2

Optimal Eigenvalue Computation on Distributed-Memory MIMD Multiprocessors

S.Crivelli and E.R. Jessup
Department of Computer Science
University of Colorado, Boulder 80309-0430

CU-CS-617-92 October 1992

University of Colorado at Boulder

Technical Report CU-CS-617-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Optimal Eigenvalue Computation on Distributed-Memory MIMD Multiprocessors

S. Crivelli and E.R. Jessup*
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

October 1992

Abstract

In [13], Simon proves that bisection is not the optimal method for computing an eigenvalue on a single vector processor. In this paper, we show that his analysis does not extend in a straightforward way to the computation of an eigenvalue on a distributed-memory MIMD multiprocessor. In particular, we show how the optimal number of sections (and processors) to use for multisection depends on variables such as the matrix size and certain parameters inherent to the machine. We also show that parallel multisection outperforms the variant of parallel bisection proposed by Swarztrauber in [15] for this problem on a distributed-memory MIMD multiprocessor. We present the results of experiments on the 64-processor Intel iPSC/2 hypercube and the 512-processor Intel Touchstone Delta mesh multiprocessor.

1 Introduction

Bisection and multisection are simple and effective techniques for accurately solving the real symmetric tridiagonal eigenvalue problem. When one eigenvalue is needed, bisection begins with an interval known to contain that eigenvalue. The interval is halved and then replaced with the interval half found to contain the eigenvalue. The process repeats recursively until the length of the interval is less than a given threshold. The midpoint of the final interval is accepted as the computed eigenvalue [14]. Multisection is a generalization of bisection that splits the initial interval into $p + 1 \geq 2$ subintervals [10].

The bisection procedure, first described by Givens [5], uses the sequence of the determinants of the principal minors of a matrix to locate its eigenvalues. Let $\mathcal{T} = [\eta_j, \zeta_j, \eta_{j+1}]$ be a real, symmetric, tridiagonal $n \times n$ matrix, with diagonal elements ζ_j, for $j = 1, \ldots, n$, and off-diagonal elements $\eta_j \neq 0$, for $j = 1, \ldots, n-1$. Let $d_{i,j}$ denote the characteristic polynomials of the submatrices of $\mathcal{T}$ consisting of rows and columns i through j, then

$$d_{1,0}(\lambda) = 1$$

*Department of Computer Science, University of Colorado, Boulder, CO 80309-0430 (crivells@cs.colorado.edu and jessup@cs.colorado.edu). Both authors were funded by DOE contract DE-FG02-92ER25122 and by NSF grant CCR-9109785. Part of this work was completed when the second author was in residence at Oak Ridge National Laboratory and funded by DOE contract DE-AC05-84OR21400.

$$
\begin{aligned}
d_{1,1}(\lambda) &= \zeta_1 - \lambda \\
d_{i,j}(\lambda) &= (\zeta_j - \lambda) d_{i,j-1}(\lambda) - \eta_{j-1}^2 d_{i,j-2}(\lambda) \qquad 2 \le i, j \le n.
\end{aligned} \tag{1}
$$

In particular, the sequence of leading principal minors of the matrix $\mathcal{T} - \lambda$ is obtained by setting $i = 1$ in equation (1). The number of eigenvalues of $\mathcal{T}$ smaller than λ is equal to the number of sign agreements of the consecutive terms in the Sturm sequence $\{d_{1,j}(\lambda)\}$ [5].

Because the linear recurrence (1) may cause overflow or underflow problems, it is preferable to use the Sturm sequence $\{f_j(\lambda)\}$ defined as

$$
\begin{aligned}
f_0(\lambda) &= \zeta_0 - \lambda \\
f_j(\lambda) &= \zeta_j - \lambda - \frac{\eta_j^2}{f_{j-1}(\lambda)} \qquad j = 1, \ldots, n-1
\end{aligned} \tag{2}
$$

where $f_j(\lambda) = d_{1,j}(\lambda)/d_{1,j-1}(\lambda)$. In this case, the number of eigenvalues of $\mathcal{T}$ smaller than λ is equal to the number of negative terms $\sigma(\lambda)$ in the sequence $\{f_j(\lambda)\}$ [2], and the number of eigenvalues in the interval $[\lambda_{-1}, \lambda_0)$ is the difference $\sigma(\lambda_0) - \sigma(\lambda_{-1})$. (Although it is generally more robust than sequence (1), Kahan shows that sequence (2) may sometimes overflow as well [9].)

