This paper describes implementations of Cuppen's method, bisection, and multisection for the computation of all eigenvalues and eigenvectors of a symmetric tridiagonal matrix on a distributed-memory hypercube multiprocessor. Numerical results and timings for Intel's iPSC are presented. Cuppen's method is the most accurate of the three. Near maximal speedups are demonstrated for Cuppen's method when little deflation occurs at intermediate steps, but speedups are significantly reduced when deflation leads to processor load imbalance. Bisection with inverse iteration is seen experimentally to be the fastest method sequentially and in parallel. The independent tasks comprising this approach lead to high parallel efficiency. The relative expected performance of parallel multisection is shown analytically to be problem dependent with arithmetic inefficiency arising in a wide class of problems. Moderate speedups are observed experimentally.

Solving the Symmetric Tridiagonal Eigenvalue Problem on the Hypercube

Ilse C.F. Ipsen and Elizabeth R. Jessup

Research Report YALEU/DCS/RR-548
July 1987

The work presented in this paper was supported by the Office of Naval Research under contracts N000014-82-K-0184 and N00014-85-K-0461. Part of this work was performed while the authors were in residence at the Computer Science and Systems Division of AERE Harwell, United Kingdom.

Table of Contents

1. Introduction

Several algorithms have been conceived specifically for determining eigenvalues and eigenvectors of symmetric, tridiagonal matrices on conventional uniprocessors. These include the QR algorithm [8], divide-and-conquer strategies [11, 22], and bisection with Sturm sequences coupled with inverse iteration [33]. QR and Cuppen's divide-and-conquer method are often used to compute all eigenvalues and eigenvectors of the matrix, while bisection with inverse iteration is normally used when only a few of the eigenvalues and corresponding eigenvectors are required. Of these schemes, the shifted QR algorithm alone does not seem to have an efficient parallel implementation: the shift computation and the application of rotations cannot be overlapped. In contrast, Cuppen's method and the generalization of bisection known as multisection have been implemented with significant speedups on shared-memory multiprocessor architectures and their simulators [6, 12, 23] and on the grid-based, bit-sliced ICL DAP [1, 2]. In this paper, we will investigate the suitability of the distributed-memory hypercube architecture as represented by Intel's iPSC for the parallel solution of the symmetric tridiagonal eigenvalue problem.

Three methods for solving the symmetric tridiagonal eigenproblem on a hypercube multiprocessor will be presented. The first is a parallel version of Cuppen's method; the second is bisection together with inverse iteration, and the third is multisection with inverse iteration. Unlike most previous parallel eigensolvers, the present implementations operate on a multiprocessor architecture in which each processor has direct access to its own local memory only. Without common memory, exchange of data between processors is accomplished through message passing. An algorithm is implemented in parallel, in general, by dividing the work required into parts or *tasks*, some of which can be executed simultaneously. On a shared-memory machine, tasks can be maintained in a queue and the task at the head of the queue allotted to the first available processor. In a message-passing environment, the assignment of one processor as a queue-manager represents a potential bottleneck. Instead, processors can pass messages to inform one another about the progress of their tasks and, thereby, coordinate further task allocation. In a static as opposed to a dynamic allocation, the time of computation of and the processor assigned to each task is predetermined. In the implementations to be discussed, tasks are assigned statically to the processors. This strategy permits simplicity of programming and reduction of scheduling overhead. Nevertheless, the cost of communication between processors is often non-negligible. For this reason, tasks must be apportioned so that communication does not occupy a significant portion of the total computation time. Solution of this scheduling problem (*i.e.*, the partitioning of the algorithm into tasks and assigning of the tasks to processors) is the basis for development of an efficient hypercube program.

Several properties of the hypercube enable it to simulate efficiently those architectures whose processor interconnections form, among others, rings, meshes, and trees [7, 9, 30, 34]. Because every processor in a p-processor hypercube is connected to $\log_2 p$ others so that no two processors are at a distance greater than $\log_2 p$, global communication or broadcasting of information from one processor to all others is fast. Restricting processors to access only local memory prevents contention problems associated with the limited memory bandwidth of shared-memory machines. The richness of its interconnection topology together with the potentially large size of its distributed memory recommend the hypercube as a worthwhile

device for the solution of large numerical problems.

A more detailed characterization of the hypercube and algorithms for data transmission are provided in Section 2. Cuppen's method, bisection, and multisection are outlined in Sections 3, 4, and 5, respectively; timings for the methods on the Intel iPSC are presented in the respective sections.

For purposes of estimating computation times on the hypercube, it is assumed throughout this paper that time $\beta + k\tau$ is required to send a vector of length k from one processor to one of its neighbors. β is the communication startup time, and τ is the time to transfer one vector element. ω is the time needed for a flop which is defined as in [19] to be the operations needed to determine

$$c_{ij} = c_{ij} + a_{ik}b_{kj}.$$

ω thus includes the time for a floating point multiplication and addition as well as time for some pointer manipulation. If the array elements are taken to be real, double precision floating point numbers, then $\frac{\beta}{\omega} \approx 10$ and $\frac{\beta}{\tau} \approx 125$ on the iPSC running operating system release R3.0.

2. The Hypercube Multiprocessor

In order to implement Cuppen's method and multisection on the hypercube, it is necessary to devise efficient ways of exchanging data among processors. Algorithms are presented for transmitting data from all processors to all others, for matrix multiplication, and for orthogonalization of a set of vectors.

2.1. Characterization of the Hypercube

A hypercube of dimension d, or d-cube, is a graph consisting of $p = 2^d$ nodes, $d \geq 0$, and is defined recursively as follows. A 0-cube is composed of a single node, while for $d > 0$, a d-cube is obtained by adding edges between corresponding nodes of two $(d-1)$-cubes. Figure 1, for example, depicts a 3-cube. From this construction, it can be seen that a d-cube is composed of 2^{d-i} i-cubes, for $0 \leq i \leq d$.

Alternatively, a d-cube may be defined by associating with each node a binary label of length d so that any edge connects two nodes whose labels differ in exactly one bit. In a d-cube, every node is connected to d others making a total of $d2^{d-1}$ edges. Assignment of labels according to a binary reflected Gray code [16, 26] induces a linear order on the nodes according to which they may be numbered $0, 1, \ldots, 2^d - 1$. Embeddings into the hypercube of toroids or one- and two-dimensional arrays are based on such Gray codes [30].

A hypercube *multiprocessor* of dimension d is made up of $p = 2^d$ processors located at the nodes and $d2^{d-1}$ processor interconnections corresponding to the edges of a d-cube. The eigenvalue codes were implemented on an Intel iPSC/d5M hypercube multiprocessor. This machine consists of 32 identical node processors each capable of communicating directly with five neighboring processors. A node can communicate with only one of its neighbors at a time and does so by issuing a *send* communication primitive to initiate a message transfer or a *receive* primitive to intercept one. Messages arriving at a node are held in a queue until selected via a receive command. Each processor has 4.5 Mbytes of local memory.

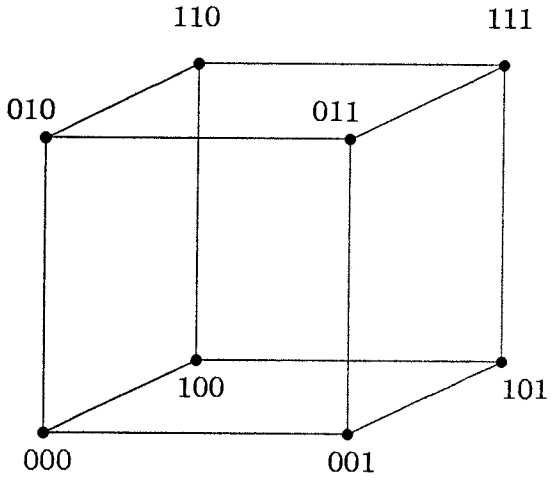

Figure 1: A 3-cube with numbered nodes.

An additional, separate processor serves as the cube manager or host machine. It can communicate with all nodes via a global bus. For our problems, the host is used for downloading code onto, passing initial data to, and accumulating final results from the node processors but does not enter into the computation in any other way. Once started, all nodes and the cube manager operate independently unless explicitly synchronized by communication requirements. In the remainder of the paper, a node processor is named by the decimal value of its binary identifier, while the cube manager is known as the host. The term *hypercube* refers to both graph and multiprocessor.

2.2. Data Transmission on the Hypercube

The *Alternate Direction Exchange Algorithm (ADE)* [29] appeals to the recursive structure of the hypercube to bring about a total exchange of the data held by the processors of a cube. All processors in a d-cube whose binary labels agree in the same bit position form a $(d-1)$-cube. Because there are d bits in the binary label, there are d ways of dividing a d-cube into two $(d-1)$-cubes. In the 3-cube of Figure 1, the subcubes are defined by the node numbers. All nodes with a zero in the leftmost position of their labels form a 2-cube (the front face) while all nodes with a one in the leftmost position make up a second 2-cube (the back face). Similarly, the upper and lower faces are 2-cubes with nodes having a common value in the center position, and the left and right faces are 2-cubes sharing the same rightmost value.

The process of broadcasting a vector of length k from one processor to all others in

a d-cube requires d steps of data transmission where the amount of data doubles in size during each step.

Algorithm ADE.
For $l = 0, 1, \ldots, d-1$

1. Processors form two $(d-1)$-cubes S_0 and S_1 according to the value of bit l in their binary labels.
2. Processors in corresponding positions of S_0 and S_1 exchange vectors of length $2^l k$.
3. Each processor concatenates its own vector with the one received to form a vector of length $2^{l+1}k$.

The time to perform algorithm ADE on a d-cube comes to

$$\begin{aligned} T^d_{ADE} &= 2[(\beta + k\tau) + (\beta + 2k\tau) + \cdots + (\beta + 2^{d-1}k\tau)] \\ &= 2[d\beta + (2^d - 1)k\tau]. \end{aligned}$$

The factor of 2 reflects the fact that vectors are not sent in both directions at once on a node-to-node link. The above algorithm is also used when data are broadcast within a subcube S of dimension $i \leq d$. S is then made up of all processors of the d-cube whose binary labels agree in exactly the same $d - i$ bit positions. The corresponding communication time is

$$T^i_{ADE} = 2[i\beta + (2^i - 1)k\tau].$$

Broadcasting within 2^{d-i} i-cubes can occur simultaneously without interference.

2.3. Matrix Multiplication on the Hypercube

The parallel implementations of Cuppen's method, bisection, and multisection take advantage of the fact that the processors along with their interconnections form a ring within the hypercube (see Section 2.1) and that subcubes correspond to contiguous sequences within the ring. A matrix may be stored in the hypercube by situating blocks of adjacent columns in neighboring processors. For this and all other ring-oriented procedures discussed in this paper, it is assumed that the ring is embedded according to a Gray code ordering of processors so that communication takes place only between neighboring processors in the cube. A processor is referred to by its position in the ring, *i.e.*, the processor with Gray code index j has left and right neighbors $j-1$ and $j+1$, respectively, where all processor identifiers in a d-cube are taken modulo $p = 2^d$.

The following algorithm, *Ring Matrix Multiplication (RMM)*, performs the multiplication $C = AB$ of two $N \times N$ matrices A and B both distributed by columns among the processors of a d-cube. The order N is taken to be a multiple of the number of processors, $N = k2^d$. RMM takes full advantage of the distributed memory on the cube and keeps the amount of storage required per processor to a minimum. Initially, processor j, $0 \leq j \leq 2^d - 1$, contains the columns $jk, \ldots, (j+1)k - 1$ of A and of B, and upon completion columns $jk, \ldots, (j+1)k - 1$ of A, B, and C. During the formation of C, the columns of B remain in their original places while the columns of A are passed cyclically from processor to processor along the ring, overwriting the previously-held columns of A in each processor.

Let $\hat{B}_{ij}$ denote the $k \times k$ block-matrix in position (i,j) of B, $0 \leq i,j \leq 2^d - 1$, and

$$\hat{B}_j = [\hat{B}_{0j}^T \ldots \hat{B}_{2^d-1,j}^T]^T$$

be the k columns $jk, \ldots, (j+1)k - 1$ of B. Algorithm RMM proceeds as follows. Indices should be taken modulo 2^d.

Algorithm RMM.

In parallel, do on all processors j, $0 \leq j \leq 2^d - 1$:

$\hat{C}_j = 0$

For $i = 0, \ldots, 2^d - 1$:

1. compute $\hat{C}_j = \hat{C}_j + \hat{A}_{j-i}\hat{B}_{j-i,j}$
2. send $\hat{A}_{j-i}$ to processor $j+1$
3. receive $\hat{A}_{j-i-1}$ from processor $j-1$

The arithmetic time at each iteration comes to $2^d k^3 \omega$ and the communication time to $\beta + 2^d k^2 \tau$, giving a total of

$$T_{RMM}^d = 2^{2d} k^3 \omega + 2^d(\beta + 2^d k^2 \tau).$$

(See also [15].) This algorithm also applies to the multiplication of two matrices of order $k2^i$ on an i-cube, $i \leq d$, whence the time becomes

$$T_{RMM}^i = 2^{2i} k^3 \omega + 2^i(\beta + 2^i k^2 \tau).$$

2.4. Modified Gram-Schmidt Orthogonalization

The modified Gram-Schmidt procedure is employed to transform a set of linearly independent vectors into a set of orthonormal vectors. A real matrix A having m linearly independent columns of length N is factored into the product of a matrix $Q \in \Re^{N \times m}$ with orthonormal columns and an upper triangular matrix $R \in \Re^{m \times m}$. The kth column of Q and the kth row of R are computed at the kth step of the modified Gram-Schmidt process [19]. Orthogonalization is necessary when inverse iteration applied to poorly separated eigenvalues produces eigenvectors that, while linearly independent, are not orthogonal [33].

Algorithm MGS describes an implementation of modified Gram-Schmidt on a p processor hypercube when $m \leq p$. Columns are assigned to processors in an embedded linear array so that processor j has column j. The orthogonalization procedure is pipelined by passing computed orthogonal columns along the array for use in orthogonalizing remaining columns. The columns of A are overwritten by the orthonormal columns of Q.

Algorithm MGS ($m \leq p$).

