Parallel Computing 18 (1992) 1301–1315
North-Holland

PARCO 727

Applying parallel computer systems to solve symmetric tridiagonal eigenvalue problems *

Mi Lu and Xiangzhen Qiao

Dept. of Electrical Engineering, Texas A&M University, College Station, TX 77843-3128, USA

Abstract

Lu, M. and X. Qiao, Applying parallel computer systems to solve symmetric tridiagonal eigenvalue problems, Parallel Computing 18 (1992) 1301–1315.

A block parallel partitioning method for computing the eigenvalues of symmetric tridiagonal matrix is presented. The algorithm is based on partitioning, in a way that ensures load balance during computation. This method is applicable to both shared memory- and distributed memory-MIMD systems. Compared with other parallel tridiagonal eigenvalue algorithms existing in the literature, the proposed algorithm achieves a higher speedup of $O(p)$ on a parallel computer with p-fold parallelism, which is linear, and the data communication between processors is less than that required for other methods. The results were tested and evaluated on an MIMD machine, and were within 62% to 98% of the predicted performance.

Keywords. General linear recurrence; parallel algorithm; parallel computer; parallel computer performance; symmetric tridiagonal eigenvalue.

1. Introduction

The numerical solution of large symmetric tridiagonal eigenvalue problem arises in many important scientific applications. Usually these problems are computing intensive and require high-speed computations. Advances in high-speed parallel computers have allowed the solution of very large scientific problems that were not possible a few years ago. This trend will continue as more sophisticated scientific models and more efficient numerical methods are developed.

Numerical algorithms can be generally classified in various ways. Since the emergence of modern parallel and vector computers, a new classification has become important: sequential and parallel. Unfortunately, algorithms are very architecture dependent with the current generation of parallel computers. A good algorithm for parallel and vector computers may be strikingly different from a good algorithm for sequential computers. Thus, it is of considerable importance to measure the effectiveness of a parallel algorithm. For a given problem with size n, let T_1 be the running time of the fastest sequential algorithm, and let T_p be the running time of a parallel algorithm using p processors. The speedup is defined as $S_p = T_1/T_p$. The

* This research was partially supported by the National Science Foundation under grant no. MIP8809328.
Correspondence to: Mi Lu, Dept. of Electrical Engineering, Texas A&M University, College Station, TX 77843-3128, USA.

efficiency is defined as $E_p = S_p/p$. It is obvious that $S_p \leq p$, and $E_p \leq 1$. The goal for the research of parallel algorithm is to construct an algorithm exhibiting linear (in p) speedup and hence utilizing the processors efficiently. For the sake of simplicity, we use the above definitions and assumptions of n, p, T_p, T_1, S_p and E_p throughout this paper. For problems of size n we want an asymptotic speedup of the form $S_p = cp - g(p, n)$, with $0 \leq c \leq 1$, $0 \leq g(p, n) = o(1)$, as $n \to \infty$; c should be independent of p. The function $g(p, n)$ can be think as a penalty for the use of parallelism on small problems. Any parallel algorithm which asymptotically attains linear speedup, is said to have the optimal speedup [12]. However, linear speedup is not always possible. There are certain computations for which the maximal speedup is $S_p \leq d$ for a constant d, which is independent of p, and such a computation clearly makes poor use of parallelism. For some problems, the maximal speedup is $S_p = cp/\log\ p - g(p, n)$, which is less than linear, though it is acceptable.

The parallel algorithms in eigenvalues computation were discussed by many researchers [4,5,8,11,13]. However, most of them were discussing the parallel algorithms on vector or shared memory computers. In fact, many of them were discussing dense symmetric eigenvalue problems. In this paper, we proposed a new parallel method for solving symmetric tridiagonal eigenvalue problems. The speedup is linear, and is greater than the existing methods.

Usually the bisection method based on the Sturm sequence [15] is used to solve the real symmetric tridiagonal eigenvalue problem. This method is inherently sequential, because of the Sturm sequence is calculated according to second order recurrence equations. There are some parallel methods existing in the literature for solving this kind of problems. Barlow and Evans [1] proposed a parallel bisection method, by using the principle on all the previously determined non-empty sub-intervals. However, the speedup is dependent upon the number of non-empty intervals available at each stage and bounded by the maximum number of eigenvalues. Another parallel method [2] is the multisection procedure, in which each sub-interval is divided into m sub-intervals. The Sturm sequence in each subinterval is determined in parallel on the available processors. Combining the advantages of the multisection method and parallel bisection method, they also developed a parallel multisection method [2]. If p processors and m domains are given, $l = p/m$ processors are located per domain. The speedup of his method lies between m and $m * \log_2(l + 1)$. In all the above methods, the parallelism is obtained through the simultaneous determination of several sequences for different sample points, but the recurrence equation is solved sequentially. Evans [6] has adopted the recursive doubling [10] method to the parallel computation of the Sturm sequence. The speedup is about $p/(4 * \log_2 n)$. The speedups of all the above methods are acceptable, but they are much less than linear. Motivated by the versatility and popularity of the divide-and-conquer technique, we developed a more powerful algorithm which combines the advantages of the segmentation and the cyclic reduction methods for solving general linear mth order recurrence equations and apply this method to solve the symmetric tridiagonal eigenvalue problem. The speedup of our algorithm is $0.4 * p$. It is linear, and the data communication between processors is less than that required for other methods. The principle exploited in our research was applied and tested on the Sequent Balance 8000 [14] parallel processing system. The test results are within 62% to 93% of the predicted performance.

The rest of this paper is organized as follows. The symmetric tridiagonal eigenvalue problem is described in Section 2. In Section 3 the formal development of the new parallel algorithm and its complexity analysis are given. Experimental results are presented in Section 4, and the paper concludes with Section 5.

