffer areas are represented as overlap regions between the processors. Let k be the number of edges that do not overlap, and l be the number of edges in each overlap region. Thus in Figure 11, $k = 7$ and $m = 3$. Note that in the new chasing step, the value η and γ are only used in computing η', ν', and γ'. Thus, the values α', β', δ', θ', λ', and $\mu'(= 0)$ are independent of η and γ. This means that entries in the matrix *downstream* do not affect the *upstream* entries. Consequently, there is no need for communication from processor $p + 1$ to processor p, and there is no need for synchronization until the end of the entire computation.

The sequence of operations on processor p is as follows:

1. Repeat items 2–5 until there are no more bulges to chase:
   2. Chase $\lceil l/2 \rceil$ bulges from the left-hand overlap region into the non-overlap region, and from the non-overlapping region into the right-hand overlap region, in processor p. This takes $\lceil (k+l)/2 \rceil$ simultaneous steps, repeated l times.
   3. Send $\lceil l/2 \rceil$ bulges from the right-hand overlap region in processor p to the left-hand overlap region in processor $p + 1$.
   4. Chase off $\lceil l/2 \rceil$ bulges from the right-hand overlap region passing the right-most edge of this region, treating the end of this overlap region as the end of the matrix. The nodes in the bulges can be ignored once they are chased past the end of the right-most overlap region, since downstream entries do not affect upstream entries.
   5. Receive $\lceil l/2 \rceil$ bulges from the right-hand overlap region in processor $p-1$ into the left-hand overlap region in processor p.

Note that using the 2-way version of the chasing algorithm, there will be two sets of data for each processor to process and communicate. They can be blocked together, which is advantageous for communication.

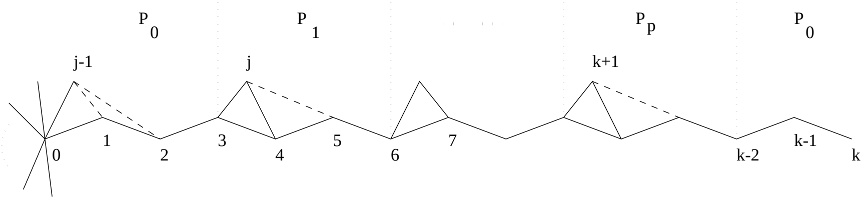

FIGURE 12. Parallel data distribution in graph notation

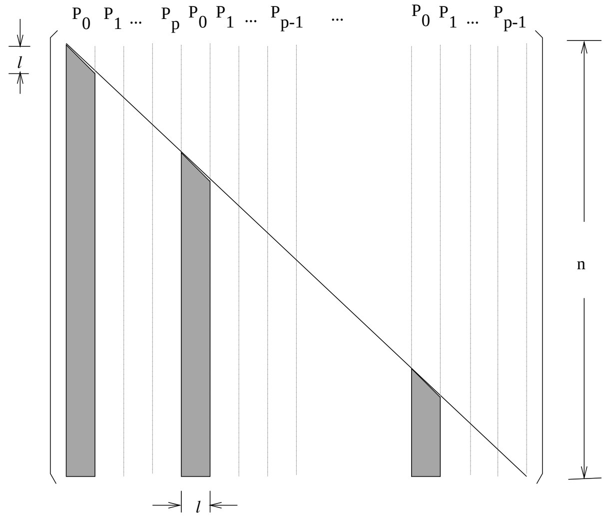

FIGURE 13. Parallel data distribution in matrix notation

6. IMPLEMENTATION

To describe the implementation issues we refer to the matrix model, even though the design of the new chasing algorithm was based on the graph of the matrix, as showed in this paper. Consider an $n \times n$ arrowhead matrix, three one-dimensional arrays are used to represent it: one for row one (or column one, since the matrix is symmetric) and two for the tridiagonal entries which will appear during the tridiagonalization process.