In parallel, do on all processors j, $0 \leq j \leq m - 1$:

1. for $k = 0, \ldots, j-1$

1.1. receive the kth column $(a_{1k},\ldots,a_{Nk})^T$ from processor $j-1$
1.2. if $k < m-1$, send column k to processor $j+1$
1.3. compute $r_{kj} = \sum_{i=0}^{N-1} a_{ik}a_{ij}$
1.4. for $i = 0,\ldots,N-1$, $a_{ij} = a_{ij} - a_{ik}r_{kj}$

2. normalize column j: $(a_{1j},\ldots,a_{Nj})^T = \frac{(a_{1j},\ldots,a_{Nj})^T}{\|(a_{1j},\ldots,a_{Nj})^T\|_2}$
3. if $j < m-1$, send column j to node $j+1$

Note that the processor holding column i requires access to orthogonalized columns 0 through $i-1$ in order to orthogonalize column i. Thus, processor i remains idle until it receives column 0. As $m \leq p$, processor i is delayed by the time needed for processor 0 to normalize column 0 plus the time for column zero to pass through i communication links or $i(\beta + N\tau)$. Upon receipt of column 0, processor i orthogonalizes column i with respect to column 0. This receipt and orthogonalization process is repeated by processor i for columns 1 through $i-1$. Note that the orthogonalization in steps 1.3 and 1.4 and the normalization in step 2 each require approximate time $2N\omega$. Thus, once processor i has received and used column 0 in step 1, it must wait an additional $2N\omega$ to account for the time needed by processor 1 to normalize column 1. A waiting time of $2N\omega$ is thus accrued by processor i for each of the processors $0,\ldots,N-1$. When $\beta + N\tau < 2N\omega$, time for communication of vectors to processor i from processors $1,\ldots,i-1$ overlaps this additional idle time and need not be considered further. For the iPSC parameters given in Section 1, $\beta + N\tau < 2N\omega$ for all $N > 5$. Processor i spends time $2iN\omega$ completing steps 1 and 2. The total time for Algorithm MGS is equal to the time needed by the processor holding the last column, and

$$T_{MGS} = [2(m-1)-1]2N\omega + (m-1)(\beta + N\tau), \quad \text{for } m \leq p.$$

When $m > p$, a ring of processors replaces the linear array. Columns may be assigned in any order but for the purposes of this analysis are numbered so that processor j has columns $j, j+p,\ldots,j+\nu p \leq m-1$. Communication proceeds in a fashion similar to that of Algorithm MGS with processor indices taken modulo p; however, column originally held in processor j is passed only as far around the ring as needed in step 2 and does not return to processor j. No processor spends more than $(p-1)(\beta + N\tau)$ waiting for column 0. Upon receipt of column k, processor j repeats steps 1.3 and 1.4 for any of the columns $j,\ldots,j+\nu p \leq m-1$ with index greater than k. As soon as step 1 has been completed, that column is normalized and sent on; thus, one normalized vector begins the circuit every $4N\omega$.

The time to complete orthogonalization is again equal to the total time required by the processor holding column $m-1$. This time, in turn, is dependent on the number of columns held by that processor. When m is sufficiently large, the slowest processor will have enough orthogonalization tasks that it never need wait for additional normalization steps performed by other processors as described above. At worst, (for example, when $m = p+1$) the processor holding the mth column, will have to wait idle through $m-2$

normalizations. The additional communication time overlaps computation, and

$$T_{MGS} < \left(\lceil \frac{m}{p} \rceil m + m - 1 \right) N\omega + (p-1)(\beta + N\tau), \quad \text{for } m > p.$$

3. Cuppen's Method

3.1. Sequential Algorithm

The first scheme to be discussed is a divide-and-conquer method described by Cuppen to find all the eigenvalues and corresponding eigenvectors of any symmetric tridiagonal matrix [11]. The method is based on the fact that a symmetric tridiagonal matrix T of even order N can be divided into a pair of equal-sized symmetric tridiagonal submatrices as follows

$$\begin{aligned} T &= \begin{pmatrix} \hat{T}_0 & \alpha e_{N/2} e_1^T \\ \alpha e_1 e_{N/2}^T & \hat{T}_1 \end{pmatrix} \\ &= \begin{pmatrix} T_0 & \\ & T_1 \end{pmatrix} + \begin{pmatrix} \theta\alpha e_{N/2} e_{N/2}^T & \alpha e_{N/2} e_1^T \\ \alpha e_1 e_{N/2}^T & \theta^{-1}\alpha e_1 e_1^T \end{pmatrix} \qquad (3.1) \\ &= \begin{pmatrix} T_0 & \\ & T_1 \end{pmatrix} + \theta\alpha \begin{pmatrix} e_{N/2} \\ \theta^{-1} e_1 \end{pmatrix} \begin{pmatrix} e_{N/2}^T & \theta^{-1} e_1^T \end{pmatrix}, \end{aligned}$$

where α is the off-diagonal element of T at position $\frac{N}{2}$, e_i is the ith unit vector of length $\frac{N}{2}$, and T_0 and T_1 are of order $\frac{N}{2}$. The sign and magnitude of θ are selected to ensure that subdivision of the matrix can be performed at this position without cancellation [12]. The original problem has now been split into two eigenproblems of half its order.

If the solutions to the two smaller eigensystems are $T_0 = X_0 D_0 X_0^T$ and $T_1 = X_1 D_1 X_1^T$, then

$$T = Q \left[D + \alpha\theta \begin{pmatrix} l_0 \\ \theta^{-1} f_1 \end{pmatrix} \begin{pmatrix} l_0^T & \theta^{-1} f_1^T \end{pmatrix} \right] Q^T$$

where

$$Q = \begin{pmatrix} X_0 & \\ & X_1 \end{pmatrix}, \quad D = \begin{pmatrix} D_0 & \\ & D_1 \end{pmatrix},$$

$l_0^T = e_{\frac{N}{2}}^T X_0$ is the last row of X_0, and $f_1^T = e_1^T X_1$ is the first row of X_1. To solve the eigenproblem for T, it is necessary to find the eigenvalues and eigenvectors of the diagonal matrix D plus a rank-one correction, $D + \rho z z^T = Q^T T Q$, where $z^T = \sqrt{\frac{\alpha\theta}{\rho}} \begin{pmatrix} l_0^T & \theta^{-1} f_1^T \end{pmatrix}$, and ρ is selected so that $\| z \|_2 = 1$. For the remainder of the paper, it is assumed that T is unreduced (*i.e.*, its off-diagonal elements are non-zero). If not, T would consist of a direct product of disjoint, lower order matrices whose eigensolutions could be determined independently.

Computation of the eigensystem of T is accomplished via a rank-one updating technique described in [18]. Namely, if all elements of z are non-zero and if the diagonal elements of D are distinct and in sorted order $d_0 > d_1 > \cdots > d_{N-1}$, then the eigenvalues of $D + \rho z z^T$ are the roots λ_i, $i = 0, 1, \ldots, N-1$, of the *secular equation* [18]

$$w(\lambda) = 1 + \rho z^T (D - \lambda I)^{-1} z. \tag{3.2}$$

This rational function takes the form shown in Figure 1 of [10].

Note that λ_i is bracketed by d_i and d_{i+1}, so the roots of $w(\lambda)$ are readily found using a rational interpolation technique given in [10]. This method is quadratically and monotonically convergent from one side of the root thus ensuring that the root λ_i can be extracted from interval (d_i, d_{i+1}) without need for safeguarding. Once the eigenvalues of $D + \rho z z^T$ have been found, the corresponding eigenvectors are computed from

$$u_i = \frac{(D - \lambda_i I)^{-1} z}{\| (D - \lambda_i I)^{-1} z \|_2}. \tag{3.3}$$

If U is the matrix with columns $u_0, u_1, \ldots, u_{N-1}$ and $\Lambda = diag(\lambda_0, \ldots, \lambda_{N-1})$, the eigendecomposition of the original matrix may now be expressed as the matrix product

$$T = QU\Lambda U^T Q^T.$$

If the elements of D are not sorted in descending order as required above, it is necessary to consider instead the matrix $J^T D J$ where J is an appropriate permutation matrix. In addition, the above procedure is dependent on the distinctness of the elements $\{d_j\}$. Although an unreduced tridiagonal matrix of order N does itself have N distinct eigenvalues, the intermediate matrix D may not. For example, if

$T = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 3 & 1 & 0 \\ 0 & 1 & 3 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}$, then $\alpha = 1$, $T_0 = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$, and $T_1 = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$. Both T_0 and T_1 have eigenvalues 1 and 3, so $D = \begin{pmatrix} 3 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$, and $d_0 = d_1, d_2 = d_3$.

In the case of multiple values $d_l = d_{l+1} = \cdots = d_{l+k}$, the eigenvalues cannot be determined from the secular equation. Rather, the eigenvector basis is rotated to zero out the elements $z_{l+1}, \ldots, z_{l+k}$ corresponding to the multiple elements $d_{l+1} = \cdots = d_{l+k}$: a product of plane rotations G_l is applied so that

$$G_l (z_l, z_{l+1}, \cdots, z_{l+k})^T = (z_l', 0, \cdots, 0)^T.$$

The eigenvalue corresponding to a zero element z_j' of the rotated vector remains unchanged ($\lambda_j = d_j$), and the corresponding eigenvector may be chosen as the appropriate unit vector of order N ($u_j = e_j$) [11]. Therefore, multiple values along the diagonal of D result in

a significant reduction in the work required to compute the eigensystem of $D + \rho zz^T$. In addition, small elements of z corresponding to distinct elements of D lead to similar savings. Numerical experiments have confirmed that the increase in speed due to this deflation is substantial [12]. Representing the product of all rotations in the matrix G, the matrix T is expressed as

$$T = QJG^T U \Lambda U^T G J^T Q^T = X \Lambda X^T, \tag{3.4}$$

where $U\Lambda U^T$ is the eigendecomposition of $GJ^T(D + \rho zz^T)JG^T$. The eigenvalues of T are the diagonal elements of Λ while the eigenvectors of T are the columns of $X = QJG^T U$.

3.2. Parallel Algorithm

The sequential divide-and-conquer technique derives from the fact that a symmetric tridiagonal matrix T can be divided into a pair of symmetric tridiagonal matrices $\hat{T}_0$ and $\hat{T}_1$ as in equation (3.1). But just as T was subdivided so too can be T_0 and T_1. Renaming the variables T, α, and b to T_{20}, α_{20}, and b_{20}, respectively, the resulting matrix $T \equiv T_{20}$ has the form

$$T \equiv T_{20} = \left[\begin{array}{c|c} \begin{array}{c|c} T_{00} & 0 \\ \hline 0 & T_{01} \end{array} + \alpha_{10} b_{10} b_{10}^T & 0 \\ \hline 0 & \begin{array}{c|c} T_{02} & 0 \\ \hline 0 & T_{03} \end{array} + \alpha_{11} b_{11} b_{11}^T \end{array}\right] + \alpha_{20} b_{20} b_{20}^T. \tag{3.5}$$

The subdivision or "tearing" process can be repeated and rank-one updating procedures applied recursively to determine the eigensystem of T. At the ith division, a submatrix T_{ij} is split into

$$T_{ij} = \begin{pmatrix} T_{i-1,2j} & \\ & T_{i-1,2j+1} \end{pmatrix} + \alpha_{ij} b_{ij} b_{ij}^T, \quad 0 \le j \le 2^{d-i} - 1.$$

It is this recursive nature of Cuppen's method that suggests its suitability to parallel implementation on the hypercube architecture. With this motivation, a high-level, top-down description of a parallel version of Cuppen's method for the hypercube is given as Algorithm C1.

Algorithm C1: Solution of Eigenproblem of Order $N = k2^d$ on a d-cube.

Recursively divide the matrix d times so that processor j contains the submatrix T_{0j} of order k, $0 \le j \le 2^d - 1$.

1. Step 0. Each processor (0-cube) independently computes the eigensystem of its order-k submatrix.
2. Step i, $1 \le i \le d$. Each $(i-1)$-cube pairs with another $(i-1)$-cube to form an i-cube by exchanging information about the eigensystems of the matrices $T_{i-1,2j}$ and $T_{i-1,2j+1}$

of order $k2^{i-1}$ computed in step $i-1$. Each i-cube independently computes the eigensystem for its matrix T_{ij} of order-$k2^i$. The 2^{d-i} i-cubes at step i are enumerated by the index j, $0 \leq j < 2^{d-i}$.

Note that the number of updating steps in Algorithm C1 is equal to the dimension of the hypercube. At step i, 2^{d-i} i-cubes independently solve eigensystems of order $k2^i$. Upon completion of step d, each processor contains k of the $N = k2^d$ eigenvalues of the original order $N = k2^d$ matrix T as well as the k corresponding eigenvectors (of length N). Algorithm C1 can now be refined to explain in greater detail the assignment of computation to processors.

Each processor starts with the solution of an eigensystem of order k and during subsequent steps is responsible for updating k eigenvalues and k eigenvectors, thereby doubling the length of the latter during each step. Thus, the jth processor in the subcube contains eigenvalues $\lambda_{jk}, \lambda_{jk+1}, \ldots, \lambda_{(j+1)k-1}$ of the matrix T as well as the corresponding eigenvectors. Figure 2 illustrates the above by depicting the communication pattern at each step of the algorithm on a hypercube of dimension 3. The labelled squares denote individual processors (the 0-cubes of step 0). A box containing 2^i processors implies that, at step i, subcubes of dimension i independently compute the needed eigensystems. All processors belonging to the same i-cube must communicate with one another during the solution step i.

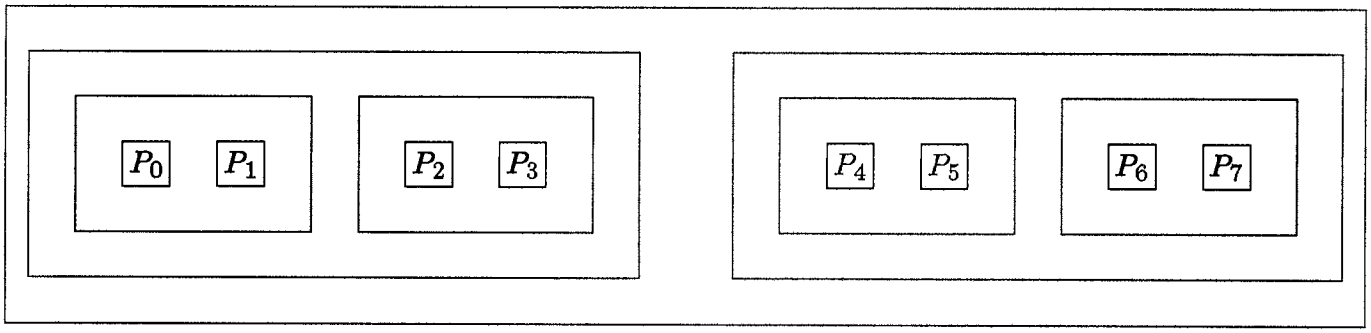

Figure 2: Formation of subcubes for data transmission during Cuppen's method.

The division process of the algorithm can be well-illustrated by means of an example for a 3-cube. A matrix $T \equiv T_{30}$ of order $N = k2^3$ is recursively divided into eight order-k tridiagonal matrices $T_{00}, T_{01}, \ldots, T_{07}$, and matrix T_{0j} is assigned to processor j, $0 \leq j \leq 7$. The allocation of submatrices to processors is given in Table 1. There, the entries T_{ij} are those matrices whose eigensystems are computed in step i by subcube j. The brackets distinguish the subcubes occupied by each eigensystem.