2. The symmetric tridiagonal eigenvalue problem

Given a real symmetric tridiagonal matrix

$$A=\begin{bmatrix} c_1 & b_2 & & & & \\ b_2 & c_2 & b_3 & & & \\ & b_3 & c_3 & b_4 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & b_n \\ & & & & b_n & c_n \end{bmatrix}, \tag{1}$$

our purpose is to solve the eigenvalue problem

$$A * X=\lambda * X, \tag{2}$$

where λ is an eigenvalue of A, X is its corresponding eigenvector. When only a few eigenvalues and/or eigenvectors are desired, the bisection method based on Sturm sequence is appropriate.

Let A_r denote the leading r-by-r principal submatrix of A, and define the polynomials f_0, $f_1,\cdots,f_n$ by

$$f_0=1,$$
$$f_r=\det(A_r-\lambda I),$$
$$r=1,2,\cdots,n.$$

A simple determinantal expansion can be used to show that:

$$f_1(\lambda)=c_1-\lambda,$$
$$f_r(\lambda)=(c_r-\lambda) * f_{r-1}(\lambda)-b_r^2 * f_{r-2}(\lambda), \tag{3}$$
$$r=2,\cdots,n.$$

Because that $f_n(\lambda)$ can be evaluated in $O(n)$ flops, it is feasible to find its roots using the bisection technique. When the kth largest eigenvalue of A (k is an integer) is desired, the bisection idea and the Sturm sequence property can be adopted. It was later pointed out that if the tridiagonal matrix A in (1) is unreducible, then the eigenvalues of A_{r-1} strictly separate the eigenvalues of A_r:

$$\lambda_r(A_r)<\lambda_{r-1}(A_{r-1})<\lambda_{r-1}(A_r)<\cdots<\lambda_2(A_4)<\lambda_1(A_{r-1})<\lambda_1(A_r).$$

Moreover, if $q(\lambda)$ denotes the number of sign changes in the sequence:

$$\{f_0(\lambda), f_1(\lambda),\cdots,f_n(\lambda)\},$$

then $q(\lambda)$ equals the number of A's eigenvalues that are less than λ. (Convention: if $f_r(\lambda)=0$ then $f_r(\lambda)$ has the opposite sign from $f_{r-1}(\lambda)$.) Let

$$\begin{cases} g_0=1, \\ g_1=c_1-\lambda, & i=2,3,\cdots,n, \\ g_i=(c_i-\lambda) * g_{i-1}-b_i^2 * g_{i-2}, \end{cases} \tag{4}$$

where $g_i=f_i(\lambda)$, $i=0,1,2,\cdots,n$. By using the Sturm sequence method, the algorithmic solution involves the repeated computing of the above recurrence Equation (4). For different values of λ, the number of negative $g_i's$, $i=0,1,2,\cdots,n$, of the above sequences will indicate the number of eigenvalues below the sample point λ. Repeated application of this procedure will separate the eigenvalue spectrum into small sub-intervals of size ϵ (pre-defined) which

contain one or more eigenvalues λ. Let

$$\begin{bmatrix} g_1 \\ g_0 \end{bmatrix} = \begin{bmatrix} c_1 - \lambda \\ 1 \end{bmatrix} = \begin{bmatrix} e_0 \\ 1 \end{bmatrix},$$

$$\theta_{i-1} = \begin{bmatrix} c_i - \lambda & -b_i^2 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} e_{i-1} & d_{i-1} \\ 1 & 0 \end{bmatrix},$$

where

$$e_{i-1} = c_i - \lambda,$$

$$d_{i-1} = -b_i^2;$$

then (4) can be written as

$$\begin{bmatrix} g_i \\ g_{i-1} \end{bmatrix} = \theta_{i-1} * \begin{bmatrix} g_{i-1} \\ g_{i-2} \end{bmatrix},$$

$$i = 2, \cdots, n.$$

Let

$$X_i = \begin{bmatrix} g_i \\ g_{i-1} \end{bmatrix},$$

$$X_1 = \begin{bmatrix} e_0 \\ 1 \end{bmatrix},$$

then

$$X_i = \theta_{i-1} * X_{i-1}, \tag{5}$$

$$i = 2, \cdots, n.$$

This is a general linear recurrence problem.

Given a general linear mth order recurrence equation

$$\begin{cases} x_1 = b_1, \quad x_0 = b_0, \quad \cdots, \quad x_{2-m} = b_{2-m}, \\ x_i = a_{i-1,1} * x_{i-1} + a_{i-1,2} * x_{i-2} + \cdots + a_{i-1,m} * x_{i-m} + b_i; \\ i = 2, 3, \cdots, n. \end{cases} \tag{6}$$

Let

$$X_i = \begin{bmatrix} x_i \\ x_{i-1} \\ \vdots \\ x_{i-m+1} \end{bmatrix},$$

$$\theta_{i-1} = \begin{bmatrix} a_{i-1,1} & a_{i-1,2} & \cdots & a_{i-1,m-1} & a_{i-1,m} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & \cdots & 0 \\ & & \cdots & & \\ & & & 1 & 0 \end{bmatrix},$$

$$B_i = \begin{bmatrix} b_i \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad i = 2, \cdots, n,$$

$$B_1 = \begin{bmatrix} b_1 \\ b_0 \\ \vdots \\ b_{2-m} \end{bmatrix}.$$

By using the vector-matrix form, we have

$$\begin{cases} X_1 = B_1, \\ X_i = \theta_{i-1} * X_{i-1} + B_i, \quad i = 2, \cdots, n. \end{cases} \tag{7}$$

So (5) is a special case of (7), where $B_i = 0$ $(i \geq 2)$, and $m = 2$. In the next section, we describe a new parallel method for solving (7) and use it to solve the symmetric tridiagonal eigenvalue problem.