It is possible to achieve parallelism either by using the recurrence formula (2) or by directly computing the principal minors of $\mathcal{T} - \lambda$ and their characteristic polynomials. In the first case, parallelism can be attained either by simultaneously evaluating the Sturm sequence at p interior points of the search interval, one evaluation per processor, or by computing different eigenvalues in parallel [7, 8]. In the second case, parallelism can be attained by using the associative property of matrix multiplication which allows splitting the evaluation of the minors among the different processors [15].

Several parallel bisection and multisection procedures have been proposed since Huang's work on the ILLIAC IV [6], but little has been proven about their efficiencies. Lo, et.al. [10] propose a combination algorithm for the Alliant FX/8 in which eigenvalues are first *isolated* within disjoint subintervals through parallel multisection and then *extracted* to a given precision by serial bisection. The second step is made parallel by assigning different eigenvalues to different processors. Their rationale for using multisection in the isolation phase is that multisection creates more tasks than bisection, thus allowing faster isolation of the eigenvalues and better processor load balance. Bernstein and Goldstein [3] argue against these reasons, noting that multisection may create a large number of tasks associated with empty intervals. They claim that an accelerated bisection technique should lead to better processor utilization with less overnead when the number of eigenvalues computed is large compared to the number of processors.

Ipsen and Jessup [7] show that on a distributed-memory multiprocessor (Intel's iPSC/1 hypercube), interprocessor communication and processor synchronization costs of the multisection phase make the combined algorithm of [10] slower than the bisection procedure alone. On the other hand, Simon [13] defends multisection by proving that, for a single vector processor, bisection is not the optimal method for computing one eigenvalue.

It is known that a single eigenvalue is computed more efficiently by serial bisection than by serial multisection [10]. In this paper, we develop a formal analysis of the relative costs of bisection and multisection on a distributed-memory multiprocessor. We compare the costs of computing a single eigenvalue by serial bisection, by parallel multisection, and by Swarztrauber's parallel bisection [15]. We show how the optimal method for computing one eigenvalue depends

on a number of variables, such as the size of the matrix and certain parameters of the multiprocessor used. Our analysis is supported by experiments on a 64-processor Intel iPSC/2 hypercube multiprocessor and the Intel Touchstone Delta, a distributed-memory MIMD machine whose 512 i860-based nodes are interconnected as a two-dimensional rectangular grid.

The paper is organized as follows. In section 2, we derive an analytical expression for the arithmetic time required to compute a single eigenvalue by multisection. We also analyze two communication schemes that are competitive on the hypercube and present the time complexity for each approach. In section 3, we determine the optimal approach by deriving an approximation to the optimal number of processors and obtaining an estimate of the error incurred by using this approximation. Numerical results are presented to illustrate the behavior of the cost function proposed in this work. The time complexity for multisection on mesh-connected architectures is investigated in section 4. In section 5, we present some comparisons between the analysis of multisection costs for the hypercubes and the one obtained by Simon for vector processors in [13]. In section 6, we derive an analytical expression for the time required to compute a single eigenvalue by Swarztrauber's parallel bisection and compare it to the multisection costs obtained in section 2. Finally, we present a brief summary in section 7.

2 Time Complexity Analysis for Multisection

In this section, we develop an analytical expression for the time required to compute a single eigenvalue by multisection on a hypercube multiprocessor. First, we determine the number of iterations needed to extract an eigenvalue from an interval of width l when the interval is split into $p+1$ subintervals at each iteration. If the final interval width turns out to be $\delta = l\epsilon$, where ϵ is a given threshold, it is necessary to carry out at least k serial *multisection* steps, until

$$l/(p+1)^k \leq \delta.$$

Replacing δ with its value $l\epsilon$ and taking the logarithm gives

$$k_p = \lceil -\log(\epsilon)/\log(p+1) \rceil.$$

Note that when $p = 1$, k_1 gives the number of *bisection* steps. This argument is independent of the base of the logarithm, but throughout this paper, we take $\log(x)$ to mean $\log_2(x)$.