Processors are connected like an assembly line as shown in Figure 12. The processor will receive new bulges from the left, as it passes bulges to the next processor on the right. Each processor will start passing bulges to next processor when it starts processing the maximum number of bulges ($\lceil k/2 \rceil$) allowed per processor. On the other side in matrix notation, the processor load is indicated in Figure 13. We assume that columns of an $n \times n$ arrowhead matrix are divided into r blocks, which are assigned to p processors in a wrap-around fashion. Let l be the length of a block and m be the number of clocks assigned to each processor (wrap around number). We have $n = lr = lpm$.

The pipeline algorithm aims to maximize processor utilization and minimize processor communication. To achieve these goals we implemented three versions

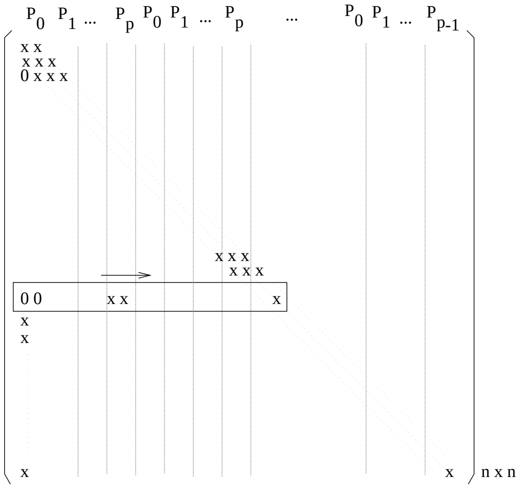

FIGURE 14. Illustration for one chasing cycle

of the algorithm employing techniques such as wrap around and message group passing:

Implementation 1: The matrix is evenly divided by the number of processors being used. Each processor is responsible for a particular portion of the matrix during the calculation.

Implementation 2: To maximize the processor utilization and to minimize processor idle time, a wrap around of paths (cyclic filling of the pipe) is used.

Implementation 3: To reduce communication time between processors, the information from several bulges is combined for passing them to other processors.

In matrix notation, it is easy to see that to reach its off-diagonal position, each element has to go through a **chasing cycle**. A chasing cycle is a collection of chasing steps, in which two entries will be chased off to the tridiagonal, one along the row and the other along the column. We need to perform $n-2$ chasing cycles for a matrix of size $n \times n$. Each chasing cycle with a different number of chasing steps. The whole process starts with one chasing step for the first chasing cycle and ends with $n-2$ chasing steps for the last one. A global view of the chasing cycle is shown in in Figure 14. Initially there is only one processor which is active since there is only one chasing step in the first chasing cycle. In the subsequent chasing cycles, the number of chasing steps for each new chasing cycle increases one by one. A processor will become active whenever the chasing cycle length is long enough to reach that processor.

7. Performance

7.1. **Analytical Model.** Because processor one is always being used (either initiating new chasing cycles or being re-used by the wrap-around storage. The total timing for the parallel algorithm is proportional to the time P_0 is active. This is represented by the shadowed areas in Figure 13. Let t_f be the time required for completing one chasing step (29 arithmetic flops), then the total computational time for P_0 is

$$\begin{aligned} T_{comp} &= t_f\{[n\ell - \frac{1}{2}\ell^2] + [(n-\ell p)\ell - \frac{1}{2}\ell^2] + \cdots + [(n-(m-1)\ell p)\ell - \frac{1}{2}\ell^2]\} \\ &= \frac{n}{2p}(n + \ell p - \ell)t_f. \end{aligned} \tag{3}$$

To estimate the communication time, we assume that passing a message with v floating-point words from one processor to its neighbor costs $t_c = t_s + v t_w$, where t_s is the startup time and t_w is the time per word transfer. One bulge data contain 6 words. Typically, $t_s \gg t_w \gg t_f$, and the ratio t_s/t_f for current parallel computers ranges from hundreds to tens of thousands. The timings to pass a message with data for g bulges is given by $t_g = t_s + 6gt_w$. Thus, we define $t_a = t_g/g$ to be the averaged message-passing time for one bulge. In Figure 13, the size of all messages sent by P_0 is represented by the right edges of the shadowed bars (that represents passing of data of chasing cycles to the next processor in the pipe). Here, we assume that each group messages with fixed length $6g$ for all processors. Thus, the communication time can be estimated by

$$\begin{aligned} T_{comm} &= \frac{t_g}{g}[(n-\ell) + (n-\ell-\ell p) + \cdots + (n-\ell-(m-1)\ell p) \\ &= \frac{1}{2\ell p}(n^2 + n\ell p - 2n\ell)t_a. \end{aligned} \tag{4}$$