3. The new parallel algorithm

The general linear first order recurrence can be expressed as the determination of the sequence of x_i:

$$\begin{cases} x_1 = b_1, \\ x_i = a_{i-1} * x_{i-1} + b_i, \quad i = 2, 3, \cdots, n, \end{cases} \tag{8}$$

where a_i and b_i $(i = 1, 2, \cdots, n)$ are known. The parallel algorithm for solving (8) is well investigated by many researchers [9,10]. The new method in this paper aims to the combination of the advantages of the segmentation [7] and the cyclic reduction method [3]. The basic idea of the cyclic reduction method for solving general linear first order recurrence problems is to combine adjacent terms of the recurrence together in such a way as to obtain a relation between every other term in the sequence, that is to relate x_j, say, to x_{j-2}. This relation is also a linear first-order recurrence, although the coefficients are different and the relation is between alternate terms. Consequently, the process can be repeated to obtain recurrence relating every fourth term, every eighth term, and so on. When the recurrence relating every nth term (i.e. after $\log_2 n$ levels of reduction), the value at each point in the sequence is related only to values outside the range which are known or equal to zero, hence the solution has been found. This method is efficient when it is used on vector computers. However, it has some disadvantages when it is used on an MIMD machine. The parallelism is changeable during the computation. The overhead is due to the heavy data transformation. The meaning of the segmentation is obvious. In fact, it is a divide-and-conquer method. The high parallelism can be achieved by this method, but it also has the disadvantage of the high overhead of data transformation. Motivated by the versatility and popularity of the divide-and-conquer technique, we attempt to develop a more powerful algorithm combining the advantages of the above two methods.

A general linear mth order recurrence problem can be described as:

$$\begin{cases} X_1 = B_1, \\ X_i = \theta_{i-1} * X_{i-1} + B_i, \quad i = 2, \cdots, n, \end{cases} \tag{9}$$

where X_i, B_i and θ_i are defined as in (7). For solving (9) on an MIMD machine with parallelism p, we divide $\{X_i\}$, $i = 1, 2, \cdots, n$, into p groups each with $k = n/p$ vectors. For the sake of simplicity and without loss of generality, we assume that k is an integer. This is the segmentation approach. Now Equation (9) is divided into p groups each with k equations. The lth group consists of the following equations.

$$X_{l*k+j+1} = \theta_{l*k+j} * X_{l*k+j} + B_{l*k+j+1},$$

where $0 \le l \le p-1$ and $1 \le j \le k$.

Instead of combining adjacent terms, here we try to relate every term with the first one in the same group.

$$X_{l*k+2} = \theta_{l*k+1} * X_{l*k+1} + B_{l*k+2},$$

and

$$\begin{aligned} X_{l*k+3} &= \theta_{l*k+2} * X_{l*k+2} + B_{l*k+3} \\ &= \theta_{l*k+2} * (\theta_{l*k+1} * X_{l*k+1} + B_{l*k+2}) + B_{l*k+3} \\ &= \theta_{l*k+2} * \theta_{l*k+1} * X_{l*k+1} + (\theta_{l*k+2} * B_{l*k+2} + B_{l*k+3}) \\ &= \theta^{(1)}_{l*k+1} * X_{l*k+1} + B^{(1)}_{l*k+3}, \end{aligned}$$

where

$$\theta^{(1)}_{l*k+1} = \theta_{l*k+2} * \theta^{(0)}_{l*k+1},$$

$$B^{(1)}_{l*k+3} = \theta_{l*k+2} * B_{l*k+2} + B_{l*k+3}.$$

Hence,

$$\begin{aligned} X_{l*k+4} &= \theta_{l*k+3} * X_{l*k+3} + B_{l*k+4} \\ &= \theta_{l*k+3} * (\theta^{(1)}_{l*k+1} * X_{l*k+1} + B^{(1)}_{l*k+3}) + B_{l*k+4} \\ &= \theta_{l*k+3} * \theta^{(1)}_{l*k+1} * X_{l*k+1} + (\theta_{l*k+3} * B^{(1)}_{l*k+3} + B_{l*k+4}) \\ &= \theta^{(2)}_{l*k+1} * X_{l*k+1} + B^{(2)}_{l*k+4}, \end{aligned}$$

where

$$\theta^{(2)}_{l*k+1} = \theta_{l*k+3} * \theta^{(1)}_{l*k+1},$$

$$B^{(2)}_{l*k+4} = \theta_{l*k+3} * B^{(1)}_{l*k+3} + B_{l*k+4}.$$

Using the method of induction, we have

$$\begin{aligned} X_{l*k+j} &= \theta_{l*k+j-1} * X_{l*k+j-1} + B_{l*k+j} \\ &= \theta_{l*k+j-1} * (\theta^{(j-3)}_{l*k+1}) * X_{l*k+1} + B^{(j-3)}_{l*k+j-1} + B_{l*k+j} \\ &= \theta_{l*k+j-1} * \theta^{(j-3)}_{l*k+1} * X_{l*k+1} + (\theta_{l*k+j-1} * B^{(j-3)}_{l*k+j-1} + B_{l*k+j}) \\ &= \theta^{(j-2)}_{l*k+1} * X_{l*k+1} + B^{(j-2)}_{l*k+j}, \\ & j = 1, 2, 3, \cdots, k. \end{aligned}$$

Assume

$$\theta^{(0)}_{l*k+1} = \theta_{l*k+1},$$

$$B^{(0)}_{l*k+j} = B_{l*k+j},$$

$$\theta^{(-1)}_{l*k+1} = I,$$

$$B^{(-1)}_{l*k+j} = 0.$$

We can obtain the following relations:

$$X_{l*k+j} = \theta_{l*k+1}^{(j-2)} * X_{l*k+1} + B_{l*k+j}^{(j-2)}, \tag{10}$$

where

$$\begin{cases} \theta_{l*k+1}^{(j-2)} = \theta_{l*k+j-1} * \theta^{(j-3)} \\ B_{l*k+j}^{(j-2)} = \theta_{l*k+j-1} * B_{l*k+j-1}^{(j-3)} + B_{l*k+j}, \end{cases} \tag{11}$$

$$0 \le l \le p-1, \; 2 \le j \le k,$$

and (9) becomes a smaller problem which needs fewer recurrent computations. Let $j = k+1$, from (10) we can get the following equation:

$$X_{(l+1)*k+1} = \theta_{l*k+1}^{(k-1)} * X_{l*k+1} + B_{(l+1)*k+1}^{(k-1)}, \tag{12}$$

$$0 \le l \le p-2,$$

where the coefficients are obtained from (11). When the first vector of every group (X_{l*k+1}'s) are calculated, the remaining $(n-p)$ X's can be obtained from (10) (or from (9)).