2.1 Computation Cost

At each iteration, p division points are determined and the recurrence in equation (2) is evaluated at each point. If the interval endpoints are λ_{-1} and λ_0, the distance between division points is $\Delta = (\lambda_0 - \lambda_{-1})/(p+1)$, and the division points are $\lambda_i = \lambda_{-1} + i\Delta$, $i = 1, \ldots, p$.

The cost of computing these points is $p\omega_1 + \omega_2$, where ω_1 is the time for a floating point addition or subtraction and ω_2 is the time for a floating point multiplication or division. The Sturm sequence evaluation at each point takes n floating point divisions and $2n$ floating point subtractions. In addition, counting the negative terms in the sequence requires n floating point comparisons. Therefore, the total cost for computing a single eigenvalue by the serial multisection algorithm is

$$T_M = k_p * [p * (n * (2\omega_1 + \omega_2 + \gamma_2) + \omega_1) + \omega_2],$$

where γ_2 is the time for a floating point comparison.

When the p Sturm sequences are evaluated in parallel —one per processor— at each multisection step, processors must communicate to determine the next search interval. Because communication costs can be quite significant in these parallel algorithms, we examine efficient communication schemes in the following subsection.

2.2 Communication Cost

On existing hypercube multiprocessors, the cost of data communication is high in comparison to the cost of floating point computation. In particular, the cost of communicating an m-byte message from one hypercube processor to a neighboring one is $\beta + m\,\tau$, where the communication startup latency β is generally large in comparison to the transmission time per byte τ and to the time for a floating point operation. On the iPSC/2, $\beta/\tau = 975$, $\beta/\omega_1 = 59$, and $\beta/\omega_2 = 56$ [4].

These cost ratios lead us to consider only communication schemes that minimize the number of message startups, i.e., that use a number of message startups proportional to the dimension d of the hypercube, with $p = 2^d$ processors [11]. We do not, however, attempt to decrease the impact of communication through redundant computation. Specifically, a processor evaluates the Sturm sequence at only one division point instead of determining the number of eigenvalues within an interval by evaluating the Sturm sequence at both endpoints.

There are two basic mechanisms for determining the next search interval from the distributed Sturm sequence counts in $O(d)$ communication steps. These depend on all-to-one or all-to-all communication primitives as discussed, for example, in [11].

2.2.1 The Gather-Broadcast Approach (GB)

The first approach is a gather-broadcast routine in which each processor sends its Sturm sequence count to a single master processor. That processor then computes the endpoints of the next search interval and broadcasts them back to all the other processors. Because the hypercube is a connected graph, it is possible to consider it as a spanning tree, which is a sub-tree of the graph that contains the master as root and all other processors as leaves. When a spanning tree based gather and broadcast are used, the communication cost is

$$2d\beta + (d-1)(4+2*8)\tau,$$

for an integer*4 processor number and real*8 endpoints [11].

The total time to compute one eigenvalue using this algorithm is

$$\begin{aligned} T_{GB} &= \{\text{time for one eigenvalue count} \\ &\quad + \text{time to find a new interval by gather-broadcast}\} \\ &\quad *\text{number of iterations} \\ &= \{n(2\omega_1 + \omega_2 + \gamma_2) + \\ &\quad 2d\beta + (d-1)20\tau\}\lceil -\log\epsilon/\log(2^d+1)\rceil. \end{aligned}$$

2.2.2 The Alternate Direction Exchange Approach (ADE)

The second communication option is based on alternate direction exchange (ADE) as described in [11]. ADE is typically used to accumulate in all $p = 2^d$ processors a vector of length pk whose

components are initially distributed evenly among them. In each of d communication steps, the d-cube splits into a different pair of $(d-1)$-cubes. In the first step, corresponding processors in the two cubes exchange their k elements and accumulate a vector of length $2k$. In subsequent steps, the processors exchange and accumulate all previously accumulated data so that in the last step processors send messages of length $2^{d-1}k$ and accumulate the full vector.