Consequently, the parallel time is given by

$$\begin{aligned} T_p &= T_{comp} + T_{comm} \\ &= \frac{n}{2p}(n + \ell p - \ell)t_f + \frac{1}{2\ell p}(n^2 + n\ell p - 2n\ell)t_a. \end{aligned} \tag{5}$$

From this expression we can derive the optimal block length ℓ

$$\ell_{opt} = \sqrt{\frac{n\ t_a}{(p-1)\ t_f}},$$

which yields the optimal number of wrap-arounds

$$m_{opt} = \frac{1}{p}\sqrt{\frac{n(p-1)\ t_f}{t_a}}$$

The speed up of a parallel algorithm is equal to the ratio between the sequential time T_s and the parallel time T_p: $S = T_s/T_p$. Here T_s is equal to $(n^2/2)t_f$ thus the speed up for this algorithm is

$$S = (\frac{n^2}{2})t_f/[\frac{n}{2p}(n + \ell p - \ell)t_f + \frac{1}{2\ell p}(n^2 + n\ell p - 2n\ell)t_a]. \tag{6}$$

If $\ell = o(n)$, then $S \to p$ as $n \to \infty$.

7.2. **Numerical Results.** The algorithm was implemented on two machines located in Texas A&M University: an NCUBE with 64 nodes and a Paragon with 32 nodes. Both are an distributed memory MIMD parallel machines.

The matrix results shown here are for a 2048×2048 matrix, unless otherwise stated. Figure 15 shows both analytical and experimental parallel timings, for the first implementation, as size of the matrix varies, for different number of processors. The analytical time is predicted by (5), where the coefficients t_f and t_a were determined by numerical experiments ($t_f = 0.000854$ and $t_a = 0.007598$ seconds for the nCUBE and $t_f = 0.000014$ and $t_a = 0.000168$ seconds for the PARAGON). From this Figure we can see that the analytical and numerical results match well.

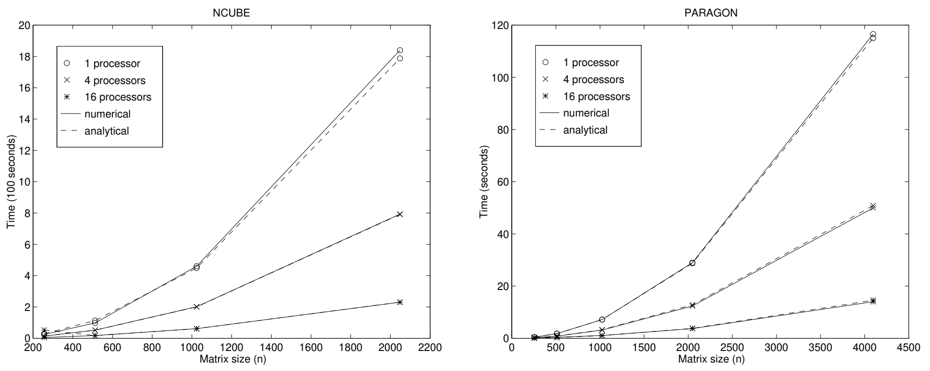

FIGURE 15. Case 1: no wrap-around and no group message passing. Parallel time with respect to different matrix size.