Thus, the all computational process consists of 3 phases:

(i) Calculate $\theta_{l*k+1}^{(j-1)}$ and $B_{l*k+j}^{(j-1)}$ from (11). This computation is parallel for l, $(0 \le l \le p-1)$, but sequential for j $(2 \le j \le k)$.

(ii) Calculate $X_{(l+1)*k+1} (0 \le l < p-1)$ sequentially,

$$X_{(l+1)*k+1} = \theta_{l*k+1}^{(k-1)} * X_{l*k+1} + B_{(l+1)*k+1}^{(k-1)} (X_1 \text{ is known}).$$

When these vectors are calculated, the results should be transferred between processors in a distributed memory MIMD system. The data communication needed is only a vector, i.e. $m-1$ data for an mth order linear recurrence problem.

(iii) Calculate X_{l*k+j},

$$0 \le l \le p-1, \quad 2 \le j \le k$$

using (10). This can be done in parallel for l on p processors, but sequential for j.

This method can be easily used to solve symmetric tridiagonal eigenvalue problems. From Equations (10)–(12) and (5), if we divide vectors $X_i = \begin{bmatrix} g_i \\ g_{i-1} \end{bmatrix}$ into p groups each with k vectors, then we have

$$\begin{aligned} \theta_{l*k+1}^{(1)} &= \theta_{l*k+2} * \theta_{l*k+1}^{(0)} \\ &= \begin{bmatrix} e_{l*k+2} & d_{l*k+2} \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} e_{l*k+1} & d_{l*k+1} \\ 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} e_{l*k+2} * e_{l*k+1} + d_{l*k+2} & e_{l*k+2} * d_{l*k+1} \\ e_{l*k+1} & d_{l*k+1} \end{bmatrix} \\ &= \begin{bmatrix} e_{l*k+1}^{(1)} & d_{l*k+1}^{(1)} \\ e_{l*k+1}^{(0)} & d_{l*k+1}^{(0)} \end{bmatrix}, \end{aligned}$$

where

$$\begin{cases} e_{l*k+1}^{(1)} = e_{l*k+2} * e_{l*k+1} + d_{l*k+2}, \\ d_{l*k+1}^{(1)} = e_{l*k+2} * d_{l*k+1}, \end{cases}$$

$$\theta^{(2)}_{l*k+1} = \theta_{l*k+3} * \theta^{(1)}_{l*k+1}$$
$$= \begin{bmatrix} e_{l*k+3} & d_{l*k+3} \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} e^{(1)}_{l*k+1} & d^{(1)}_{l*k+1} \\ e^{(0)}_{l*k+1} & d^{(0)}_{l*k+1} \end{bmatrix}$$
$$= \begin{bmatrix} e_{l*k+3} * e^{(1)}_{l*k+1} + d_{l*k+3} * e^{(0)}_{l*k+1} & e_{l*k+3} * d^{(1)}_{l*k+1} + d_{l*k+3} * d^{(0)}_{l*k+1} \\ e^{(1)}_{l*k+1} & d^{(1)}_{l*k+1} \end{bmatrix}$$
$$= \begin{bmatrix} e^{(2)}_{l*k+1} & d^{(2)}_{l*k+1} \\ e^{(1)}_{l*k+1} & d^{(1)}_{l*k+1} \end{bmatrix}$$

and

$$e^{(2)}_{l*k+1} = e_{l*k+3} * e^{(1)}_{l*k+1} + d_{l*k+3} * e^{(0)}_{l*k+1},$$
$$d^{(2)}_{l*k+1} = e_{l*k+3} * d^{(1)}_{l*k+1} + d_{l*k+3} + d^{(0)}_{l*k+1}.$$

In general (from the method of induction),

$$\theta^{(j-2)}_{l*k+1}$$
$$= \theta_{l*k+j-1} * \theta^{(j-3)}_{l*k+1}$$
$$= \begin{bmatrix} e_{l*k+j-1} & d_{l*k+j-1} \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} e^{(j-3)}_{l*k+1} & d^{(j-3)}_{l*k+1} \\ e^{(j-4)}_{l*k+1} & d^{(j-4)}_{l*k+1} \end{bmatrix}$$
$$= \begin{bmatrix} e_{l*k+j-1} * e^{(j-3)}_{l*k+1} + d_{l*k+j-1} * e^{(j-4)}_{l*k+1} & e_{l*k+j-1} * d^{(j-3)}_{l*k+1} + d_{l*k+j-1} * d^{(j-4)}_{l*k+1} \\ e^{(j-3)}_{l*k+1} & d^{(j-3)}_{l*k+1} \end{bmatrix}$$
$$= \begin{bmatrix} e^{(j-2)}_{l*k+1} & d^{(j-2)}_{l*k+1} \\ e^{(j-3)}_{l*k+1} & d^{(j-3)}_{l*k+1} \end{bmatrix}.$$