In the case of multisection, it is not necessary for all processors to accumulate all Sturm sequence counts. If eigenvalue m is sought, each processor only needs the division points λ_i and λ_{i+1} surrounding eigenvalue m. If division point λ_i is assigned to processor i, $i = 1, \ldots, p$, this information can be computed by any processor from Sturm sequence counts $\sigma(i)$ marked with the identifiers of the counting processor i. At each step of the ADE for multisection, each processor retains only the count closest to the eigenvalue index m.

Thus, at each step of the ADE for multisection, two integers (i and $\sigma(i)$) are sent in each direction between each pair of processors. The processors then determine which identifiers to retain. Every step then takes

$$2d(\beta + (2 * 4)\tau) + d\gamma_1,$$

where γ_1 is the time for an integer comparison. If the machine allows simultaneous bidirectional sends across the same communication channel, the total cost reduces to

$$d(\beta + (2 * 4)\tau + \gamma_1),$$

and the alternate direction scheme involves roughly half the communication latency of the gather-broadcast scheme. The iPSC/2 supports such sends when the message size is less than 100 bytes [12].

Using the ADE communication scheme, the total cost for computing a single eigenvalue by parallel multisection becomes

$$\begin{aligned} T_{ADE} &= \{\text{time for one eigenvalue count} \\ &\quad + \text{time to find a new interval by ADE}\} \\ &\quad * \text{number of iterations} \\ &= \{n(2\omega_1 + \omega_2 + \gamma_2) + \omega_1 + \omega_2 + \\ &\quad d(\beta + 8\tau + \gamma_1)\} * \lceil -\log(\epsilon)/\log(2^d + 1)\rceil. \end{aligned}$$

When $d = 0$, multisection reduces to serial bisection.

Therefore, the ADE approach performs better than GB for parallel multisection because it requires half the number of communication startups.

3 The Optimal Approach

In this section, we use parallel multisection based on an ADE in order to determine the optimal number of sections and processors to use. However, it is important to note that the same conclusions can be derived by using any other communication scheme that involves $O(d)$ startups.

For further analysis, we neglect the contributions of data transfer τ and integer comparison γ_1 because the startup time β is much larger than either ($\beta \gg \tau, \gamma_1$) and write α = (arithmetic costs)/β. The cost function for multisection then becomes

$$T_c = \beta(d + \alpha)\lceil -\log \epsilon / \log(2^d + 1)\rceil. \tag{3}$$

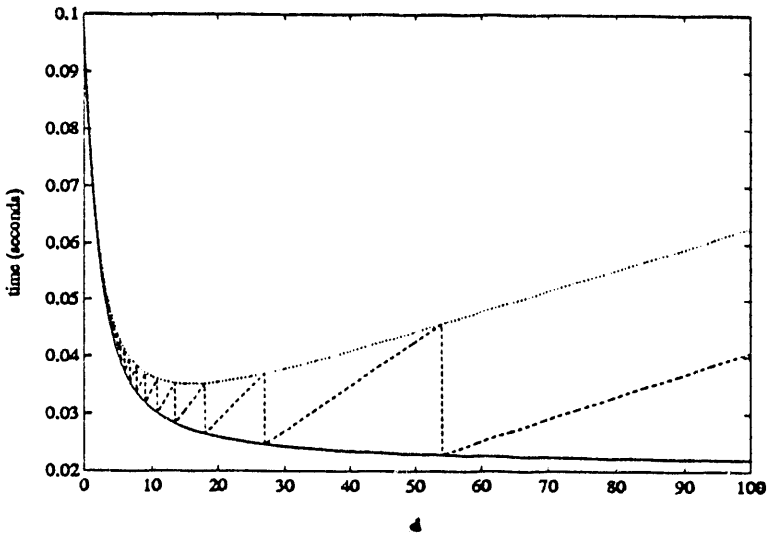

Figure 1: Functions T_c, T_-, and T_+.

The location of the minimum of T_c defines the optimal number of division points to use in bisection or multisection. In this section, we determine how the location of that minimum depends on the parameters α, β and ϵ.