During step 0, processor j (a 0-cube) computes the eigensystem (Λ_{0j}, X_{0j}) of the matrix T_{0j}. Because each rank-one updating step requires the eigensystems of two smaller

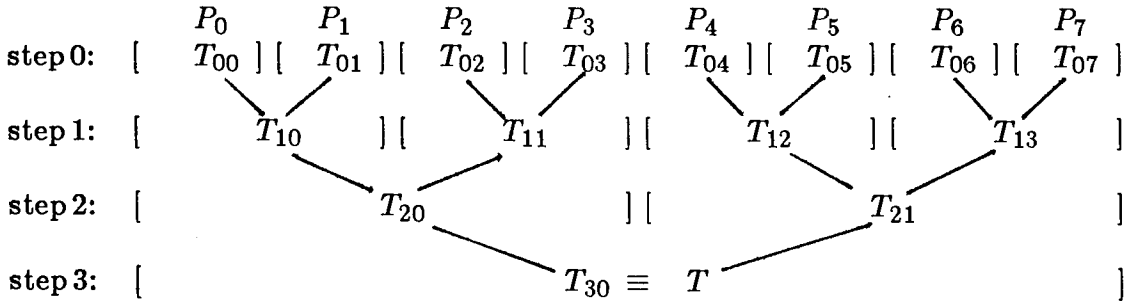

Table 1: Assignment of submatrices to the processors of a d-cube during Cuppen's method.

matrices, processors must pair up in step 1 and exchange information within 1-cubes to compute the eigensystems of the four order-$2k$ matrices $T_{10}, \ldots, T_{13}$. Figure 3.2 shows that after exchanging information about the eigensystems of matrices T_{00} and T_{01}, processors 0 and 1 can together compute the eigensystem (Λ_{10}, X_{10}) of T_{10}. Processor 0 is left with the leading k eigenvalues and eigenvectors of T_{10}, while processor 1 holds the trailing k. The remaining steps proceed in a similar fashion until, at the end of step 3, each of the eight processors contains k eigenvalues and k eigenvectors of length $8k$ of the original matrix $T \equiv T_{30}$. For $0 \leq j \leq 7$, processor j holds eigenvectors indexed $jk, \ldots, (j+1)k - 1$.

To begin the solution of the eigenvalue problem by Cuppen's method, each node requires a sequence of diagonal and off-diagonal elements of the matrix T. For the sample problem of Figure 3 on a cube dimension 3, node 0 needs the submatrix T_{00} as well as the off-diagonal elements α_{10}, α_{20} and α_{30}. For the purposes of this paper, it is assumed that each node contains the needed matrix elements. Algorithm C2 details the steps in the determination of all eigenvalues and eigenvectors of a matrix T of order $N = k2^d$ on a d-cube. Note that only parts $i.1$ and $i.4$ of step i, $1 \leq i \leq d$, require data communication. In Algorithm C2, italics distinguish comments from instructions.

Step 0: Compute the eigensystems of T_{00}, $T_{01}, \ldots, T_{07}$.
Step 1: Compute the eigensystems of

$$T_{10} = \begin{pmatrix} T_{00} & \\ & T_{01} \end{pmatrix} + \alpha_{10} b_{10} b_{10}^T,$$

$$\vdots$$

$$T_{13} = \begin{pmatrix} T_{06} & \\ & T_{07} \end{pmatrix} + \alpha_{13} b_{13} b_{13}^T.$$

Step 2: Compute the eigensystems of

$$T_{20} = \begin{pmatrix} T_{10} & \\ & T_{11} \end{pmatrix} + \alpha_{20} b_{20} b_{20}^T = \begin{pmatrix} T_{00} & & & \\ & T_{01} & & \\ & & T_{02} & \\ & & & T_{03} \end{pmatrix} + \begin{pmatrix} \alpha_{10} b_{10} b_{10}^T & \\ & \alpha_{11} b_{11} b_{11}^T \end{pmatrix} + \alpha_{20} b_{20} b_{20}^T$$

and

$$T_{21} = \begin{pmatrix} T_{12} & \\ & T_{13} \end{pmatrix} + \alpha_{21} b_{21} b_{21}^T = \begin{pmatrix} T_{04} & & & \\ & T_{05} & & \\ & & T_{06} & \\ & & & T_{07} \end{pmatrix} + \begin{pmatrix} \alpha_{12} b_{12} b_{12}^T & \\ & \alpha_{13} b_{13} b_{13}^T \end{pmatrix} + \alpha_{21} b_{21} b_{21}^T$$

Step 3: Compute the eigensystem of

$$T \equiv T_{30} = \begin{pmatrix} T_{20} & \\ & T_{21} \end{pmatrix} + \alpha_{30} b_{30} b_{30}^T.$$

Figure 3: Submatrices occurring during the recursive construction of the eigensystem of $T \equiv T_{30}$ on a 3-cube.

Algorithm C2: Solution of eigenproblem of order $N = k2^d$ on a d-cube.

Recursively divide the matrix $T \equiv T_{d0}$ d times and allocate submatrix T_{0j} and the d appropriate off-diagonal elements to processor j, $0 \leq j \leq 2^d - 1$.

Step 0:

Processor j (0-cube) computes the eigensystem (Λ_{0j}, X_{0j}) of the matrix T_{0j} of order k, the diagonal of Λ_{0j} contains the eigenvalues of T_{0j} in descending order, and the columns of X_{0j} are the corresponding eigenvectors, $0 \leq j \leq 2^d - 1$.

Step $1 \leq i \leq d$:

$\{2^{d-i}$ *i-cubes independently compute the eigensystems* (Λ_{ij}, X_{ij}) *of the matrices* T_{ij}

of order $k2^i$ using the eigensystems from step i-1:

$$T_{ij} = \begin{pmatrix} T_{i-1,2j} & \\ & T_{i-1,2j+1} \end{pmatrix} + \alpha_{ij} b_{ij} b_{ij}^T$$
$$= \begin{pmatrix} X_{i-1,2j} & \\ & X_{i-1,2j+1} \end{pmatrix} \left[\begin{pmatrix} \Lambda_{i-1,2j} & \\ & \Lambda_{i-1,2j+1} \end{pmatrix} + \rho_{ij} z_{ij} z_{ij}^T \right] \begin{pmatrix} X_{i-1,2j} & \\ & X_{i-1,2j+1} \end{pmatrix}^T.$$

Denote by S a subcube of dimension i and index j which consists of processors $j2^i$ through $(j+1)2^i - 1$. For simplicity, denote these processors by $P_0, P_1, \ldots, P_{2^i-1}$ and replace subscripts of the form $(i-1, 2j)$ with 0 and $(i-1, 2j+1)$ with 1. With the new notation, processors $P_0, P_1, \ldots, P_{2^i-1}$ compute the eigensystem (Λ, X) of

$$T = \begin{pmatrix} T_0 & \\ & T_1 \end{pmatrix} + \alpha b b^T = \begin{pmatrix} X_0 & \\ & X_1 \end{pmatrix} \left[\begin{pmatrix} \Lambda_0 & \\ & \Lambda_1 \end{pmatrix} + \rho z z^T \right] \begin{pmatrix} X_0 & \\ & X_1 \end{pmatrix}^T$$

from (Λ_0, X_0) and (Λ_1, X_1). From step $i-1$, processor P_l for $0 \le l \le 2^{i-1} - 1$ contains eigenvalues of T_0 indexed $lk, \ldots, (l+1)k - 1$ and columns $lk, \ldots, (l+1)k - 1$ of X_0. These processors form a subcube of dimension $i-1$ called S_0. Similarly, processor P_l for $2^{i-1} \le l \le 2^i - 1$ contains eigenvalues of T_1 indexed $lk, \ldots, (l+1)k - 1$ and columns $lk, \ldots, (l+1)k - 1$ of X_1. These processors form a subcube of dimension $i-1$ called S_1.}

i.1 By means of algorithm ADE, processors in S_0 and S_1 exchange their k elements of Λ_0 and Λ_1, respectively, and their k elements of the last row of X_0 or the first row of X_1, respectively, so that each processor in S contains Λ_0 , Λ_1, the last row of X_0 and the first row of X_1.

i.2 Each processor in S

i.2.*a* computes z and ρ from the last row of X_0, the first row of X_1, and α_{ij}.

i.2.*b* determines a permutation matrix J by merging the sorted sequences $\mathrm{diag}(\Lambda_0)$ and $\mathrm{diag}(\Lambda_1)$ so that the diagonal elements of $D = J^T \begin{pmatrix} \Lambda_0 & \\ & \Lambda_1 \end{pmatrix} J$ are sorted in descending order.

i.2.*c* permutes the elements of z accordingly: $z \Leftarrow J^T z$.

i.2.*d* applies the product of plane rotations G to zero out the elements in z that correspond to multiple elements in D.

i.3 Each processor P_j in S

i.3.*a* computes elements $jk, \ldots, (j+1)k - 1$ of Λ (eigenvalues of $D + \rho z z^T$) and the corresponding eigenvectors $u_{lk}, \ldots, u_{(l+1)k-1}$ according to formulae 3.2 and 3.3.

i.3.*b* updates the eigenvectors: $(v_{lk}, \ldots, v_{(l+1)k-1}) = G^T (u_{lk}, \ldots, u_{(l+1)k-1})$.

i.4 By means of Algorithm RMM, processors in S_0 and S_1 send their columns of X_0 and X_1, respectively, to all other processors in S so that processor j can determine its k columns of X via

$$(x_{lk}, \ldots, x_{(l+1)k-1}) = \left[\begin{pmatrix} X_0 & \\ & X_1 \end{pmatrix} J \right] (v_{lk}, \ldots, v_{(l+1)k-1}), \quad 0 \le j \le 2^{d-i} - 1.$$

In Algorithm C2, it is assumed that the order N of T is a multiple of number of processors in the hypercube. If this is not the case, T is recursively divided so that the orders of the smallest submatrices T_{0j} differ by no more than one.

3.3. Analytical and Experimental Results

A first order approximation to an upper bound on the time required to determine all the eigenvalues and eigenvectors of a symmetric, tridiagonal matrix of order $k2^d$ on a d-cube is obtained through examination of the steps of Algorithm C2. This section assesses the contribution of communication and of floating point operations to the time requirements of Cuppen's method on the hypercube. A quantitative analysis is given for a problem in which no deflation occurs. Experimental results showing the effects of deflation follow.

Time Requirements for Algorithm C2 (No Deflation)

This summary follows the steps of Algorithm C2, outlining at each the time needed.

Step 0:

The solution of $p = 2^d$ independent symmetric tridiagonal eigenproblems of order k done in parallel on p processors requires time proportional to $k^3\omega$.

Step $1 \leq i \leq d$:

i.1 The exchange of Λ_0 and Λ_1 among the processors of subcube S requires time T^i_{ADE} as does the alternate direction exchange of X_0 and X_1.

i.2 The computation of ρ and the subsequent normalization of vector z takes time $2^{i+1}k\omega$ plus the time for a single square root. The merging of sequences Λ_0 and Λ_1 requires 2^ik comparisons; its contribution is not counted in the total time. Likewise, the pointer manipulation for permutation of z is neglected. Because no deflation occurs, step $i.2.d$ is not performed.

i.3 Computation of k eigenvalues using formula 3.2 requires time $\kappa_i 2^i k^2 \omega$, where the value of κ_i depends on the number of iterations needed for root finding. Computation of k eigenvectors using formula 3.3 requires time $2^ik^2\omega$. As it was assumed that no deflation takes place, no time is allotted for step $i.3.b$.

i.4 Step i.4 involves the updating of the new eigenvectors by computing

$$X = \left[\begin{pmatrix} X_0 & \\ & X_1 \end{pmatrix} J\right] G^T U.$$

When no deflation occurs, $G = I$ and $V = G^TU = U$.

Each processor in the S has a copy of J but only k columns of the operands $\begin{pmatrix} X_0 & \\ & X_1 \end{pmatrix} J$ and V. The product is thus found by using Algorithm RMM with $A = \begin{pmatrix} X_0 & \\ & X_1 \end{pmatrix}$ and $B = V$. J provides the permuted column indices for A. The time required for updating, however, is less than T^i_{RMM}: the block structure of $A = \begin{pmatrix} X_0 & \\ & X_1 \end{pmatrix}$ provides savings of

both communication and computation. The columns of X_0 and X_1 are of length $2^{i-1}k$, half the length of the columns of V. Thus, communication of A is accomplished by sending only X_0 and X_1 (and not the zero off-diagonal blocks). Because columns sent around the ring need not be returned to the originating processor, each block of columns need only be passed along $2^i - 1$ links of the ring. This communication can be completed in time $(2^i - 1)(\beta + 2^{i-1}k^2\tau)$. Furthermore, multiplication by the zero elements need not be performed. Total time for step $i.4$ is then

$$(2^i - 1)(\beta + 2^{i-1}k^2\tau) + 2^{2i-1}k^3\omega.$$

The high order terms in the total time needed for Algorithm C2 are found from summing the time requirements for both arithmetic and communication in its $d+1$ individual steps. Arithmetic is performed in all $d+1$ steps of Algorithm C2. Step 0 completes in time $T_A^0 = \kappa_0 k^3\omega$. Let $\kappa = max_{1\le i\le d}\kappa_i + 1$, then, neglecting the time for square roots, the arithmetic time for step i, $1 \le i \le d$, is

$$T_A^i \le \omega(k2^{i+1} + k^2 2^i \kappa + k^3 2^{2i-1}).$$

The total arithmetic time for all steps is

$$\begin{aligned} T_A = \sum_{i=0}^{d} T_A^i &\le \kappa_0 k^3\omega + \omega \sum_{i=1}^{d}(k2^{i+1} + k^2 2^i \kappa + k^3 2^{2i-1}) \\ &\approx \omega\left[k^3(\kappa_0 + \frac{2}{3}p^2) + 2k^2\kappa p + 4kp\right] \end{aligned}$$

Expressing the total in terms of matrix order $N = k2^d = kp$,

$$T_A \approx \omega\left(\kappa_0\left(\frac{N}{p}\right)^3 + \frac{2}{3}\frac{N^3}{p} + 2\kappa\frac{N^2}{p} + 4N\right).$$

No communication occurs in step 0. Communication for step i, $1 \le i \le d$, is limited to a pair of alternate direction exchanges plus the modified RMM described in step $i.4$. The time for communication at step i is thus seen to be

$$T_C^i = 4[i\beta + (2^i - 1)k\tau] + (2^i - 1)(\beta + k^2 2^{i-1}\tau).$$

Summing over i, gives an approximate total communication time of

$$T_C = \sum_{i=1}^{d} T_C^i \approx \beta(2d^2 + 2p + d) + (8p - 4d)k\tau + \left(\frac{2}{3}p^2 - p\right)k^2\tau.$$

In terms of matrix order N,

$$T_C \approx \beta(2d^2 + d + 2p) + \left(\frac{2}{3}N^2 + 8N - \frac{N^2}{p} - 4d\frac{N}{p}\right)\tau.$$

The total time for solving an order N symmetric tridiagonal eigenvalue problem on a d-cube using Cuppen's method can then be written

$$T_A+T_C \approx \omega\left(\kappa_0\left(\frac{N}{p}\right)^3+\frac{2}{3}\frac{N^3}{p}+2\kappa\frac{N^2}{p}+4N\right)+\beta(2d^2+d+2p)+\left(\frac{2}{3}N^2+8N-\frac{N^2}{p}-4d\frac{N}{p}\right)\tau.$$

The computation of eigenvalues and eigenvectors through equations (3.2) and (3.3) contribute the quadratic term, while updating the eigenvectors using matrix multiplication at each step leads to the cubic term.