Defining

$$e^{(-1)}_{l*k+1} = 1,$$
$$d^{(-1)}_{l*k+1} = 0,$$
$$e^{(0)}_{l*k+j} = e_{l*k+j},$$
$$d^{(0)}_{l*k+j} = d_{l*k+j},$$

we have

$$\begin{cases} e^{(j-2)}_{l*k+1} = e_{l*k+j-1} * e^{(j-3)}_{l*k+1} + d_{l*k+j-1} * e^{(j-4)}_{l*k+1}, \\ e^{(j-2)}_{l*k+1} = e_{l*k+j-1} * d^{(j-3)}_{l*k+1} + d_{l*k+j-1} * d^{(j-4)}_{l*k+1}, \end{cases}$$
$$l = 0, 1, \cdots, p-1, \quad \text{and } j = 2, 3, \cdots, k+1.$$

Obviously,

$$B_{l*k+j} = 0.$$

Substitute $\theta^{(k-1)}_{l*k+1}$ into (12), we have,

$$X_{(l+1)*k+1} = \begin{bmatrix} g_{(l+1)*k+1} \\ g_{(l+1)*k} \end{bmatrix}$$
$$= \theta^{(k-1)}_{l*k+1} * \begin{bmatrix} g_{l*k+1} \\ g_{l*k} \end{bmatrix}, \tag{13}$$
$$i = 0, 1, \cdots, p-1.$$

As g_1 and g_0 are known, we can calculate all the $g_{(l+1)*k+1}$'s, and $g_{(l+1)*k}$'s from (13). Subsequently, all the other g_i's can be calculated using the following equations

$$X_{l*k+j} = \theta_{l*k+j-1} * X_{l*k+j-1} \tag{14}$$

or

$$X_{l*k+j} = \theta_{l*k+1}^{(j-2)} * X_{l*k+1}, \quad j = 2, 3, \cdots, k. \tag{15}$$

In fact this can be simply done by using the following formulas

$$g_{l*k+j} = e_{l*k+j-1} * g_{l*k+j-1} + d_{l*k+j-1} * g_{l*k+j-2} \tag{16}$$

or

$$g_{l*k+j} = e_{l*k+1}^{(j-2)} * g_{l*k+1} + d_{l*k+1}^{(j-2)} * g_{l*k}. \tag{17}$$

From the above analysis, the parallel procedure for determining the Sturm sequence at point λ can be given in the following algorithms.

Algorithm I

Input (i) sample point λ

(ii) a symmetric tridiagonal matrix

$$A = \begin{bmatrix} c_1 & b_2 & & & \\ b_2 & c_2 & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & b_n \\ & & & b_n & c_n \end{bmatrix}$$

and its order n. Suppose $n = p * k$.

Output

sequence $\{f_0(\lambda), f_1(\lambda), \cdots, f_n(\lambda)\}$,

where $f_i(\lambda)$'s, $(0 \le i \le n)$, are defined as in Section 2.

Algorithm

(i) divide $\{c_1, c_2, c_3, \cdots, c_n\}$ and $\{1, d_2, d_3, \cdots, d_n\}$ into p parts and assign the lth part on processor l, where $d_i = -b_i^2$ $(2 \le i \le n)$.

(ii) **for** $l = 0$ **to** $p - 1$ **parallel do**

$$e_{l*k+j} = c_{l*k+j+1} - \lambda, \quad 0 \le j \le k-1.$$

(iii) **for** $l = 0$ **to** $p - 1$ **parallel do**

$$e_{l*k+1}^{(j-1)} = e_{l*k+j} * e_{l*k+1}^{(j-2)} + d_{l*k+j} * e_{l*k+1}^{(j-3)},$$

$$d_{l*k+j}^{(j-1)} = e_{l*k+j} * d_{l*k+1}^{(j-2)} + d_{l*k+j} * d_{l*k+1}^{(j-3)},$$

$$j = 2, 3, \cdots, k,$$

where

$$d_{l*k+1}^{(-1)} = 0, \quad \text{and} \quad e_{l*k+1}^{(-1)} = 1.$$

(iv) **for** $l = 0$ **to** $p - 1$ **sequentially do**

$$g_{(l+1)*k+1} = e_{l*k+1}^{(k-1)} * g_{l*k+1} + d_{l*k+1}^{(k-1)} * g_{l*k},$$

$$g_{(l+1)*k} = e_{l*k+1}^{(k-2)} * g_{l*k+1} + d_{l*k+1}^{(k-2)} * g_{l*k}.$$

(v) **for** $l = 0$ **to** $p - 1$ **parallel do**

$$g_{l*k+j} = e_{l*k+j-1} * g_{l*k+j-1} + d_{l*k+j-1} * g_{l*k+j-2}$$

or

$$g_{l*k+j} = e_{l*k+1}^{(j-2)} * g_{l*k+1} + d_{l*k+1}^{(j-2)} * g_{l*k},$$
$$j = 2, 3, \cdots, k-1.$$

(vi) $f_{l*k+j}(\lambda) = g_{l*k+j}$.

Using *Algorithm I*, we can easily obtain the parallel algorithm for determining an eigenvalue of a symmetric tridiagonal matrix. For the sake of simplicity and without loss of generality, we only give the algorithm for calculating the largest eigenvalue of a symmetric tridiagonal matrix.

Algorithm II

Input (i) A symmetric tridiagonal matrix A, as in *Algorithm I*, and its order n.

(ii) An interval $[\lambda_1, \lambda_2]$ containing the largest eigenvalue λ and a small positive number ϵ.

Output The largest eigenvalue λ satisfying the accuracy ϵ.

Algorithm

(i) Calculate sequence $\{f_0(\lambda), f_1(\lambda), \cdots, f_n(\lambda)\}$ $(\lambda = (\lambda_1 + \lambda_2)/2)$, using *Algorithm I*.