Figure 16 shows both analytical and experimental parallel timings for implementation 2 with various processor and wrap around numbers. Both analytical and numerical results indicate that the optimal wrap-around number is 8 for two processors, 7 for four processors, 5 for eight processors, and 4 for sixteen processors. Figure 17 shows both analytical and experimental parallel timings for implementa-

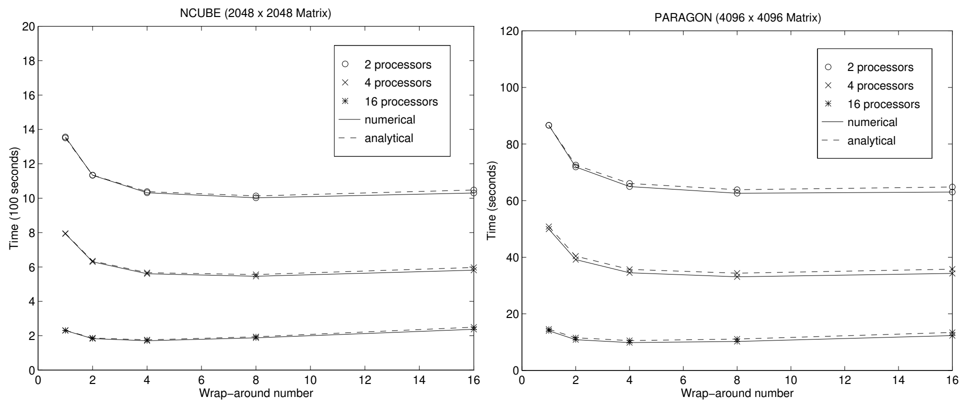

FIGURE 16. Parallel timings as a function of the number of wrap-arounds. No group messages considered (implementation 2)

tion 3 with different group message numbers. Here the wrap-around number is set to 8.

Efficiency results, corresponding to the three different ways of implementing the algorithm, are shown in Figure 18. The results are shown for four and sixteen processors. If we consider four processors, for example, we can make the following observations: For the first implementation we obtained efficiency around 50%. For the implementation with wrap around, the efficiency reaches 72%. For the third

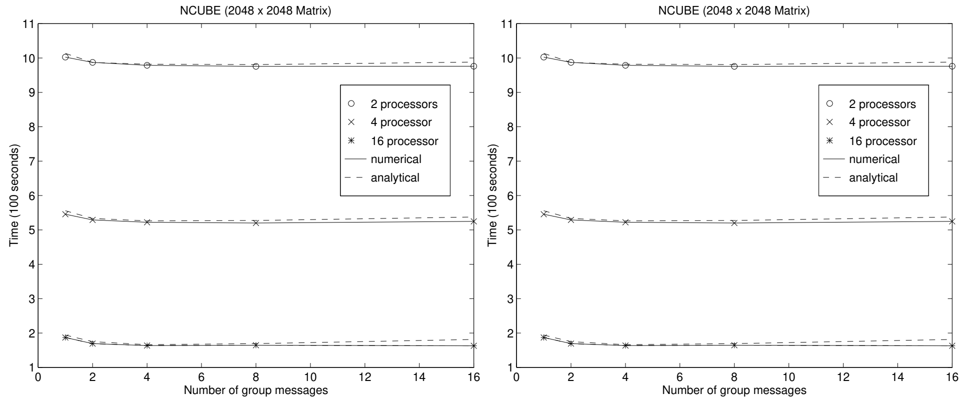

FIGURE 17. Case 3: Wrap around and group message-passing. Parallel time with respect to the number of group messages.