Algorithm C2 produces an eigenvector matrix X distributed by block columns around a ring of processors embedded in the hypercube. An alternative design[1] leaves computed eigenvectors distributed according to a block row partitioning. In this row version, each processor must compute N eigenvectors of $D+\rho zz^T$ (rather than the k vectors required of each processor in step $i.3$ of Algorithm C2.) Because the matrix $\begin{pmatrix} X_0 & \\ & X_1 \end{pmatrix} J$ is distributed by rows and the matrix V is distributed by columns, the matrix product $X = \left[\begin{pmatrix} X_0 & \\ & X_1 \end{pmatrix} J\right] V$ (step $i.4$ of Algorithm C2) can be computed without communication. Each processor computes a block of k rows of X. This savings of communication at the expense of increased serial arithmetic, however, leads to a higher complexity for the block row-distributed version of Cuppen's method than for the block column approach of Algorithm C2 for nearly all problem sizes and cube dimensions. In the portion of the computation not involving eigenvectors, both versions require the same arithmetic operations and roughly equal communication.

The total time for eigenvector computation at steps $i.3$ and $i.4$ of Algorithm C2

$$T_{COL} = (2^i k^2 + 2^{2i-1}k^3)\omega + (2^i - 1)(\beta + 2^{i-1}k^2\tau).$$

The same procedure for the row version takes time

$$T_{ROW} = 2^{2i}\left(k^2+\frac{k^3}{2}\right)\omega.$$

Defining the difference $\Delta = \frac{T_{COL}-T_{ROW}}{(2^i-1)\beta}$, leads to

$$(2^i-1)\beta\Delta = (2^i - 2^{2i})l^2\omega + (2^i-1)(\beta + k^2 2^{i-1}\tau).$$

Dividing both sides by $(2^i-1)\beta$ gives

$$\Delta = 1 - k^2 2^{i-1}(\frac{2\omega}{\beta}+\frac{\tau}{\beta}).$$

When $2\omega > \tau$, as it is for the iPSC, the second term is positive. Furthermore, for most values of k on all sized cubes, $\Delta < 0$, and the column version is faster than the row version.

[1] E. de Doncker and J. Kapenga, private communication, 1986.

While deflation is not readily quantified in an analytical description of Cuppen's method, its influence is evident in the experimental results. To demonstrate the effects of deflation, two test problems are examined. The first is finding the eigensystem of the tridiagonal matrix with all off-diagonal elements equal to 1 and all diagonal elements equal to 2. Henceforth, this matrix is denoted [1,2,1]. The structure of [1,2,1] leads to significant deflation in the root finding step (i.3) of Algorithm C2: only half of the eigenvalues and eigenvectors must be computed from formulae 3.2 and 3.3. The second tridiagonal test matrix, [1,μ,1], has the value 1.0 in each off-diagonal position and the value $\mu \times 10^{-6}$ in the $\mu - th$ diagonal position. For matrix [1,μ,1], the intermediate eigenvalues (the diagonal elements of D) are distinct at each step. In addition, the small diagonal elements of [1,μ,1] (relative to its off-diagonal elements) ensure non-negligible elements of z. Thus, little or no deflation occurs at each step in the solution process. Dongarra's and Sorensen's sequential version of Cuppen's method SESUPD [12] computes the eigensystem of [1,2,1] in approximately half the time it needs for [1,μ,1] with exact times dependent on matrix order.

Speedup is defined as the time required to solve a problem using the fastest sequential method on one node divided by the time required to solve the same problem on $p \geq 2$ nodes. For Cuppen's method on the hypercube, speedup over SESUPD on one node is strongly dependent on the problem solved. SESUPD is the fastest available sequential method for finding to high accuracy all eigenvalues and orthogonal eigenvectors of a symmetric, tridiagonal matrix of order larger than 50 [12]. On one processor of the iPSC, SESUPD requires 8809.9 seconds, and EISPACK's TQL2 takes 28,341.9 seconds for matrix [1,2,1] of order 512. Figure 4 shows speedup for 32 processors as a function of matrix order for [1,2,1] and [1,μ,1]. Speedups for [1,2,1] are marked with squares and for [1,μ,1] are marked with circles. Data points connected by a solid line are measured at matrix orders equal to a multiple of 32. The points recorded at other matrix orders are joined by a dotted line. Only orders 32 through 1024 are presented graphically in order to best display the structure of the curves. Although near maximal speedup occurs in the case of little deflation, speedup of only about one half is seen when deflation is prevalent. This difference does not indicate that the eigensystem of [1,μ,1] is found in less time than that of [1,2,1] on the hypercube, but rather that Cuppen's method on the hypercube does not exhibit the large time savings from deflation shown by SESUPD. On more than one processor, SESUPD requires approximately the same time for [1,2,1] as for [1,μ,1], but on one processor, SESUPD runs considerably faster for [1,2,1] than for [1,μ,1]. Speedup for [1,μ,1] is thus greater.

The effects of deflation are reduced by other factors related to implementation for the hypercube. When the cube dimension is larger than one, the processors no longer solve identical problems at each step. However, the data exchange requirements of Algorithm C2 synchronize the processors. Thus, although a single processor may encounter significant savings when deflation occurs, the gain may not be shared by the cube as a whole. Unless the effects of deflation are evenly distributed over the processors of the cube, any time gained during root-finding by a single processor will be lost as it waits for the slower processors during the data exchange routines. The improvement due to deflation measured for sequential or for shared memory machines, on which root-finding tasks are scheduled

dynamically by a queue manager, is not expected for the statically-scheduled hypercube.

In addition, at some points in the algorithm, it is more efficient to allow all processors in the cube to perform the same computation than it is to communicate the data required for parallel computation (in step $i.2.d$ of Algorithm C2). These redundant computations cause root finding to occupy a smaller fraction of the total computation time on the hypercube than on a sequential machine. The savings in root finding are thus lessened. Introduction of the communication required for the hypercube implementation has a similar effect. As total time no longer consists entirely of computation time, the savings due to deflation are reduced.

Figure 5 shows the fraction of time spent in communication by the processors of a 5-cube while determining the eigenvalues and eigenvectors of matrices of various orders for matrix [1,2,1]. Computation time is shown to be greater than communication time for matrix orders as small as 64. For eight or more eigenvectors per processor, the communication cost levels off at about 16% of the total time.

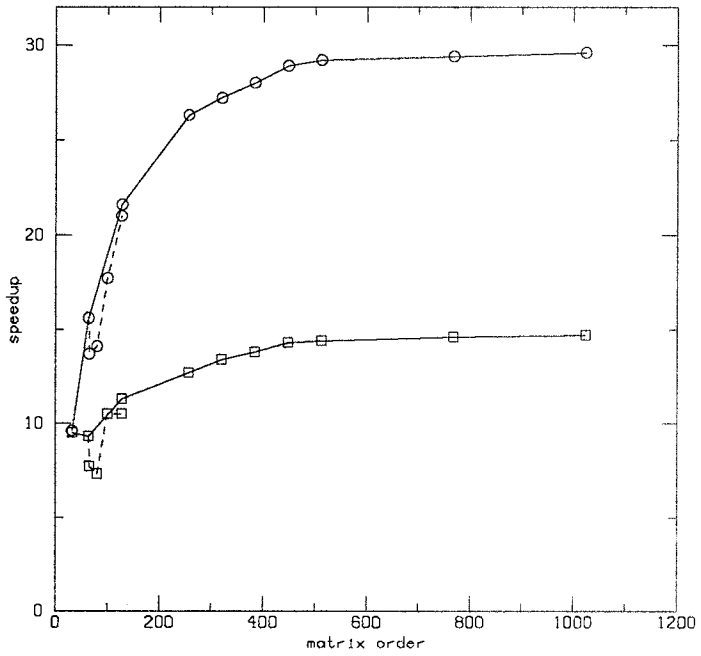

Figure 4: Cuppen's Method on a 5-cube: Speedup for [1,2,1] (squares) and [1,μ,1] (circles) *versus* Matrix Order. Points for matrix orders that are multiples of 32 are connected with solid lines. Other points are connected with dotted lines.

4. Bisection and Inverse Iteration

4.1. Sequential Algorithm

The method to be described in this section is based on the determination of eigenvalues of a symmetric, tridiagonal matrix from Sturm sequences and the subsequent computation

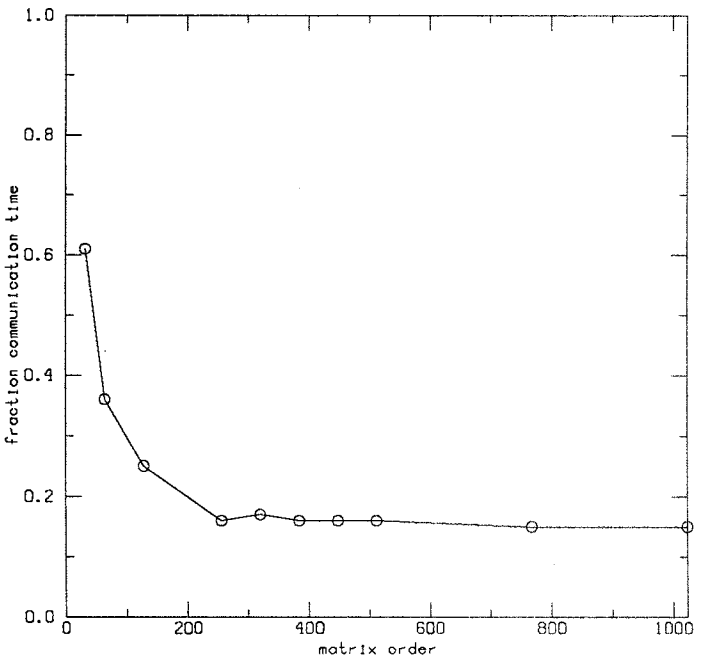

Figure 5: Cuppen's Method on a 5-cube: Fraction of Total Time Spent in Communication *versus* Matrix Order for Matrix [1,2,1].

of eigenvectors via inverse iteration. The use of this combination is generally recommended when a few of the eigenvalues and their corresponding eigenvectors are needed [32], but for purposes of comparison with Cuppen's method, the following sections concentrate on full solution of the symmetric, tridiagonal eigenproblem via bisection.

Let T be an unreduced symmetric tridiagonal matrix of order N with diagonal elements $d_0, d_1, \cdots, d_{N-1}$ and off-diagonal elements $f_1, f_2, \cdots, f_{N-1}$. According to the Gerschgorin Disk Theorem, the ith eigenvalue of T is confined to a disk on the real line centered at d_i with radius $|f_i| + |f_{i+1}|$. All eigenvalues of T are then known to lie within the union of its N Gerschgorin disks. Individual eigenvalues are located in this interval by solving the characteristic equation $\det(T - \lambda I) = 0$.

The sequence of principal minors of the matrix $T - \lambda I$ is given by the linear recurrence

$$\begin{aligned} p_0(\lambda) &= 1 \\ p_1(\lambda) &= d_0 - \lambda \\ p_i(\lambda) &= (d_i - \lambda)p_{i-1} - f_i^2 p_{i-2}(\lambda), \ i = 1, \cdots, N-1. \end{aligned} \tag{4.1}$$

As noted by Givens [17], the number of eigenvalues of T less than λ is equal to the number of sign changes in the sequence $\{p_i(\lambda), i = 1, 2, \cdots, N\}$. During numerical computation, however, the linear recurrence in equation (4.1) is prone to overflow and underflow. To avoid these difficulties, $\{p_i(\lambda)\}$ is replaced by the nonlinear recurrence $\{q_i(\lambda)\}$, where

$$q_i(\lambda) = \frac{p_i(\lambda)}{p_{i-1}(\lambda)}, \ i = 1, \cdots, N. \tag{4.2}$$

The number of eigenvalues less than λ is equal to the number of negative terms in $\{q_i(\lambda)\}$ [3]. Denoting this number by $c(\lambda)$, the number of eigenvalues in the interval $[a, b)$ is given by the difference $c(b) - c(a)$.

This Sturm sequence property is used to refine the eigenvalues of T through repeated bisection of the initial Gerschgorin intervals and determination of the numbers of eigenvalues lying in the resulting subintervals. Any interval half found to contain no eigenvalues is discarded from the search area. Occupied intervals are further bisected until single eigenvalues have been extracted to a given tolerance or until intervals containing more than one eigenvalue have been reduced to a width smaller than that tolerance. In the latter case, the confined intervals are considered to represent a *cluster* of computationally coincident eigenvalues.

When a single eigenvalue has been isolated within an interval, its determination can be accelerated by using an interpolation scheme [6] or a root finder such as Zeroin [14]. These methods, although faster than bisection, require use of equation (4.1) and are not included in this discussion.

Once the eigenvalues have been computed using bisection, their corresponding eigenvectors are found with inverse iteration. When the eigenvalues are well-separated, inverse iteration converges quickly and generates orthogonal eigenvectors [33]. These properties make inverse iteration the method of choice when no clusters are present. When close eigenvalues do occur, the rate of convergence of inverse iteration decreases and the resultant eigenvectors, although independent, are not necessarily orthogonal [33]. An additional orthogonalization step must be included. Furthermore, if the close eigenvalues are actually computationally coincident, standard inverse iteration, which converges to a single eigenvector, cannot be employed to find the eigenspace corresponding to the multiple eigenvalue [33]. These considerations suggest that an alternate approach is needed when eigenvalues are poorly separated.

One remedy to the difficulty is to perturb close eigenvalues to a nonproblematic distance. Wilkinson [33] maintains that a separation between eigenvalues on the order of three times machine precision is sufficient to produce independent eigenvectors. These vectors do suffer the loss of orthogonality associated with close eigenvalues, so the modified Gram-Schmidt procedure is recommended for orthogonalizing the vectors. This perturbed inverse iteration followed by orthogonalization has gained acceptance as implemented in the routine TINVIT of EISPACK [32].

The procedure for finding all eigenvalues of a symmetric, tridiagonal matrix using bisection and the corresponding eigenvectors using inverse iteration is summarized in Algorithm B1.

Algorithm B1: Steps in the Bisection-Inverse Iteration Procedure

Step 0: *Determination of initial search area.* Find intervals containing all eigenvalues (*e.g.*, from Gerschgorin disks.)

Step 1: *Computation of eigenvalues.* Use bisection to determine all eigenvalues.

Step 2: *Computation of eigenvectors.* Compute the eigenvectors by inverse iteration. Treat eigenvectors corresponding to clustered eigenvalues by appropriately perturbing the eigenvalues in the cluster.

Step 3: *Orthogonalization of eigenvectors.* Employ modified Gram-Schmidt to orthogonalize eigenvectors corresponding to close eigenvalues.

In order to provide a clear extension of previous studies of bisection and multisection [23, 32], the easily parallelized method of inverse iteration is employed for the computation of eigenvectors in Algorithm B1. Nevertheless, other methods for computation of the eigenvectors could be considered. One method that can be useful for the computation of eigenvectors of clustered eigenvalues is that of subspace iteration [19, 24]. This generalization of the power method iterates on a subspace of dimension $m > 1$ rather than on a single vector and, for the iteration matrix $(T - \lambda I)^{-1}$, converges to the m eigenvectors corresponding to the m eigenvalues of T nearest to λ. In typical implementations, orthogonalization steps are interspersed with the iterations to ensure computation of orthogonal vectors [4, 24]. Subspace iteration thereby provides the required orthogonal eigenspace corresponding to pathologically close eigenvalues.