Because the straightforward analysis of the ceiling function in T_c is unwieldy, we first study the auxiliary functions T_- and T_+

$$\begin{aligned} T_- &= \beta(\alpha + d)\frac{-\log \epsilon}{\log(2^d + 1)} \\ T_+ &= T_- + \beta(\alpha + d). \end{aligned} \tag{4}$$

Figure 1 depicts the functions T_c, T_-, and T_+ plotted against the continuous variable d for $\alpha = 4.6$, $\beta = 390$ and $-\log(\epsilon) = 54$. This choice of variables corresponds to computing an eigenvalue of an order 100 matrix on an iPSC/2 to precision $\epsilon/2 \approx 10^{-16}$.

3.1 Estimating the Location of the Minimum of T_c

As d increases, the cost function T_c zigzags between T_- and T_+, with its local maxima lying on T_+ and its local minima on T_-, until the point where it grows linearly as the function $\beta(\alpha + d)$. Function T_- has one global maximum then decreases monotonically, converging asymptotically to the constant function $-\beta \log \epsilon$. Function T_+ increases then, depending on the value of α, may fall to one local minimum; T_+ ultimately rises asymptotically to the function T_c. In Figure 1, T_+ has a local minimum for a value of $d > 0$.

Because T_c does not have a global minimum, we need to study the behavior of the function in the interval of processors available to us in order to find a local minimum. Let $D = [0, d_{map}]$ denote that interval, where d_{map} is the maximum number of available processors. The location d_{min} of the minimum value of T_c in D determines the number $p_{min} = 2^{d_{min}}$ of division points (and processors) to use for most efficient bisection or multisection.

Because the minima of T_c lie on the smooth curve T_-, we employ T_- to estimate the global minimum of T_c in D. Because T_- has exactly one maximum located at d_{max}, its minimum in D lies at an endpoint of that interval. We will locate this global minimum of T_- in D relative to the position of the global maximum of T_-.

Now, if $d_{max} \leq 0$, T_- decreases monotonically in the interval D and takes its minimum value at $d = d_{map}$. Let us assume for the moment that T_c and T_- are somewhere coincident in D after reaching their maximum values. In particular, if $T_c(d_{map}) = T_-(d_{map})$, the minimum of T_c also occurs at $d = d_{min} = d_{map}$. However, because T_c is not smooth, its minimum may actually lie to the left of $d = d_{map}$. In fact, the minimum of T_c lies at the largest $d \leq d_{map}$ such that $T_c(d) = T_-(d)$. In this case, $d_{min} > 0$, and multisection with $p = 2^{d_{min}}$ is the most efficient method.

If $d_{max} > 0$, the minimum value of T_- is attained at one of the endpoints of D. Hence, the minimum of T_c is attained either at $d = 0$ or at some $0 < d \leq d_{map}$. (As above, d_{min} can equal d_{map} only if $T_c(d_{map}) = T_-(d_{map})$.) There are then two possibilities:

1. If $d_{min} = 0$, increasing the number of division points increases the run time. Therefore, bisection is optimal.

2. If $d_{min} > 0$, multisection into $p + 1 = 2^{d_{min}} + 1$ intervals is optimal.

Because the sign of d_{max} is so important we now investigate d_{max}. The maximum of T_- lies where $\partial T_-/\partial d = 0$. Although the zero of this derivative cannot be determined analytically, we can determine the problem size at which the maximum lies at $d_{max} = 0$. Manipulation of $\partial T_-/\partial d = 0$ leads to

$$\frac{(2^{d_{max}} + 1)\log(2^{d_{max}} + 1)}{2^{d_{max}}} - d_{max} = \alpha.$$

When $d_{max} = 0$, $\alpha = 2$. For problems with $\alpha \geq 2$, $d_{max} \leq 0$, and multisection is always the most efficient option.