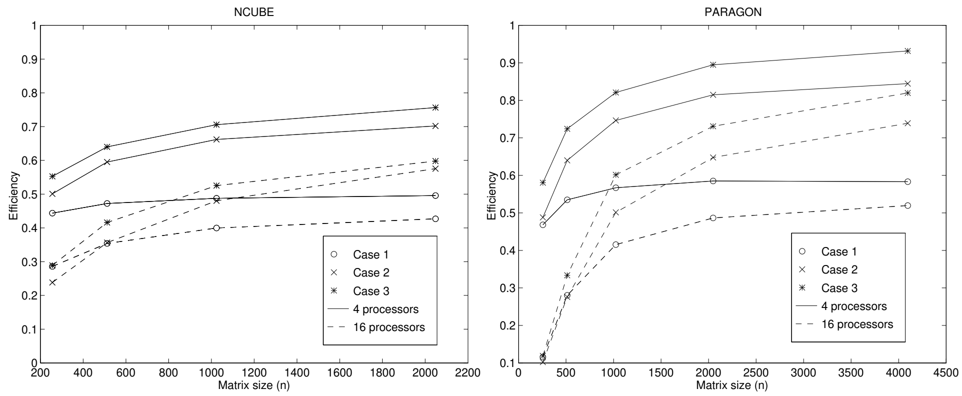

FIGURE 18. Efficiency with respect to matrix sizes

implementation (with wrap around and group message passing) we reached 76% efficiency). We can see from the results that the last implementation is the more efficient of the three considered, as one should expect.

Finally we want to show that, indeed our new algorithm performs better than the traditional approach due to the reasons stated in this paper. This is clearly confirmed by the timings in Figure 19. The timings shown there indicate a modest difference between the old and new methods. This difference increases at a constant ratio, as the size of the matrix grows.

Acknowledgments

I would like to thank Dr. D. Stewart who introduced me to the graph theory approach while I was visiting the Australian National University and also my students Z. Chen and Y. Deng who worked on the parallel implementations.

References

[1]

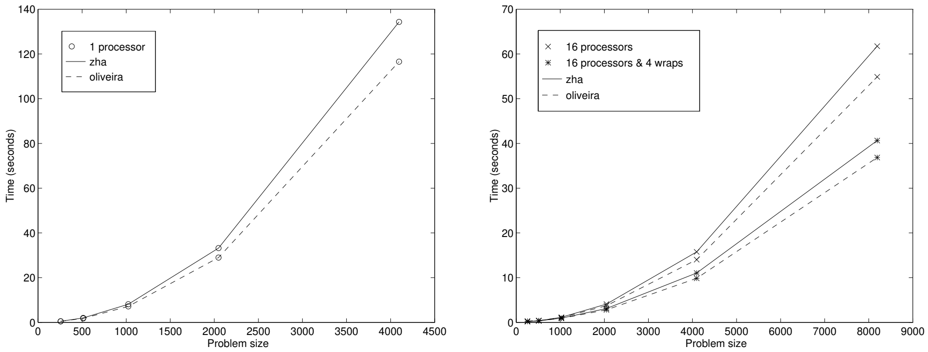

FIGURE 19. Comparison between standard and new chasing algorithm

[2] Z. CHEN, Y. DENG, AND S. OLIVEIRA, *An efficient parallel chasing algorithm*, in SIAM Conference on Parallel Processing for Scientific Computing Proceedings, March 1997. CD-ROM publication (6 pages).

[3] Y. DENG, *Some applications of pipelining techniques in parallel scientific computing*, Master's thesis, Texas A&M University, 1996.

[4] G. GOLUB AND C. VAN LOAN, *Matrix Computations*, Johns Hopkins Press, 2nd ed., 1989. 1st edition published 1983 by North Oxford Academic.

[5] W. B. GRAGG AND W. J. HARRROD, *The numerically stable reconstruction of jacobi matrices from spectral data*, Numer. Math., 44 (1984), pp. 317–335.

[6] H. RUTISHAUSER, *On Jacobi rotation patterns*, in Proceedings of Symposia in Applied Mathematics, vol. XV, Providence, RI, 1963, pp. 219–239.

[7] D. E. STEWART, *A graph theoretical model of Givens rotations and its implications*, Linear Alg. Appl., 257 (1997), pp. 311–320.

[8] S. VAN HUFFEL AND H. PARK, *Parallel tri- and bi-diagonalization of bordered bidiagonal matrices*, Parallel Computing, 20 (1994), pp. 1107–1128.

[9] H. Y. ZHA, *A two-way chasing scheme for reducing an arrowhead matrix to tridiagonal form*, J. Numer. Lin. Alg. Appl., 1 (1992), pp. 49–57.