In practice, some variations are needed to make subspace iteration a useful tool. For example, the rate of convergence of simple subspace iteration is is dependent on the distance from the cluster to the nearest eigenvalue of T [19]. When this distance is small, a slow rate of convergence is expected. In this instance, use of a convergence acceleration scheme is recommended [19, 24, 27, 28]. The rate of convergence of simple or accelerated subspace iteration is also strongly dependent on the initial subspace. In general, the dimension of the initial and intermediate subspaces must be significantly larger than that of the desired eigenspace [4]. We intend to examine the potential of methods other than inverse iteration for parallel implementation in future work.

4.2. Parallel Algorithm

The hypercube version of bisection involves a straightforward partitioning of the computing tasks outlined in Algorithm B1. For a matrix of order $N = kp$, each processor in a hypercube of $p = 2^d$ processors computes k eigenvalues and eigenvectors. When N is not a multiple of the number of processors, work is assigned so that no processor computes more than one eigenvalue and eigenvector more than any other. Eigenvalue computation is accomplished by use of the EISPACK routine TRIDIB. This implementation of bisection allows computation of any number of consecutive eigenvalues specified by their indices [32]. Upon input of the diagonal and off-diagonal elements of the symmetric, tridiagonal matrix as well as the index of the first eigenvalue to be found and the total number of eigenvalues to be determined, all Gerschgorin disks are computed. (On a hypercube where both message startup and elemental transfer times are small, each processor should compute a share of disks and communicate them to all other processors.) An initial bisection procedure is carried out to determine that portion of the Gerschgorin interval containing the desired set of eigenvalues. From this reduced interval, the individual eigenvalues are again extracted via bisection. On the hypercube, processor i uses TRIDIB to compute eigenvalues ik through $(i + 1)k - 1$.

With all eigenvalues extracted, it remains only to determine orthogonal eigenvectors. When the eigenvalues are well-separated, each eigenvector is computed in a small number of inverse iterations. An effective load balance is then achieved by having each processor compute $\frac{N}{p}$ of the vectors. No communication is required, and an equal number of

eigenvectors is stored in each processor. If N is not a multiple of p, remaining vectors are computed and stored, one to a processor as necessary.

When close eigenvalues occur, the rate of convergence of inverse iteration decreases, and the computed eigenvectors must be orthogonalized. Although the time for the second task is easily quantified, the additional iteration time due to slower convergence cannot be predicted. At best, an approximate time weighting of eigenvector computing tasks can be based on cluster size. In addition, any scheme geared toward even distribution of computation time must also be examined from the point of view of memory allocation: solution of large problems demands even distribution of memory resources. Three approaches to the resulting load balancing problem are considered below. The first and third are static schemes in which assignment of eigenvectors is based solely on a processor's location in the ring. The second provides *a priori* distribution of work according to approximate time requirements for computations. In all three schemes, the processors of a d-cube are numbered in a ring according to a Gray code ordering (as in Section 2.3.)

1. *Block Distribution.* In this case, processor j is assigned the computation of eigenvectors indexed $jk, jk+1, \ldots, (j+1)k-1$, $0 \leq j \leq 2^d-1$, when $N = k2^d = kp$. Each processor thus requires the same amount of memory for eigenvector storage.

   Although the storage needs are well-balanced, the time required to complete eigenvector computation is strongly dependent on eigenvalue distribution. If clusters of eigenvalues are unevenly distributed across the spectrum, one processor can encounter a heavy workload of computing and orthogonalizing the eigenvectors of many clusters while another has the faster job of determining only eigenvectors corresponding to well-separated eigenvalues. For example, although the smallest few eigenvalues of the Wilkinson matrix W_N^+ are well-spaced, for large enough order N, the largest ones occur in computationally coincident pairs. Under the block distribution scheme, processors assigned blocks of large eigenvalues will perform more operations for inverse iteration on clusters and orthogonalization than those given smaller eigenvalues.

   The eigenvalue distribution effects not only the load balance but also the communication requirements. If clusters are small and are confined to a single processor, no communication is needed. Any cluster of the form $\lambda_{lk-m} \approx \ldots \approx \lambda_{lk+n}$ for positive integers l, m, and n must be split among at least two processors. After computation of the eigenvectors, the processors must communicate to complete the modified Gram-Schmidt process. The efficient pipelining of computations among many processors used in Algorithm MGS is not possible when processors hold sets of adjacent eigenvectors. The block scheme provides a regular and even distribution of eigenvectors, but, in the worst case, can result in severe imbalance of communication and computation. Furthermore, extra communication requirements surface as special cases.

2. *Weighted Task Scheduling.* In this approach, each computational task is assigned a weight or *time value* based on its expected completion time. The weighted tasks are apportioned between processors according to a scheduling rule designed to give a fast completion time [25]. As noted earlier, cluster size can be used as a weighting for approximate load balancing.

   A simple scheduling heuristic has some significant drawbacks when employed for eigenvector computation. First, unlike an implicit allotment of eigenvectors (such as the

block distribution), creation and manipulation of the task queue introduces computational overhead. Second, assignment of tasks to processors is based solely on time values. Because the time required to compute the eigenvectors of a cluster of a given size is greater than that needed for the same number of well-separated eigenvalues, a processor given a cluster by the schedule can be required to compute many fewer vectors than one assigned only the eigenvectors corresponding to single eigenvalues. While such assignment gives a balance of time requirements, it can lead to a poor distribution of memory use if one processor is required by the schedule to compute many more eigenvectors than another.

Splitting of large clusters into smaller pieces can solve the memory allocation problem, but this action complicates the implementation by forcing special case task assignments as well as introducing the need for communication between processors for eigenvector orthogonalization. A final difficulty with weighted scheduling is that the computed eigenvectors are arranged arbitrarily across processors according to the schedule. The regularity of the block (or the following cyclic) distribution scheme is lost.

3. *Cyclic Distribution.* Like the block distribution scheme, this third load balancing approach employs a uniform eigenvector assignment: processor j in the ring of $p = 2^d$ processors computes eigenvectors indexed $j, j+p, j+2p, \ldots, j+\nu p \leq N$.

   The most appealling feature of cyclic distribution is its regularity. The eigenvectors are systematically distributed across processors and can be located without regard to computation time requirements. By the same token, memory allocation is maximally efficient as no processor computes more than one eigenvector more than any other processor. Cyclic distribution is expected to provide a good balance of processor work load. Unless a cluster is very large (including more than p eigenvalues), no processor is required to handle more than one vector from any given cluster. Furthermore, except in the event of small, cyclically distributed clusters, the burden of cluster handling is spread across processors.

   The biggest drawback to cyclic distribution is the extent of communication required. Yet, while communication for orthogonalization must take place whenever a cluster occurs, the frequency of messages initiated by one processor is independent of the number of eigenvalues in the cluster. Furthermore, an effective computational pipeline is possible (see Algorithm MGS.) Communication for clusters in different processors takes place simultaneously. With overlap of communication and computation, idle time is minimized.

For the reasons of regularity, simplicity, and a good expected load balance, cyclic distribution is employed during computation and orthogonalization of eigenvectors. Because the eigenvalues are computed according to a block distribution, the eigenvalue computation is followed by a redistribution of eigenvalues (by Algorithm ADE) so that each processor has access to all computed eigenvalues.

Each processor examines the complete list of eigenvalues and perturbs those too close for inverse iteration. Processor i employs the EISPACK routine TINVIT [32] to compute eigenvectors corresponding to eigenvalues indexed $i, i+p, \ldots, i+\nu p \leq N$ by inverse iteration. If clustered eigenvalues occur, their perturbed values are employed during the eigenvector computation. The unperturbed computed eigenvalues and the orthogonal

eigenvectors comprise the solution to the eigenproblem.

The exchange of eigenvalues permits a transition from the block distribution of eigenvalues to the cyclic distribution needed for orthogonalization without communication of vectors. Computation of a cyclic selection of eigenvalues would obviate the need for exchange but would prevent the savings of Sturm sequence evaluations recognized in computing adjacent eigenvalues. When adjacent eigenvalues are computed, the Gerschgorin interval is narrowed initially to one containing only the $\frac{N}{p}$ eigenvalues. This interval is reduced further with the computation of each eigenvalue. This continuous reduction of the initial search area is not possible when computing cyclically distributed eigenvalues.

The steps for bisection on the hypercube are given as Algorithm B2.

Algorithm B2: Solution of Eigenproblem of Order $N = kp$ on a p-Processor Hypercube.

Each processor has all diagonal and off-diagonal elements of the matrix. In parallel, all p processors perform the following steps.

Step 0: *Determination of initial search area.* All processors compute all Gerschgorin disks to find the interval containing all N eigenvalues and use bisection to narrow the interval to that containing $\frac{N}{p}$ specified eigenvalues. Processor i finds the interval containing eigenvalues indexed ik through $(i+1)k-1$.

Step 1: *Computation of eigenvalues.* Processor i uses bisection to determine the $\frac{N}{p}$ eigenvalues ik through $(i+1)k-1$.

Step 2: *Communication of eigenvalues.* All processors exchange computed eigenvalues using algorithm ADE.

Step 3: *Perturbation of eigenvalues.* Each processor sorts and perturbs any of the N eigenvalues spaced too closely for inverse iteration.

Step 4: *Computation of eigenvectors.* Processor i computes the $\frac{N}{p}$ eigenvectors corresponding to eigenvalues indexed $i, i+p, \ldots, \nu p \leq N$.

Step 5: *Orthogonalization of eigenvectors.* All processors employ algorithm MGS to orthogonalize eigenvectors corresponding to close eigenvalues.

4.3. Analytical and Experimental Results

As noted in the preceding sections, both bisection and inverse iteration are readily implemented on local-memory multiprocessors. The efficiency of this approach is reflected in the plots given in Figure 6 of speedup *versus* matrix order for matrix [1,2,1] and for random matrices. The speedup is calculated as the time for EISPACK's BISECT and TINVIT combination run on a single processor divided by the greatest node time for the hypercube bisection and inverse iteration procedures executed on a 32-node iPSC. (BISECT with TINVIT is the fastest method for finding all eigenvalues and eigenvectors of a symmetric, tridiagonal matrix of any order on one processor of the iPSC. For order 512 and matrix [1,2,1], BISECT and TINVIT take a total of 2340.0 seconds.) Circles mark speedup values for matrix [1,2,1], while each square corresponds to the speedup for a random matrix. Data points for matrix [1,2,1] measured at matrix orders equal to multiples of 32 are connected with a solid line; points at other orders are joined with a dashed line. Different random matrices were generated for each order, so no relation is

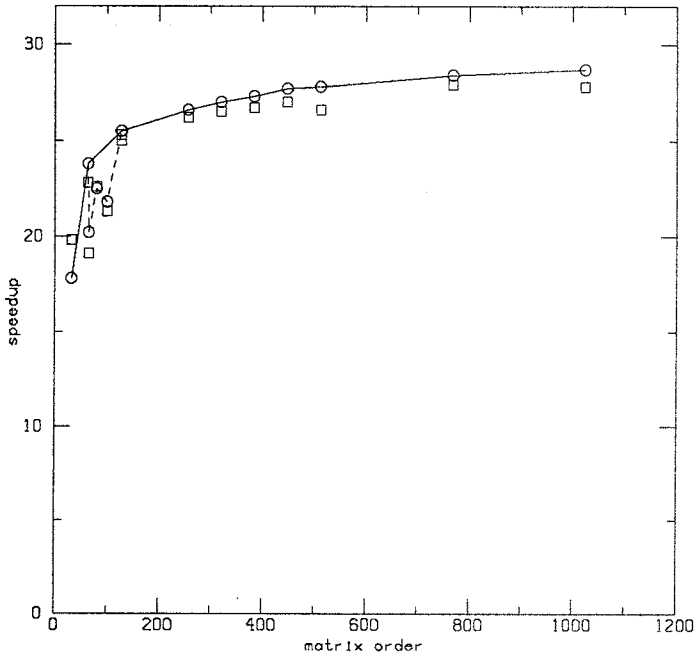

Figure 6: Bisection on an iPSC/d5M: Speedup for [1,2,1] (circles) and random matrices (squares) *versus* Matrix Order.

expected between random matrix data points.

The speedup for [1,2,1] is seen to increase smoothly with matrix orders proportional to the number of processors. Efficiencies ranging from 77% to 89% are achieved for matrix orders above 100. Only matrix orders 32-1024 are presented in Figure 6 in order to preserve readability. The largest problem with order equal to a multiple of 32 that can be solved under the current operating system on the iPSC has order 3520. Comparable speedups found for random matrices of all orders show that the results are not strongly dependent on properties particular to matrix [1,2,1]. Efficiencies for other matrix orders fall to as much as 12% below the smooth line of those recorded for multiples of 32. This reduction of speedup results from the fact that some processors are required to compute one more eigenvalue and eigenvector than are others. The alternate direction exchange of results following the computation of eigenvalues (in step 2 of Algorithm B2) serves to synchronize the processors. Those processors with a lesser workload are idle until the processors with a greater workload enter the exchange. The orthogonalization of eigenvectors can be similarly delayed by the uneven distribution of inverse iteration tasks. Hence, the time to complete the parallel computation is determined by the processor with the largest assignment of work.

The reduction of speedup is well-illustrated with an example. For matrix order 65, processor 0 computes three eigenvalues while all remaining processors compute two. Because the processors are delayed until completion of the largest set of tasks, the order 65 problem takes as long as if all processors were required to compute three eigenvalues, *i.e.*,

as if the matrix order were 96. The sequential solution of an order 65 problem, however, takes little more time than the solution of the order 64 problem but considerably less than that of the order 96 problem. Thus, the computed speedup for order 65 is appreciably less than that for order 64. In contrast, the order 127 problem, for which both sequential and parallel times are close to those required for order 128, has a computed speedup approximately equal to that for order 128.

Despite the extent of the parallelism inherent in the bisection and inverse iteration procedures, maximal speedup is not achieved. This loss of speedup results from time spent in non-arithmetic tasks as well as from some non-parallel computation discussed below. The contribution of this overhead is seen to be comparable in magnitude to the reduction in speedup. Because the amount of communication in Algorithm B2 is problem dependent (due to Algorithm MGS), the time spent in communication cannot be determined by executing the algorithm without the arithmetic steps. Measurement of the time spent in communication by a single processor thus includes time spent waiting for messages. Figure 7 shows the average fraction of the total time spent idle or in communication by one processor. Again, the points for matrix [1,2,1] measured at orders divisible by 32 define a smooth curve falling from 20% of the total time at matrix order 32 to about 2% of the total for matrix orders larger than 320. An increase in non-arithmetic activity, due to the load imbalance discussed above, is noted for most orders not equal to multiples of 32. As expected, this increase is significant for order 65 but barely discernible for order 127.