(ii) Calculate the number of sign changes in the above sequence, denoted by $q(\lambda)$. If $q(\lambda) = n$, let $\lambda_2 = \lambda$; if $q(\lambda) = n - 1$, let $\lambda_1 = \lambda$.

(iii) if $|\lambda_1 - \lambda_2| \geq \epsilon$ go to (i).

(iv) $\lambda = (\lambda_1 + \lambda_2)/2$ is the largest eigenvalue.

Next we analyze the time complexity of *Algorithm I* and its efficiency. The time needed for evaluating a Sturm sequence on one processor is

$$T_1 = (2T_* + 2T_+) * n,$$

where T_+ and T_* are the time needed for one operation of addition and multiplication respectively. The time needed for *Algorithm I* on a parallel machine with parallelism p should be:

$$T_p = k * T_+ + (k-1) * (4 * T_* + 2 * T_+) + p * (2 * T_* + T_+) + (k-2) * (2T_* + T_+).$$

Assume $T_* = T_+$, and notice that usually $k \gg 2$, we have

$$T_p \approx k * (6 * T_* + 4 * T_+) + p(2 * T_* + T_+)$$
$$\approx 10 * k * T_+ + 3 * p * T_+,$$

and

$$S_p = \frac{T_1}{T_p}$$
$$\approx \frac{4 * n * T_+}{10k * T_+ + 3 * p * T_+}$$
$$\approx 0.4 * p/(1 + 0.3 * p * p/n)$$
$$\approx 0.4 * p - 0.12 * p * p * p/n.$$

Here the term $0.3 * p * p/n$ represents the sequential fraction – non-parallel fraction in this computation. According to the criterion proposed in [8], the speedup is optimal, and it is greater than the previous results (see the results in [4]). When $n \gg p$, $S_p \approx 0.4 * p$, $E_p \approx 40\%$. In this case, this fraction can be ignored, and the speedup is a linear function of the number of the processors.

For a fixed n, when p increases, the S_p increases too. Because of the influence of the fraction of sequential computation, there should be a critical point, forbidding the increase of S_p. It can be observed that $P_c = \sqrt{\left(\frac{5}{3} * n\right)}$. This provides an upper bound on the number of processors, beyond which no reduction in computation time can be achieved. This bound determines that, for example, when $n = 1024$, one should not use more than 25 processors even though they are available. This knowledge can be very useful in the efficient use of processors in a multiprogramming environment. For a fixed p, when n is too small compared with p, perhaps another algorithm, even sequential algorithm, is better than ours. This is because of the fact that the gain in speed achieved from increasing the amount of parallel execution may not outweigh the cost of introducing extra arithmetic operations. It can be seen that the critical point for the problem size, n, should be $n_c = 0.3 * p * p/(p-1)$. Usually in scientific computation, n is much greater than n_c. Thus, the proposed method is practical and acceptable.

4. Experimental results

We have run a test program on the Sequent Balance 8000 parallel processing system. This is a shared memory MIMD machine [14]. There are 10 processors in this system and one is dedicated to the operating system. This is a practical multiprocessor system. It incorporates multiple identical processors (CPUs) and a single common memory. The Sequent CPUs are general-purpose, 32-bit microprocessors. They are tightly coupled. All processors share a single pool of memory, to enhance resource sharing and communication among different processors.

Sequent systems support the two basic kinds of parallel programming: multiprogramming and multitasking. Multiprogramming is an operating system feature that allows a computer to execute multiple unrelated programs concurrently. Multitasking is a programming technique that allows a single application to consist of multiple processes executing concurrently. The multitasking yields an improved execution speed for individual programs. The Sequent system supports multitasking by allowing a single application to consist of multiple, closely cooperating processes. The Sequent language software includes multitasking extension to C, Pascal, and FORTRAN. The Parallel Programming Library is to support the multitasking. Sequent FORTRAN includes a set of special directives for parallel loops. With these directives, we mark the loops to be executed in parallel and classify loop variables so that data is passed correctly between loop iterations. The FORTRAN compiler interprets the directives and restructures the source code for data partitioning. In addition to these directives and Parallel Programming Library, the difference between a sequential and parallel program is distinguished by the compiler option-*mp*.

In adopting an application for multitasking, we often have the following goals: run as much of the program in parallel as possible and balance the computational load as evenly as possible among parallel processes. For efficient multitasking we have to choose the right programming method. Most applications naturally lend themselves to one of two multiprogramming methods: data partitioning and function partitioning. Data partitioning involves creating multiple, identical processes and assigning a portion of the data to each process.

Table 1(a)
Execution times (in seconds)

n / p	480	960	1920	3840	7680	11520	15360
1	7.05	14.11	28.22	56.42	112.88	169.88	225.94
4	4.79	9.28	18.30	36.29	72.33	108.38	143.30
5	4.24	8.20	16.13	32.00	63.64	95.34	126.69
6	3.91	7.5	14.75	29.14	58.00	86.90	115.50
8	3.51	7.00	14.00	27.53	53.47	78.54	104.09

n: size of the matrix
p: number of processors

Table 1(b)
Speedups

n / p	480	960	1920	3840	7680	11520	15360
4	1.47	1.52	1.54	1.55	1.56	1.57	1.58
5	1.66	1.72	1.75	1.76	1.77	1.78	1.78
6	1.80	1.88	1.91	1.94	1.95	1.95	1.96
8	2.01	2.02	2.02	2.11	2.05	2.16	2.17

n: size of the matrix
p: number of processors

Table 1(c)
Efficiency (%)

n / p	480	960	1920	3840	7680	11520	15360
4	36.80	38.01	38.55	38.87	39.02	39.19	39.41
5	33.25	34.41	34.99	35.26	35.47	35.64	35.67
6	30.05	31.35	31.89	32.27	32.44	32.58	32.60
8	25.14	25.20	25.20	25.62	26.39	27.04	27.13

n: size of the matrix
p: number of processors

Function partitioning, on the other hand, involves creating multiple unique processes and having them simultaneously perform different operations on a shared data set. It can be seen that the data partitioning is appropriate for our new parallel algorithm. It is done by executing the outer loop iterations in parallel in step (ii), (iii) and (v) of *Algorithm I*.