It remains to determine when multisection should be applied if $\alpha < 2$, which corresponds to the case $d_{max} > 0$ discussed above. Note that bisection and multisection take the same time to compute the eigenvalue when $T_c(0) = T_c(d_{map})$, that is, when

$$\alpha = d_{map}\left\lceil \frac{-\log\epsilon}{\log(2^{d_{map}} + 1)} \right\rceil \Big/ \left(\lceil -\log\epsilon \rceil - \left\lceil \frac{-\log\epsilon}{\log(2^{d_{map}} + 1)} \right\rceil \right).$$

On the iPSC/2 the maximum number of available processors corresponds to a cube dimension $d_{map} = 6$. Thus, assuming $-\log(\epsilon) = 54$, the crossover point between multisection into $p_{min} + 1$ intervals and bisection occurs at $\alpha = 1.2$. Now, since $\beta = 390$ microseconds and $\alpha \approx 0.046n$, this crossover point corresponds to a matrix of order $n \approx 27$.

To find the minimum of T_c we assumed that T_c and T_- were coincident in at least one point of interval D. To prove it, we need the following lemma.

Lemma 3.1 *Let S be the step function defined as:*

$$S(d) = \lceil -\log\epsilon/\log(2^d + 1) \rceil$$

and S_- the continuous function defined as:

$$S_-(d) = -\log\epsilon/\log(2^d + 1).$$

Then the distance Δ between two consecutive peaks of the function T_c is such that

$$\Delta \leq \frac{-1}{\partial S_- / \partial d}.$$

Proof: The distance between two consecutive peaks of the function T_c is determined by the width of the steps of the function S. If Δ is the width of the step where $S(d)$ lies, the following holds:

$$S_-(d+\Delta) - S_-(d) \leq -1. \tag{5}$$

Because S_- is monotonically decreasing, for any increment in d, δd, we get

$$\frac{\partial S_-}{\partial d}\delta d \leq S_-(d+\delta d) - S_-(d). \tag{6}$$

where the derivative $\partial S_- / \partial d$ is evaluated at d. In particular, taking $\delta d = \Delta$ and applying (5) to (6) we get:

$$\Delta \leq \frac{-1}{\partial S_- / \partial d}.$$

■

According to the lemma, at $d = 0$, $\Delta \leq 2/-\log \epsilon$. Consequently, to have at least one coincident point in $D = [0, d_{map}]$, we need $\Delta < d_{map}$ which is accomplished for those values of ϵ such that $\epsilon < 2^{-2/d_{map}}$. Using $d_{map} = 6$ we get $\epsilon < 0.7937$, which in general can be easily satisfied for $\epsilon =$ length final interval/length initial interval. Recall that, in our example on the iPSC/2, we consider $\epsilon/2 \approx 10^{-16}$ which means that we can expect coincidence between T_- and T_c in the interval D.

3.2 The Error Incurred by Using T_- to Model T_c

We have determined that the optimal number of processors is given by $p = 2^{d_{min}}$, where d_{min} is the minimum of T_c in D. Because we cannot determine d_{min} analytically, we decided to use the minimum of T_- in D which occurs either at $d = 0$, meaning that bisection is optimal, or d_{map}, meaning that multisection is optimal. Next, we determine the error introduced by using the minimum of T_- in place of the minimum of T_c when multisection is optimal. Because T_c zigzags between T_- and T_+, the maximum error made by taking d_{map} in place of d_{min} is given by $T_+(d_{map}) - T_-(d_{map}) = \beta(\alpha + d_{map})$.

Although we choose d_{map} as an approximation for the minimum of T_c, it is important to note that there are some cases where it really is the optimal choice. In fact, there are some values of α for which the function T_- is so steep at d_{map} that

$$|T_-(d_{map}) - T_-(d_{map} - 1)| > T_+(d_{map}) - T_-(d_{map}) = \beta(\alpha + d_{map}).$$

That is, the change in T_- for d varying from $d_{map} - 1$ to d_{map} is larger than the error incurred by considering d_{map} as the minimum. That error is bounded by the difference between T_+ and T_- at d_{map}.

In our example on the iPSC/2, we get $\alpha > 8.73$. This means that when the size of the matrix is larger than 189, d_{map} is a good approximation, and we can choose multisection with $p = 2^{d_{map}}$ as the optimal method.