The eigenvalue computation begins with each processor independently computing all Gerschgorin disks then carrying out an initial bisection to determine the interval containing its share of the eigenvalues. While this process could be altered to result in greater parallelism, its present contribution to the total time is small. For a variety of matrices including [1,2,1], the initial bisection time decreases smoothly for all matrix orders from a maximum of about 15% of the eigenvalue computation time at order 32 to about 3% at order 512 and 1% at order 1024. The grouping of close eigenvalues done in parallel by all processors occupies less than 5% of computation time for all matrix orders. The time needed for starting the pipeline of communication for algorithm MGS represents an additional loss of efficiency amounting to less than 2% of the total time for all orders of matrix [1,2,1]. Thus, at order 32, about 42% of the total time is spent in communication and non-parallel arithmetic. This slowing of the parallel execution time completely accounts for the the observed efficiency of 58%. Similarly, the 10% of the total time spent in overhead for order 1024 approximates the observed lowering of the speedup curve below optimal.

Figure 8 shows the average fraction of the total time spent in computing, exchanging, and grouping eigenvalues, in determining eigenvectors, and in orthogonalizing eigenvectors corresponding to closely spaced eigenvalues for several orders of matrix [1,2,1]. Finding and distributing the eigenvalues to all processors occupies more than 80% of the total time, while computing the eigenvectors occupies most of the remaining time. The low contribution of the orthogonalization step confirms the effectiveness of the cyclic distribution approach to and the use of a large number of processors for the modified Gram-Schmidt procedure. Modified Gram-Schmidt occupies 2.6% of the total time for order 500 and 10.2% of the total for order 1000 for a similar algorithm executed on an Alliant FX/8 [23].

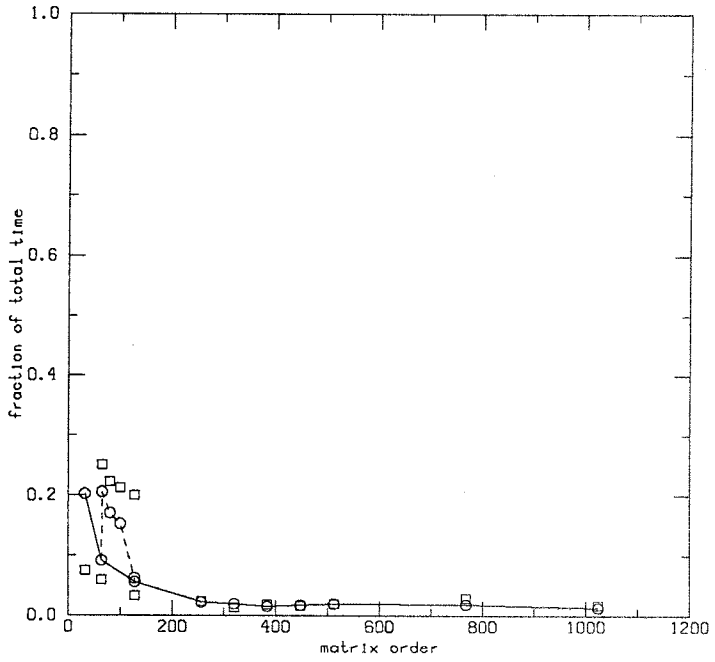

Figure 7: Bisection on an iPSC/d5M: Communication Overhead as a Fraction of the Total Time *versus* Matrix Order.

As was true for the speedup and communication curves, data points for matrix orders that are multiples of 32 lie on a smooth curve, and points for other orders do not. The increase in eigenvalue computation time for the deviant points corresponds to the increase in non-arithmetic time recorded in Figure 7. The increase is offset by equivalent decreases in fractions of eigenvector computation time and orthogonalization.

While the exact time distribution among computing tasks is problem dependent, analytic examination of a simple model problem can shed light on the expected arithmetic requirements of the bisection method. Consider the symmetric, tridiagonal matrix T_{model} of order $N = kp$ having eigenvalues $0, \frac{\alpha}{N}, \ldots, \frac{(N-1)\alpha}{N}$. Suppose that its N Gerschgorin disks overlap to form a continuous interval from 0 to α. In this way, the spectrum of T_{model} approximates that of [1,2,1] which has N eigenvalues initially confined within an interval of length 4 for all values of N.

During the initial bisection of Algorithm B2, each processor finds an interval of length $\frac{\alpha}{p}$ containing k eigenvalues. This step takes takes $l \geq \log_2 p$ iterations for a total of $l+1$ Sturm sequence evaluations. In subsequent steps, each processor extracts its k eigenvalues. Finding its largest eigenvalue first, each processor reduces an interval of width $\frac{\alpha}{p}$ to one of width δ in $l_1 \geq \log_2 \frac{\alpha}{p\delta}$ bisections. After this computation, the portion of the interval following the first eigenvalue computed is discarded, leaving a new search interval of length $\frac{\alpha}{p} - \frac{\alpha}{N}$. The second eigenvalue is then extracted in $l_2 \geq \log_2 \frac{\frac{\alpha}{p} - \frac{\alpha}{N}}{\delta}$ bisections. In general, the ith eigenvalue is found in an interval of width $\frac{\alpha}{p} - (i-1)\frac{\alpha}{N}$ in $l_i \geq \log_2 \frac{\frac{\alpha}{p} - (i-1)\frac{\alpha}{N}}{\delta}$

bisections, a process requiring $l_i + 1$ Sturm sequence evaluations. If the time to complete one Sturm sequence evaluation is $2N\omega$, the total time for the eigenvalue computation is

$$T_B \geq \left[(l+1) + \sum_{i=1}^{k} \left(log_2 \frac{\alpha i}{N\delta} + 1 \right) \right] 2N\omega$$

$$\approx \left[\log_2 p + 1 + k \log_2 \frac{\alpha k}{N\delta} + k \right] 2N\omega.$$

The eigenvectors are computed using inverse iteration. For a tridiagonal matrix, each iteration requires time of about $5N\omega$. Assuming that convergence is achieved in an average of two iterations, computing k eigenvectors takes time $T_I = 10Nk\omega$. The spacing of the eigenvalues is assumed wide enough that additional orthogonalization of eigenvectors is not required.

The attainable tolerance δ is related to both machine precision and the largest eigenvalue magnitude [32]. For the model problem in double precision, $\delta \approx 10^{-15}\alpha \approx 2^{-50}\alpha$. Thus, for the model problem with orders 32 through 1024 on a 5-cube, $\frac{T_B}{T_I}$ decreases from 12 to about 11. This ratio of pure arithmetic times indicates that the eigenvalue computation dominates the total arithmetic time for the model problem. The eigenvalues of matrix [1,2,1] are more closely spaced than those of T_{model} and so require fewer bisections to extract. The eigenvalue computation for [1,2,1] occupies about four times the time of its eigenvector computation. (As indicated by Figure 7, non-arithmetic operations contribute minimally to the experimental result.) In addition, the predicted reduction in eigenvalue computation time with respect to eigenvector time for increasing matrix order of T_{model} is also reflected in Figure 8. For [1,2,1], $\frac{T_B}{T_I}$ falls from 7.1 at order 32 to 4.3 at order 1024.

The model problem further follows the experimental results by having a communication complexity significantly smaller than its arithmetic complexity and decreasing with matrix order. A single alternate direction exchange of $\frac{N}{p}$ eigenvalues per processor comprises the total communication requirement of Algorithm B2. Under the assumption that $\beta \approx 10\omega$ and $\tau \approx \frac{\omega}{125}$, the communication time is given by

$$T_C = 2 \left(\log_2 p\beta + (p-1)\frac{N}{p}\tau \right)$$

$$\approx 2 \left(10 \log_2 p + \frac{p-1}{125p} N \right) \omega.$$

For the model problem on a 5-cube, the ratio of communication time to computation time $\frac{T_C}{T_B}$ falls from .02 at order 32 to 3×10^{-5} at 1024. That the fraction of communication time shown for [1,2,1] in Figure 7 is greater than the predicted value for T_{model} is again due to the larger predicted time for eigenvalue computation for the latter. As shown in Figure 8, orthogonalization does not greatly alter the run time for matrix [1,2,1].

5. Multisection and Inverse Iteration

5.1. Sequential Algorithm

Bisection involves repeated halving of a search interval. A generalization of bisection known as *multisection* is the division of an interval into $p \geq 2$ equal sized pieces. Through

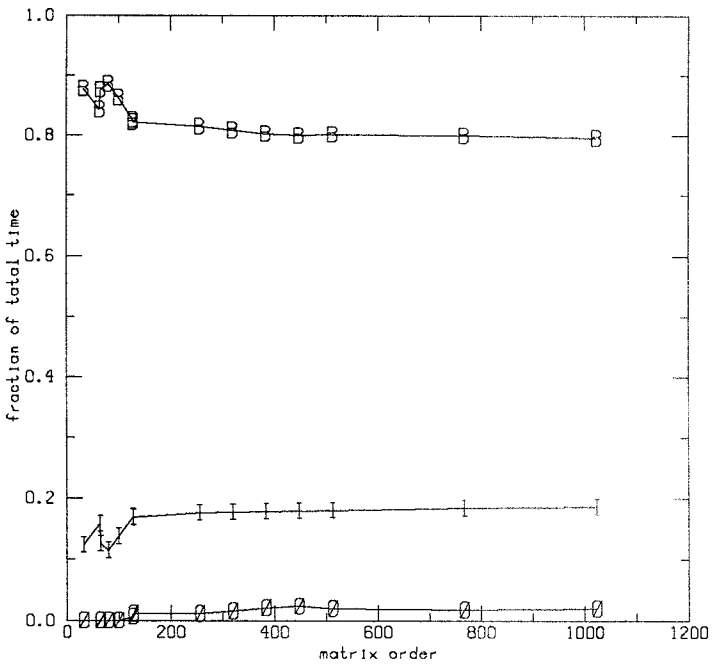

Figure 8: Bisection on a 5-cube: Fraction of Total Time Spent in Eigenvalue Computation (B), Eigenvector Computation (I), and Orthogonalization (O) *versus* Matrix Order.

Sturm sequence evaluation at the endpoints of these sections, p eigenvalue counts are obtained. Sections found to contain more than one eigenvalue are themselves multisected in order to locate individual eigenvalues or clusters of eigenvalues within disjoint sections. As in bisection, subintervals found to hold no eigenvalues are discarded.

In practice, a pairing of multisection and bisection provides an effective mechanism for eigenvalue determination [23]. Given an initial search interval, multisection is used to separate that interval into sections containing either one eigenvalue or a cluster of eigenvalues. When a section is found to contain only one eigenvalue, multisectioning is discontinued regardless of interval width. In the terminology of [23], an eigenvalue confined alone in this way has been *isolated*. Single eigenvalues determined to within a given tolerance δ have been *extracted*. Eigenvalues grouped within an interval of width less than δ are not isolated individually but rather are considered to form an extracted cluster.

As shown in [23], single eigenvalues isolated by multisection should be further refined not by multisection but rather by bisection. Because the interval is known to contain only one eigenvalue, it is certain that more "empty space" can be discarded per Sturm sequence evaluation through bisection than through multisection. For an isolated eigenvalue, bisection is optimal on a single processor.

Once all eigenvalues have been extracted, the computation of eigenvectors proceeds as for bisection (see Algorithm B1). Clustered eigenvalues are perturbed to an appropriate separation, and eigenvectors are computed for this altered set of eigenvalues. Eigenvectors

corresponding to close eigenvalues are orthogonalized through the modified Gram-Schmidt procedure. The unperturbed eigenvalues and orthogonalized eigenvectors are returned.

The procedure for finding all eigenvalues of a symmetric, tridiagonal matrix using multisection and the corresponding eigenvectors with inverse iteration is summarized in Algorithm M1.

Algorithm M1: Steps in the Multisection-Inverse Iteration Procedure.
Step 0: *Determination of initial search area.* Compute all Gerschgorin disks to provide an initial search area for all eigenvalues.
Step 1: *Isolation of eigenvalues.* Use multisection recursively to divide intervals until each resulting subinterval holds either one eigenvalue or an extracted cluster of eigenvalues.
Step 2: *Extraction of eigenvalues.* Recursively use bisection to divide intervals containing one eigenvalue until the eigenvalue has been approximated to a specified tolerance.
Step 3: *Computation of eigenvectors.* Perturb close eigenvalues as necessary for inverse iteration. Compute all eigenvectors.
Step 4: *Orthogonalization of eigenvectors.* Orthogonalize eigenvectors corresponding to close eigenvalues.

5.2. Parallel Algorithm

In Algorithm B2, bisection of a search interval is carried out by a single processor. Parallel multisection provides a means of using all processors to find the eigenvalues in one interval. Algorithm M1 constitutes the basis for a parallel multisection algorithm. Multisection for isolation of eigenvalues entails cooperative use of all processors. Because each processor only has access to its own memory, communication of intermediate results is necessary. Bisection, multisection for extraction, and inverse iteration operate independently on single processors. Orthogonalization of eigenvectors is performed by all processors holding eigenvectors corresponding to close eigenvalues. Details of the algorithm are largely concerned with the best assignment of eigensolving tasks to the processors. Eigenvector determination proceeds as in Algorithm B2. The approaches to load balancing taken during eigenvalue computation are as follows.

1. *Determination of Initial Search Area.*

   Given the diagonal and off-diagonal elements of the matrix, each processor determines the initial search area by computing all Gerschgorin disks. As noted in Section 4.2, parallelism of this step can be increased when communication costs are low.

2. *Isolation of Eigenvalues.*

   For a cube with $p = 2^d$ processors, an interval (conglomerate of Gerschgorin disks) known to contain more than one eigenvalue is divided into p equal sized subintervals. Processor i determines the number of eigenvalues located in the ith subinterval. All processors then employ Algorithm ADE to exchange their eigenvalue counts. Using the collected numbers of eigenvalues in the p intervals, each processor then discards empty intervals and retains for extraction intervals containing a single, isolated eigenvalue. Each processor places in a queue all intervals requiring further parallel multisectioning. Parallel multisection must be discontinued when the resulting sections cannot be evenly divided among the processors. In this stopping criterion, the parallel implementation

differs from Algorithm M1. In that case, clusters were extracted during the isolation phase. On the hypercube, a cluster of eigenvalues is considered to be isolated when it has been confined to an interval of width less than $p\delta$. The width of such an interval is not necessarily less than δ and is dependent on the cube dimension. The isolated cluster has not yet been extracted, and the endpoints of its surrounding interval are listed for extraction. Thus, every processor creates and maintains an identical queue of remaining multisectioning tasks and a list of extraction tasks.

In lock step, the processors perform multisection on the interval currently at the head of the queue. Processor i again checks section i. Multisectioning continues until all single eigenvalues are isolated (and the multisection queue is empty) or until the resulting sections are too small to evenly partition among p processors. A balanced load is achieved through equal computation and communication. The exchange of eigenvalue counts at each iteration ensures that all space found to be empty is discarded.

3. *Extraction of Eigenvalues.*

The sets of interval endpoints generated during the isolation phase define two distinct sets of extraction tasks. Sequential bisection is used to extract isolated single eigenvalues from intervals of arbitrary length. One sequential multisection into sections of width δ is used to extract each cluster from its interval of length less than $p\delta$. The simple synchronization scheme of isolation is no longer applicable.