For a given symmetric tridiagonal matrix and different size n, we evaluate its largest eigenvalue by bisection method using *Algorithm I* and *II*. There are two programs in our test. The first one named TRID uses the general bisection method, in which the Sturm sequence is calculated sequentially. The second one named PTRD uses the bisection method, with parallel Sturm sequence calculations. The parallelism is achieved by using the Parallel Programming Library function *m_fork* on the Sequent B8000. The program PTRD was compiled with the option-*mp*, and their CPU times are recorded. The measured time, the speedup and efficiency for different n (matrix size) and p (number of processors) are given in *Table 1*. In each case, 5 runs were made, and the measured time represents the average of 3 experiments obtained by discarding the largest and the smallest figures. It can be seen that the achieved results for the speedup and efficiency are within 62% and 98% of the predicted

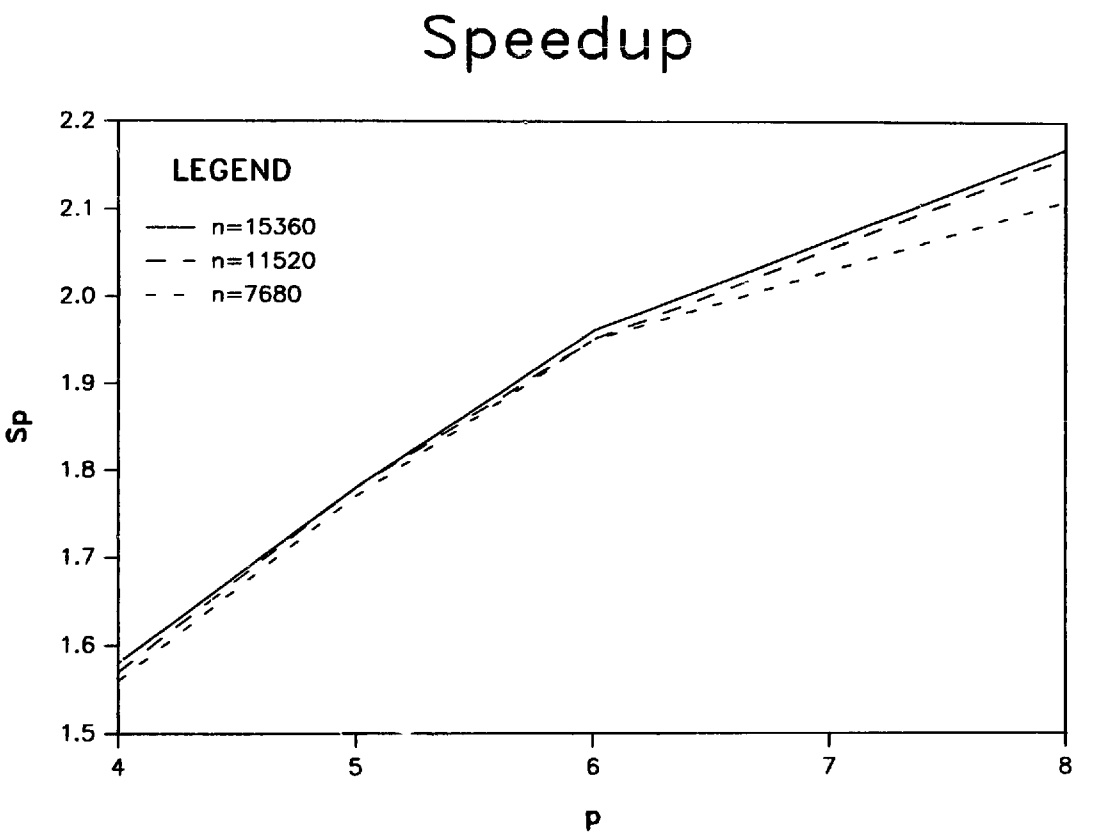
Speedup
LEGEND
n=15360
n=11520
n=7680
Sp
p
2.2
2.1
2.0
1.9
1.8
1.7
1.6
1.5
4
5
6
7
8

Fig. 1.