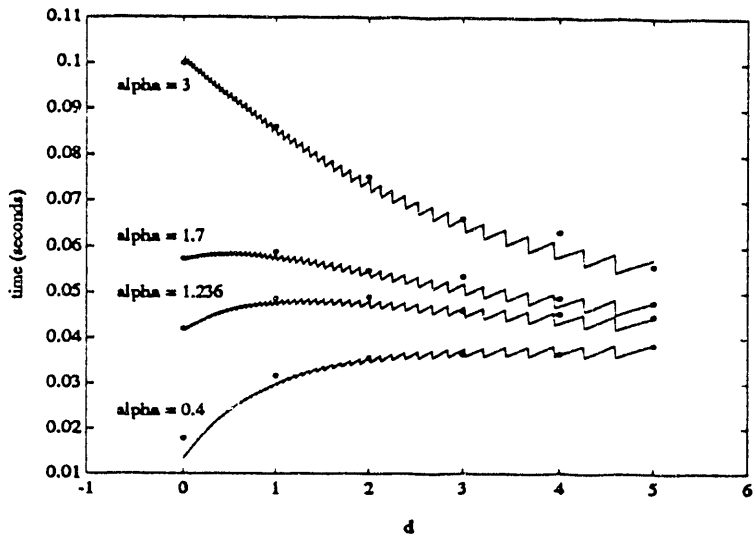

Figure 2: The cost function T_c for different values of α (computed and measured).

Figure 2 depicts the function T_c plotted against the continuous variable d for several values of α (solid lines as marked). Also shown are times measured on the iPSC/2 (circles) for hypercubes of dimension $d = 0, 1, \ldots, 6$. On a cube of dimension d, $p = 2^d$ division points are used. The figure shows good agreement between the model and actual timings in the range of available processors. Table 1 shows experimental results on the iPSC/2 using $d_{map} = 6$.

We have seen that the maximum error made by taking the minimum of T_- instead of the minimum of T_c is bounded by the difference between $T_+(d_{map})$ and $T_-(d_{map})$. Because this difference increases as d increases, it is necessary to analyze the benefits of adding more and more processors versus the magnitude of the error incurred. Furthermore, the function T_c becomes a straight line after some point, after which it is no longer beneficial to add processors.

To facilitate the analysis, we define two important values of d: $d_c \neq 0$ is the point at which T_+ attains its local minimum (if there is one), and d_l is the point at which T_c begins to rise linearly. The point d_l is thus the smallest value of d that satisfies the equation $\lceil -\log \epsilon / \log(2^d + 1) \rceil = 1$. These values in turn distinguish three regions:

region 1 begins at $d = 0$ and ends at the point $min(d_c, d_l)$. In this region, T_c zigzags between two decreasing functions. Therefore, we can expect better results when we increase the number of processors.

region 2.a begins at the point $min(d_c, d_l)$ and ends at d_l. Since the cost function jags between the asymptotically decreasing function T_- and the now increasing function T_+, processors in this region must be added carefully. In fact, while we still can get better results by adding more processors, we can also incur substantial errors by approximating the minimum of T_c with the minimum of T_-.

region 2.b begins at d_l. In this region the cost function increases linearly as $\beta(\alpha + d)$. Therefore, we will not find any minimum value greater than d_l for T_c in this region.

Bisection vs. Multisection				
n	d_{min}	bisection	multisection	ratio
10	0	0.02336	0.04472	1.91
20	0	0.03726	0.04342	1.17
25	0	0.04330	0.04472	1.03
26	0	0.04460	0.04496	1.01
27	5	0.04696	0.04522	0.96
28	5	0.04832	0.04542	0.94
29	5	0.04968	0.04566	0.92
30	5	0.05116	0.04606	0.90
50	5	0.07900	0.05186	0.66
100	5	0.14852	0.07218	0.49

Table 1: Execution times for computing the smallest eigenvalue of matrix [1,2,1] on the iPSC/2.