The shared memory approach to load balancing involves use of a queue of tasks held in common memory: an idle processor removes from the queue and performs the first available task. All processors are kept busy with tasks selected from the queue until all tasks have been completed. By assigning one or more processors to queue management tasks, the queue-based dynamic task scheduling procedure could be carried out on the distributed-memory hypercube. Removing the managing processors from computing tasks and introducing the possibility of communication bottlenecks during task assignments, however, reduces the total possible efficiency of the cube. For this reason, a static task assignment is chosen.

The problem of scheduling such a set of nonidentical tasks to the processors so as to minimize completion time is NP-complete [31]. In an examination of standard weighted scheduling algorithms, the Longest Job First (LJF) schedule is seen to provide a finish time for γ jobs on p processors no greater than than $\frac{4}{3} - \frac{1}{3p}$ times the optimal in $O(\gamma \log \gamma)$ operations [20]. This heuristic requires that the tasks be sorted according to decreasing time value and then assigned, in turn, to the processors. The longest available job is given to the processor having the lightest total workload.

As each processor in the hypercube has a copy of all interval endpoints, each makes estimates of the time required to extract the eigenvalues. Extraction of an isolated eigenvalue in an interval of width I by bisection requires $\lceil log_2 \frac{I}{\delta} \rceil$ iterations for a total of $\lceil log_2 \frac{I}{\delta} \rceil + 1$ Sturm sequence evaluations. Multisection of the same interval (when $I < p\delta$) can be completed in a single iteration with subdivision into $\frac{I}{\delta} < p$ sections. No more than $\frac{I}{\delta} + 1$ sequence evaluations are performed. While δ, as defined in Section 5.1 is actually dependent on the values of the interval endpoints, rough estimates of the times required to extract all eigenvalues from different intervals can be based solely

on the intervals' widths. Every extraction task is therefore assigned a time value of $\lceil log_2 \frac{I}{\delta} \rceil + 1$ if it is a bisection and $\frac{I}{\delta} + 1$ if it is a multisection. The LJF heuristic is efficiently implemented using a heap to record task assignments for all processors. Use of this data structure dictates a sequential implementation; therefore, the sorting and scheduling procedure is performed by every processor. After all eigenvalues and clusters of eigenvalues have been extracted, an alternate direction exchange ensures that all processors have all eigenvalues.

Recall that static allocation of tasks and synchronization of processors prevents the predicted gains from deflation for Cuppen's method. Unless each processor receives a selection of tasks requiring an equal amount of time, some processors will remain idle while the more heavily loaded processors complete their tasks and begin communication. In this way, the static task scheduling followed by collection of data can reduce speedup of the extraction process. For example, the weight assigned for multisection tasks represents the worst case time for extracting eigenvalues. In fact, the eigenvalues may be distributed within the input interval so that all can be extracted within the first few sections. The remaining Sturm sequence evaluations need not be completed. Only if all processors experience such savings, however, is an overall decrease in runtime recorded. Accelerated bisection methods [5, 13] are similarly foiled. A uniform distribution of work among processors results only when estimated time values are accurate.

Once the eigenvalues have been computed, they are exchanged among all processors. The eigenvectors are the computed and orthogonalized following the steps of Algorithm B2. Hence, multisection on the hypercube is performed according to the procedure summarized as Algorithm M2.

Algorithm M2: Solution of Eigenproblem of Order $N = kp$ on a p-Processor Hypercube.

Each processor has all diagonal and off-diagonal elements of the matrix. In parallel, all p processors perform the following steps.

Step 0: *Determination of initial search area.* All processors compute all Gerschgorin disks to provide an initial search area for all eigenvalues. All isolated disks are added to the extraction list. The extreme boundaries of overlapping disks are queued for isolation.

Step 1: *Isolation of eigenvalues.* Each processor divides the first interval in the isolation queue into p equal sized sections. Processor i determines the number of eigenvalues in section i. All p processors use Algorithm ADE to exchange their eigenvalue counts. Each processor discards any of the p sections found to contain no eigenvalues, adds sections containing one eigenvalue to the extraction list, and appends all sections containing more than one eigenvalue to the end of the isolation queue. An interval containing more than one eigenvalue in a width less than $p\delta$ is added to the extraction list. Step 1 is repeated until the isolation queue is empty.

Step 2: *Extraction of eigenvalues.* Each processor determines its share of eigenvalues from the extraction list according to a LJF scheduling heuristic. Each processor uses bisection to extract single eigenvalues and multisection to extract clusters in its share of intervals.

Step 3: *Exchange of eigenvalues.* All processors exchange all computed eigenvalues using Algorithm ADE.

Step 4: *Perturbation of eigenvalues.* Each processor perturbs close eigenvalues as necessary for inverse iteration.
Step 5: *Computation of eigenvectors.* Processor i computes eigenvectors $i, i+p, \ldots, i+\nu p \leq N$ with inverse iteration.
Step 6: *Orthogonalization of eigenvectors.* Each processor orthogonalizes those of its eigenvectors corresponding to close eigenvalues. The processors employ Algorithm MGS.

5.3. Analytical and Experimental Results

As for bisection, the speedup for multisection is calculated as the time for EISPACK's BISECT and TINVIT combination run on a single processor divided by the longest node time for the hypercube multisection and inverse iteration procedures executed on an iPSC/d5M. The results are given in Figure 9. Data points for matrix [1,2,1] are marked with circles; points for random matrices are plotted as squares.

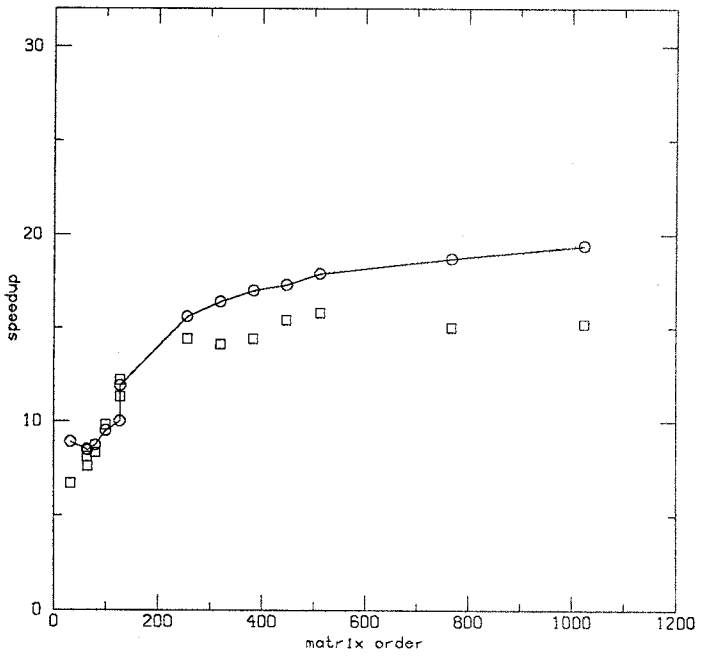

Figure 9: Multisection on a 5-cube: Speedup on the iPSC for [1,2,1] (circles) and random matrices (squares) *versus* Matrix Order.

The speedup for matrix [1,2,1] increases monotonically from 8.5 at order 64 to 19.4 at order 1024. The slight increase for speedup at 32 over that at order 64 is an artifact of having a problem size equal to the number of processors. Note that between orders 64 and 1024, the efficiency ranges between 26% and 60% as compared to 74%-89% for the same set of problems solved by bisection (see Figure 6). This relative lowering in efficiency for multisection is due, in part, to increased overhead. As shown in Figure 10, average communication and idle time account for over 40% of the total time for multisection at low orders and for 5% of the total time at order 1024. As shown in Figure 7, overhead

for the same range of problems solved by bisection varies from 20% to 2% of the total time. Non-parallel arithmetic in the multisection procedure occurs in the grouping of the eigenvalues by all processors, in the pipeline startup time for orthogonalization, and in the load balancing routines. These three processes contribute a combined total of less than 12% of the computation time for all matrix orders. The remaining loss of speedup can be attributed to inefficiency in the actual isolation and extraction processes. The presence of additional arithmetic in the determination of eigenvalues is evident in a division of total time according to computational tasks.

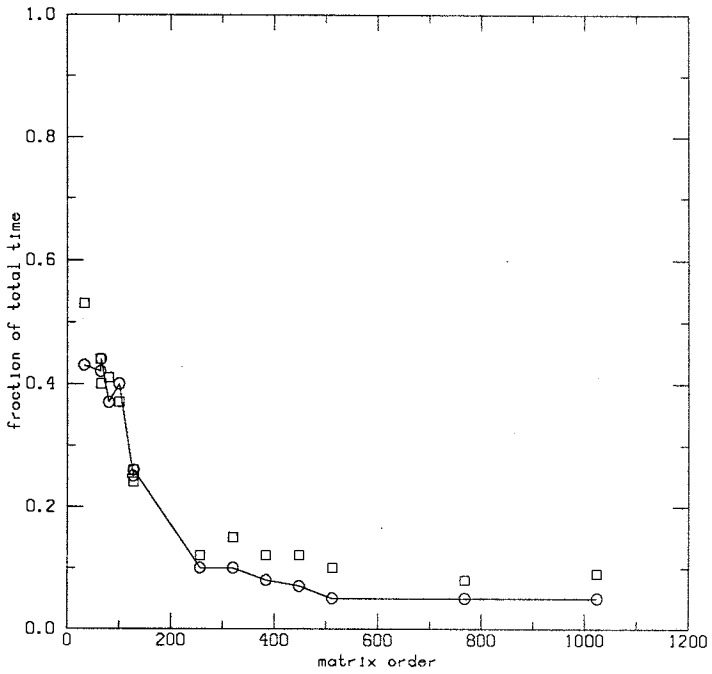

Figure 10: Multisection on an iPSC/d5M: Communication Overhead as a Fraction of the Total Time *versus* Matrix Order.

The average fractions of the total time spent in computation and grouping of eigenvalues, in computation of eigenvectors, and in orthogonalization of eigenvectors are shown in Figure 11. The fraction of time spent in eigenvalue computation is further partitioned into the fractions of the total time spent in isolating and extracting the eigenvalues. The complete eigenvalue computation is seen to take more than 90% of the total time for all matrix orders. The contribution of the eigenvector computation to the total grows from 7.5% to 11.6% between orders 32 and 1024, while the modified Gram-Schmidt procedure occupies up to 1.4% of the total. Note that the steps of Algorithms M2 and B2 pertaining to eigenvector computation and orthogonalization are identical and therefore take equal time. That these steps represent about 18.6% and 1.9% of the total time for bisection but only 11.6% and 1.4% of the time for multisection indicates that the eigenvalue computation takes not only a greater percentage of the total time for multisection but also a greater

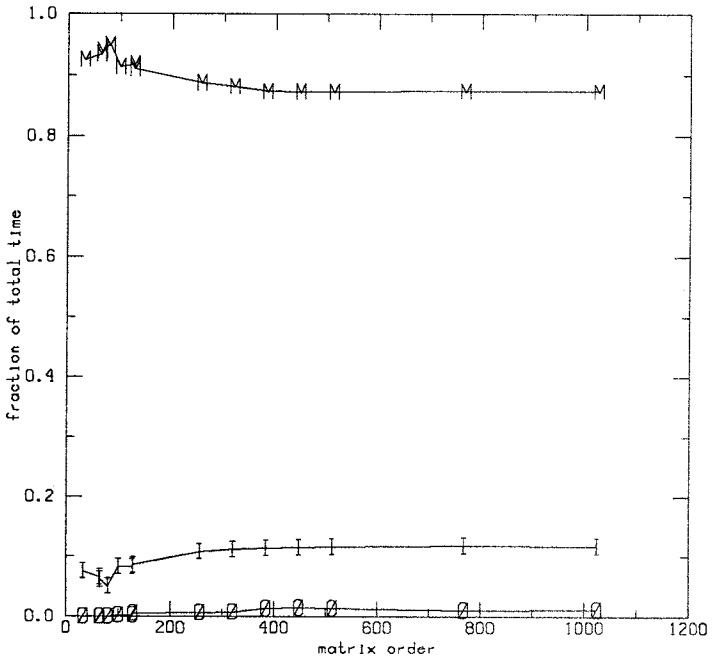

Figure 11: Multisection on a 5-cube: Fraction of Total Time Spent in Eigenvalue Computation (M), Eigenvector Computation (I), and Orthogonalization (O) *versus* Matrix Order. The fractions of time spent in isolation (i) and extraction (e) sum to the fraction in eigenvalue computation.

elapsed time. Some sample times for bisection and multisection given in Section 6 confirm that bisection is in fact the faster.

A return to the model problem of Section 4.3 helps to explain the the curves in Figure 11. Recall that T_{model} of order $N = kp$ has the N eigenvalues $0, \frac{\alpha}{N}, \ldots, \frac{(N-1)\alpha}{N}$ and an initial Gerschgorin interval reaching from 0 to α. Algorithm M2 begins by isolating the eigenvalues with multisection. The initial interval of length α is divided into p sections of length $\frac{\alpha}{p}$ and containing k eigenvalues. Every one of these sections is then multisected in turn to create a total of p^2 sections each containing $\frac{k}{p}$ eigenvalues. In general, the ith multisection step produces p^i intervals holding $\frac{N}{p^i}$ eigenvalues apiece. For the model problem, multisectioning stops when the resulting sections are no wider than $\frac{\alpha}{N}$, that is, after j steps, where $\frac{\alpha}{p^{j+1}} \leq \frac{\alpha}{N} < \frac{\alpha}{p^j}$. At each multisection step, a processor must evaluate Sturm sequences for both endpoints of every interval examined. The total number of Sturm sequence evaluations for the isolation phase is then

$$S_{isol} = 2\sum_{i=0}^{j} p^i \geq 2\frac{N-1}{p-1}.$$

Each Sturm sequence evaluation takes time $2N\alpha\omega$ so that

$$T_{isol} \geq 4\frac{N-1}{p-1}N\omega.$$

Isolation leaves in every processor a total of $p^j \leq \frac{N}{p}$ intervals of length $\frac{\alpha}{N}$ each containing one eigenvalue. Extracting a single eigenvalue to tolerance δ from one of the intervals requires l bisections, where $2^l \geq \frac{\alpha}{N\delta}$. The total number of Sturm sequence evaluations performed by one processor is then

$$S_{extr} = k(l+1) \geq k\left(\log_2 \frac{\alpha}{N\delta} + 1\right).$$

The total time for extraction is

$$T_{extr} \geq k\left(\log_2 \frac{\alpha}{N\delta} + 1\right) 2N\omega.$$

For all orders, the inital Gerschgorin interval of matrix [1,2,1] covers the real axis from 0 to 4. Thus, Figure 11 shows the variation in time distributions as an increasing number of eigenvalues are determined from an interval of constant length. This can be approximated analytically using the model problem with increasing N. For T_{model}, N eigenvalues are initially confined within an interval of width α.

As for Algorithm B2, time $T_I = 10Nk\omega$ is needed for inverse iteration for the model problem. The inter-eigenvalue spacing $\frac{\alpha}{N}$ is assumed large enough that none of the eigenvectors need be orthogonalized. The time for eigenvalue computation compared to the time for eigenvector computation for the double precision model problem is

$$\frac{T_M}{T_I} = \frac{T_{isol} + T_{extr}}{T_I} = \frac{1}{5k}\left(2\frac{N-1}{p-1} + klog_2\frac{\alpha}{N\delta} + k\right).$$

$\frac{T_M}{T_I}$ falls from about 10 at matrix order 32 to about 9 at order 1024. These values are in close agreement with those obtained experimentally for matrix [1,2,1]. The experimental results also exhibit the predicted decrease in the ratio of eigenvalue and eigenvector computation times for increasing matrix order.