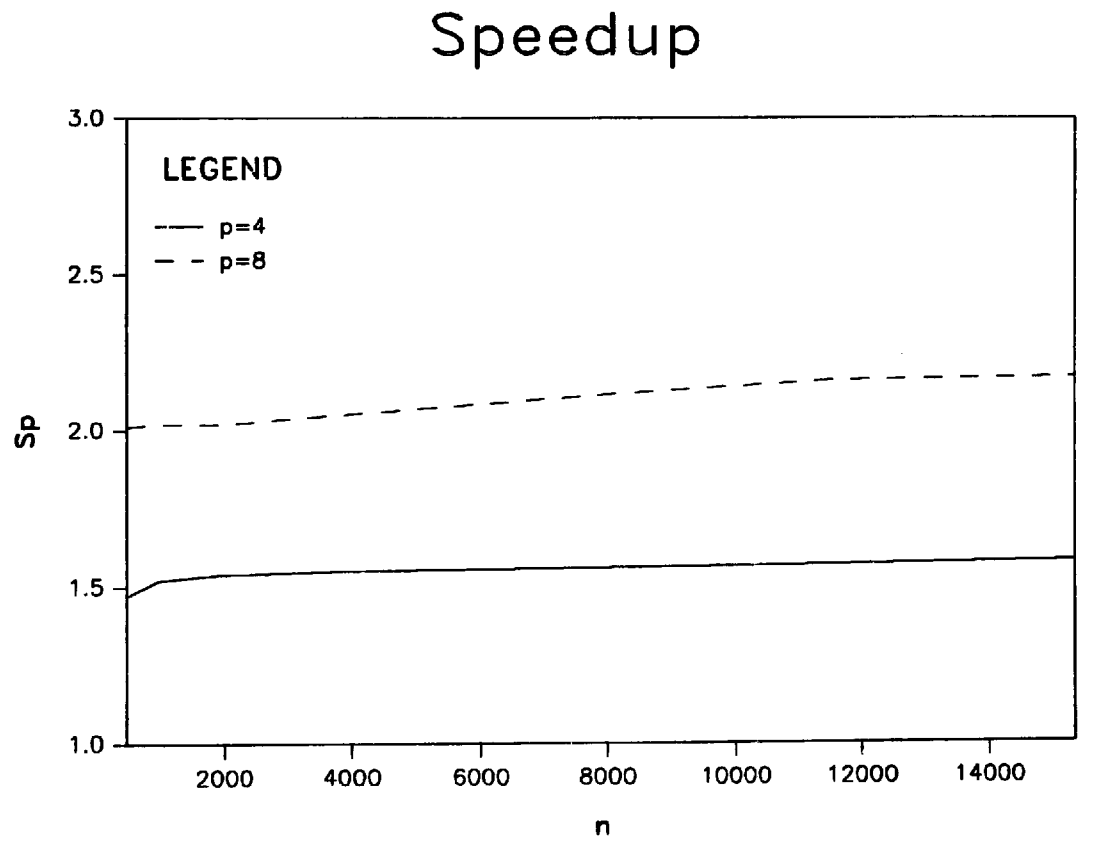
Speedup
LEGEND
p=4
p=8
Sp
n
3.0
2.5
2.0
1.5
1.0
2000
4000
6000
8000
10000
12000
14000

Fig. 2.

results (see *Fig. 1*). For a fixed n, the speedup increases as the number of processor increases. It is nearly a linear function of p. This proves the optimal performance of the proposed algorithm. For a fixed p, the efficiency increases with n. Two cases ($p = 4$ and $p = 8$) are plotted in *Fig. 2*. For $p = 4$, the theoretical speedup should be 1.6. The practical results are between 1.47 and 1.58. They are within 91.99 and 98.54% of the predicted results. For $p = 8$, the theoretical speedup should be 3.2. The practical results are between 2.04 and 2.17. They are within 62.77 and 67.83% of the predicted results. This is because that the influence of the sequential computation in this algorithm and the system overhead increase with p. These results are consistent with the theoretical analysis in the last section.

The sequential part of the program PTRD is by no means restricted to the execution on one processor. In fact, the calculation of $g_{(l+1)*k}$ can be done in parallel. For large values of p, this calculation can be parallelized recursively.

5. Conclusions

A new parallel method for solving symmetric tridiagonal eigenvalue problems is proposed and analyzed in terms of its performance on the MIMD environment. Compared with previous results, this method has a linear speedup, which is greater than the existing results, and the data communication between processors is less than that required for other methods. The experimental results are shown to achieve the expected performance and are of testimony to the efficiency of the parallel algorithm. The results also reflect that the overhead and the influence of sequential fraction in a parallel algorithm of an MIMD computation becomes less significant as the ratio between the problem size and the number of processors increases.

In conclusion, we have noted that the segmentation (or partitioning) method can be used in many numerical computation problems for parallel execution. One problem is how to decide the partitioning strategy to minimize the overhead. Much research work needs to be done in the future.

References

[1] R.H. Barlow and Evans, A parallel organization of the bisection algorithm, *Comput. J.* 22, (1978) 267–269.
[2] R.H. Barlow, D.J. Evans and J. Shanehchi, Parallel multisection applied to the eigenvalue problem, *Comput. J.* 26 (1983) 6–9.
[3] B.L. Buzbee, G.H. Golub and C.W. Neilson, On direct methods for solving Poisson's equations, *SIAM J. Numer. Anal.* 7 (1970) 627–656.
[4] J.J. Dongarra and D.C. Sorensen, Fully parallel algorithm for the symmetric eigenvalue problem, Government Report DE86007553.
[5] J.J. Dongarra, Squeezing the most out of eigenvalue solvers on high-performance computers, Government Report DE85007569.
[6] D.J. Evans, Design of numerical algorithms for Supercomputers, in: J.T. Devreese et al., eds., *Scientific Computing on Supercomputers,* (1989) 101–131.
[7] Q.S. Gao, X. Zhang and J.M. Wang, Another efficient parallel algorithm for recurrence problem, *Comput. Appl. Applied Math. China* 4 (1978) 11–15.
[8] M.T. Health and D.C. Sorensen, Pipelined Givens method for computing the QR factorization of a sparse matrix, Government Report DE85009772.
[9] R.W. Hockney and C.R. Jesshope, *Parallel Computers, Architecture, Programming and Algorithms* (Adam Hilgor, Bristol, 1988).
[10] D.J. Kuck, *The Structure of Computers and Computations, vol. 1* (Wiley, New York, 1978).
[11] D.J. Kuck and A.H. Sameh, Parallel computation of eigenvalues of real matrices, *IFIP Congress, 1971* Vol. 2 (North-Holland, Amsterdam, 1972) 1266–1272.

[12] S. Lakshmivarahan and S.K. Dhall, *Analysis and Design of Parallel Algorithm Arithmetic and Matrix Problems* (McGraw-Hill, New York, 1990).
[13] B. Philippe, Solving large sparse eigenvalue problems on supercomputers, NASA CR 185421, NASA Ames Research Center, 1988.
[14] Sequent Computer Systems, Inc., Sequent Guide to Parallel Programming, 1986.
[15] J.H. Wilkinson, *The Eigenvalue Problem* (Oxford Univ. Press, New York, 1988).