For matrix orders below 200 in Figure 11, extraction and isolation take approximately equal portions of the total time. As the order increases, extraction dominates. For the model problem, the difference in the numbers of Sturm sequence evaluations for isolation and extraction is

$$D = S_{extr} - S_{isol} = k\left(\log_2 \frac{\alpha}{N\delta} + 1\right) - 2\frac{N-1}{p-1}.$$

D is positive whenever $\alpha\delta < 1$, so that more Sturm sequence evaluations are performed in extracting the eigenvalues of T_{model} than in isolating them. The derivative of D with respect to k is

$$\frac{dD}{dk} = \log_2 \frac{\alpha}{N\delta} + 1 - \log_2 e - 2\frac{p}{p-1} \approx k \log_2 \frac{\alpha}{N\delta}.$$

This quantity is greater than zero when $\frac{\alpha}{N} > \delta$ and decreases in magnitude with increasing N. Hence, D for the model problem has the same qualitative behavior as recorded for matrix [1,2,1].

While the model problem helps to explain some of the experimental results for [1,2,1], it does not account for the speedups observed for bisection and multisection. For matrix [1,2,1], the speedup of multisection is less than that of bisection for all matrix orders. For T_{model}, $T_B > T_M$ for all matrix orders, meaning that the predicted speedup is in fact greater for multisection than for bisection. This discrepancy stems from the differences in the spectra of the two matrices. While the eigenvalues of T_{model} are uniformly spaced, the eigenvalues of the Toeplitz matrix [1,2,1] of order N are given by $\lambda_i = 2(1 + \cos\frac{i\pi}{N+1})$ for $i = 1, 2, \ldots, N$ [21] and so are more tightly spaced at the extremes of the spectrum than near the center.

For a fixed number of eigenvalues, the time spent in isolation is greater for close eigenvalues than for well-separated. Furthermore, the time to extract eigenvalues from the narrow intervals arising from isolating close eigenvalues is less than for wider ones. An extreme example shows that the relative numbers of Sturm sequence evaluations for multisection and bisection are strongly dependent not only on the number of eigenvalues in the initial search area but also on their distribution within that interval. Let T'_{model} be a matrix of order N having $N = kp$ eigenvalues of multiplicity 2 uniformly distributed at spacing $2\frac{\alpha}{N}$ in the interval $[0, \alpha)$. Consider the case $N = 2p$. The number of Sturm sequence evaluations needed to determine all eigenvalues to a tolerance δ via Algorithm B2 on a p-processor hypercube is

$$S'_B = \log_2 \frac{\alpha}{\delta} + 2.$$

The number needed to find the eigenvalues using Algorithm M2 is

$$S'_M = \frac{\frac{\alpha}{\delta} - 1}{p - 1}.$$

Hence, $S'_M >> S'_B$, and the speedup of bisection is much greater than that of multisection.

That the speedup worsens as the role of isolation grows is evident in an examination of Figure 11. For matrix order 1024, where $\frac{T_{isol}}{T_{extr}}$ is about .27, the speedup for multisection is 67% that of bisection. For matrix order 32, $\frac{T_{isol}}{T_{extr}}$ is 1.3 and the speedup of multisection is only 35% that of bisection. For the random matrices tested, the effects are pronounced at order 1024 where $\frac{T_{isol}}{T_{extr}}$ is .46 and the ratio of speedups is .54. The relative times and speedups for the random matrix at order 64 are the same as for [1,2,1]. For intermediate orders of [1,2,1] and random matrices, increased isolation also leads to decreased speedup.

Matrices [1,2,1], T_{model}, and T'_{model} all indicate the trends of multisection as eigenvalues become increasingly clustered. Attempts to generalize this information by plotting time distribution as functions of the size or number of eigenvalue clusters for a variety of matrices of a single order were unsuccessful. Other problem dependencies mask any structure in the curves.

6. Comparison and Conclusion

Method	Order	Time (seconds)	Residual $\|TX - \Lambda X\|$	Orthogonality $\|X^T X - I\|$
Cuppen's	32	1.3	9.2e-16	7.0e-16
Bisection		0.6	9.4e-15	3.3e-14
Multisection		1.2	7.4e-15	3.2e-14
Cuppen's	100	10.5	1.9e-15	1.9e-15
Bisection		4.5	3.3e-14	3.1e-14
Multisection		9.7	1.3e-14	2.8e-14
Cuppen's	512	611.8	8.4e-15	1.8e-14
Bisection		88.7	8.8e-13	6.0e-13
Multisection		141.5	6.1e-13	3.9e-13

Table 2: Method Comparison for Matrix [1,2,1] on a 5-Cube.

Method	Order	Time (seconds)	Residual $\|TX - \Lambda X\|$	Orthogonality $\|X^T X - I\|$
Cuppen's	32	1.1	2.7e-15	2.7e-15
Bisection		0.5	2.9e-15	3.0e-14
Multisection		1.6	5.9e-15	1.2e-13
Cuppen's	100	10.4	7.4e-15	8.9e-15
Bisection		4.6	3.6e-14	6.5e-14
Multisection		10.7	4.7e-14	7.1e-14
Cuppen's	512	623.9	7.9e-15	1.3e-14
Bisection		88.3	5.3e-13	2.1e-13
Multisection		160.3	6.5e-13	7.1e-12

Table 3: Method Comparison for Random Matrices on a 5-Cube.

Tables 2 and 3 show the total time, the residual, and the deviation from orthogonality for several orders of matrix [1,2,1] and of random matrices, respectively, for all three eigensolvers. Bisection is seen to be the fastest technique for finding all the eigenvalues and eigenvectors at all orders. Although multisection can be shown analytically to have a lower time complexity than bisection for at least one contrived problem, in practice, arithmetic inefficiency in the isolation phase of multisection causes bisection to complete in less time. Multisection and Cuppen's method are of comparable speed at low orders; multisection is the more rapid of that pair for large orders.

As can be seen from the given measures of residual and orthogonality, introduction of communication for the hypercube does not alter the numerical properties of the methods. Cuppen's method gives the most accurate results, consistently achieving lower residual and orthogonality norms than both bisection and multisection for all matrices tested.

Although not discussed in this paper, the sectioning algorithms are readily modified to

permit computation of a subset of eigenvalues and eigenvectors. Cuppen's method can only be recommended when the entire eigensystem is needed. In that event, Cuppen's method, which requires sufficient storage for parallel matrix multiplication by Algorithm RMM, can be used for problems only as large as order 2080 on the iPSC/d5M. Bisection and multisection, which require no matrix multiplication, can both be used for problems through order 3520.

In preceding sections, speedup over the appropriate sequential method is shown to be problem dependent for all three schemes. Because speedup curves for given test matrices level off with increasing order, maximal speedup cannot be expected when bisection or multisection is applied to problems of large order, neither can it be expected when Cuppen's method encounters significant deflation in some but not all subproblems. In all three methods, efficient task scheduling is crucial for good performance. The extensive communication requirements of a dynamic scheduler for the hypercube and the potential loss of computing power from processors dedicated to scheduling tasks advise the use of static scheduling schemes. The resulting loss of parallelism at the root finding level for Cuppen's method and the simultaneous execution of an explicit scheduler on all processors for multisection, however, prevent full speedup.

The efficiency of solving the symmetric, tridiagonal eigenproblem on the local-memory hypercube as opposed to a shared-memory multiprocessor is dependent on the method used. In [23], timings are presented for SESUPD (a shared-memory implementation of Cuppen's method [12]), BISECT with TINVIT, and TREPS1 (multisection using bisection during the extraction phase) on eight processors of an Alliant FX/8. In particular, speedups of these algorithms with respect to the time to find all eigenvalues and eigenvectors of matrix [-1,2,-1] of order 500 via EISPACK's TQL2 on one processor are given. Defining a measure of the efficiency of algorithm i as compared to TQL2 by

$$E_i = \frac{\text{time for TQL2 on 1 processor}}{p * (\text{time for algorithm } i \text{ on } p \text{ processors})},$$

the shared-memory implementations have values $E_{SESUPD} = 3.4$, $E_{BISECT} = 0.5$, and $E_{TREPS1} = 4.0$. In contrast, the equivalent algorithms for the 5-cube have values $E_{C2} = 1.4$, $E_{B2} = 10.0$, and $E_{M2} = 6.2$ for matrix [1,2,1] of order 512. These relative efficiencies are greater than unity because TQL2 is slower on one processor than either SESUPD or BISECT. The greatest values of E_i occur for the algorithms which are most effectively implemented in parallel. The independent tasks of the bisection and multisection algorithms are seen to be especially adaptable to a local-memory architecture. Cuppen's method, on the other hand, is more efficient on the shared-memory machine, where a dynamic scheduler is supplied.

The hypercube implementations of all three methods are promising for large scale computation. Communication time represents a small portion of the total time for bisection and multisection as well as for Cuppen's method. The absence of communication bottlenecks indicates that matrix order is not an obstacle to performance on the hypercube. The large distributed memory is available without contention problems.

Acknowledgements

The authors wish to thank Stan Eisenstat, Bill Gropp, Cleve Moler, and Dan Sorensen

for many helpful discussions.

References

[1] R.H. Barlow and D.J. Evans, *A Parallel Organization of the Bisection Algorithm,* The Computer Journal, 22 (1977), pp. 267–69.

[2] R.H. Barlow, D.J. Evans, and J. Shanehchi, *Parallel Multisection Applied to the Eigenvalue Problem,* The Computer Journal, 26 (1983), pp. 6–9.

[3] W. Barth, R.S. Martin, and J.H. Wilkinson, Calculation of the Eigenvalues of a Symmetric Tridiagonal Matrix by the Method of Bisection, Contribution II/5 to, *Handbook for Automatic Computation, Vol.II: Linear Algebra,* Springer Verlag, 1971, pp. 249–256.

[4] K.-J. Bathe and E.L.Wilson, *Numerical Methods in Finite Element Analysis,* Prentice Hall, 1976.

[5] H.J. Bernstein, *An Accelerated Bisection Method for the Calculation of Eigenvalues of a Symmetric Tridiagonal Matrix,* Numer. Math., 43 (1984), pp. 153–60.

[6] H.J. Bernstein and M. Goldstein, *Parallel Implementation of Bisection for the Calculation of Eigenvalues of Tridiagonal Symmetric Matrices,* Computing, 37 (1986), pp. 85–91.

[7] S.N. Bhatt and I.C.F. Ipsen, *How to Embed Trees in Hypercubes,* Research Report 443, Dept Computer Science, Yale University, 1985.

[8] H. Bowdler, R.S. Martin, and J.H. Wilkinson, *The QR and QL Algorithms for Symmetric Matrices,* Num. Math., 11 (1968), pp. 227–240.

[9] J.E. Brandenburg and D.S. Scott, *Embedding of Communication Trees and Grids into Hypercubes,* Technical Report 1, Intel Scientific Computers, 1986.

[10] J.R. Bunch, C.P. Neilsen, and D.C. Sorensen, *Rank-One Modification of the Symmetric Eigenproblem,* Numer. Math., 31 (1978), pp. 31–48.

[11] J.J.M. Cuppen, *A Divide and Conquer Method for the Symmetric Tridiagonal Eigenproblem,* Numer. Math., 36 (1981), pp. 177–95.

[12] J.J. Dongarra and D.C. Sorensen, *A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem,* SIAM J. Sci. Stat. Comput., 8 (1987), pp. s139-s154.

[13] D.J. Evans, J. Shanehchi, C.C. Rick, *A Modified Bisection Algorithm for the Determination of the Eigenvalues of a Symmetric Tridiagonal Matrix,* Numer. Math., (1982), pp. 417–419.

[14] G.E. Forsythe, M.A. Malcolm, and C.B.Moler, *Computer Methods for Mathematical Computations,* Prentice Hall, 1977.

[15] G. Fox, A.J.G. Hey, S. Otto, *Matrix Algorithms on the Hypercube I: Matrix Multiplication,* Technical Report, California Institute of Technology, 1985.

[16] E.N. Gilbert, *Gray Codes and Paths on the N-Cube,* The Bell System Technical Journal, (May 1958).

[17] W. Givens, *Numerical Computation of the Characteristic Values of a Real Symmetric Matrix,* Technical Report ORNL-1574, Oak Ridge National Laboratory, 1954.

[18] G.H. Golub, *Some Modified Matrix Eigenvalue Problems,* SIAM Review, 15 (1973), pp. 318–34.

[19] G.H. Golub and C.F. van Loan, *Matrix Computations,* The Johns Hopkins Press, Baltimore, MD, 1983.

[20] R.L. Graham, *Bounds on Multiprocessor Timing Anomalies,* SIAM J. Appl. Math, 17 (1969), pp. 416–29.

[21] R.T. Gregory and D.L. Karney, *A Collection of Matrices for Testing Computational Algorithms,* John Wiley and Sons, Inc., 1969.

[22] A.S. Krishnakumar and M. Morf, *Eigenvalues of a Symmetric Tridiagonal Matrix: A Divide-and-Conquer Approach,* Numer. Math., 48 (1986), pp. 349–368.

[23] S. Lo, B. Phillipe, and A. Sameh, *A Multiprocessor Algorithm for the Symmetric Tridiagonal Eigenvalue Problem,* SIAM J. Sci. Stat. Comput., 8 (1987), pp. s155-s165.

[24] B.N. Parlett, *The Symmetric Eigenvalue Problem,* Prentice Hall, Englewood Cliffs, NJ, 1980.

[25] J.L. Peterson and A. Silberschatz, *Operating System Concepts,* Addison-Wesley Publishing Company, 1983.

[26] E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms,* Prentice Hall, Englewood Cliffs, NJ, 1977.

[27] H. Rutishauser, *Computational Aspects of F.L. Bauer's Simultaneous Iteration Method,* Numer. Math., 13 (1969), pp. 4–13.

[28] ————, *Simultaneous Iteration Method for Symmetric Matrices,* Numer. Math., 16 (1970), pp. 205–223.

[29] Y. Saad and M.H. Schultz, *Data Communication in Hypercubes,* Research Report 428, Dept Computer Science, Yale University, 1985.

[30] ————, *Some Topological Properties of The Hypercube Multiprocessor,* Research Report 389, Dept Computer Science, Yale University, 1984.

[31] S.K. Sahni, *Algorithms for Scheduling Independent Tasks,* JACM, 23 (1976), pp. 116–127.

[32] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, C.B. Moler, *Matrix Eigensystem Routines–EISPACK Guide, Lecture Notes in Computer Science, Vol. 6, 2nd edition,* Springer-Verlag, 1976.

[33] J.H. Wilkinson, *The Algebraic Eigenvalue Problem,* Clarendon Press, Oxford, 1965.

[34] A.Y. Wu, *Embedding of Tree Networks into Hypercubes,* Jour. Par. Distr. Comp., 2 (1985), pp. 238–49.