We now estimate the values of d_c and d_l in order to define those regions more precisely. To find the point d_c we need to find the solution to $\partial T_+/\partial d = 0$. Because we cannot solve this equation analytically we approximate $1/\log(2^d+1)$ by $1/d$. The relative error that this approximation incurs is

$$E_r(d) = |\frac{1/\log(2^d+1) - 1/d}{1/\log(2^d+1)}| = |1 - \frac{\log(2^d+1)}{d}|,$$

which is small for $d \geq 1$. In fact, it can be easily seen that, for any $d > 1$, $E_r(d) > E_r(1) \approx 0.58$. Therefore, for $d \geq 1$, T_+ can be approximated by the function:

$$T_+ \approx \beta(\alpha + d)\left[\frac{-\log\epsilon}{d} + 1\right],$$

and its first derivative is

$$-\beta = \partial T_+/\partial d \approx \beta\left[\frac{\log(\epsilon)\alpha}{d^2} + 1\right] = 0.$$

Manipulating this expression and considering only the positive root we get

$$d_c = \sqrt{-\log(\epsilon)\alpha}. \tag{7}$$

To find the point d_l where the cost function T_c becomes a straight line, it is necessary to solve

$$-\log\epsilon/\log(2^d+1) = 1.$$

Manipulating this equation we get d_l as:

$$d_l = \log\frac{1-\epsilon}{\epsilon}. \tag{8}$$

Finally, to illustrate how the different regions look like, we give particular values to the parameters in equations (7) and (8). Recall that, on the iPSC/2, a value of $\alpha = 4.6$ corresponds to a matrix of order 100. Thus, for the problem of evaluating a single eigenvalue of an order 100 matrix on an iPSC/2 to precision $\epsilon/2 \approx 10^{-16}$, the values of d_c and d_l are 13 and 37 respectively. Because $d_{map} = 6$, we can conclude that, for these parameters, D is contained in region 1. Because $\alpha > 1.2$, multisection performs better than bisection, and therefore, we can take d_{map} as the optimal number of processors to use in this problem.

4 On Mesh-Connected Architectures

In the preceeding sections we analyzed parallel implementations of multisection and bisection on the hypercube. Because of the high ratio between the communication and computation costs on the iPSC/2, we considered only those algorithms involving $O(d)$ communication startups. Among those approaches, the Alternate Direction Exchange multisection algorithm was shown to be the most efficient optimal approach. In fact, while ADE takes only d communication startups to determine the new search interval, the GB approach takes $2d$ startups to accomplish the same task. However, although ADE outperforms GB on the hypercube, this algorithm is intimately tied to the hypercube topology. Because we cannot implement ADE on mesh-connected architectures, we investigate in this section the time complexity of the GB approach.

As described for hypercubes in section 2.2.1, each processor computes a Sturm sequence count and sends it to the master processor. The master gathers the information, computes the next interval, and broadcasts it back to the rest of the processors. While the computation cost of this algorithm is the same for mesh-connected processors as for hypercubes, it is necessary to reconsider the communication time complexity for the new architecture. According to [1], there is an optimal broadcast (gather) algorithm for meshes that does not cause network contention and has the same logarithmic time complexity as for hypercubes. Figure 3 depicts a contention-free broadcast from node 0 to all others on a linear array [1].

If we have a two-dimensional grid of $p = p_1 \times p_2$ points, where $p_i = 2^{d_i}$, $i = 1, 2$, it will be necessary to perform $d = d_1 + d_2 = \log(p_1) + \log(p_2) = \log(p)$ steps to complete a broadcast (gather) operation [1]. The basic idea is to partition the two-dimensional array along one dimension, thereby reducing the problem to that for linear arrays. These, in turn, are recursively partitioned, doubling the number of partitions at each step, and creating distinct sub-arrays which can proceed independently with the broadcast (gather) procedure. In this way, a minimum spanning tree broadcast (gather) can be performed on mesh-connected architectures as efficiently as on hypercubes. The only difference is in the way that the minimum spanning tree is derived using the binary representation of the nodes. While the order in which bits are toggled to derive the tree is not important for hypercubes, it is for meshes due to contention problems [1].

Therefore, under reasonable assumptions, the time complexity analysis of the GB multisection approach remains the same on both hypercube and mesh-connected architectures. Figure 4 depicts the cost function T_{GB} corresponding to the GB multisection approach for computing the smallest eigenvalue of the 1000×1000 matrix $[1, 2, 1]$ plotted as a solid line against the continuous variable $d = \log(p)$. On the same plot, it also shows the times measured on the Intel Touchstone Delta (circles). The values of the parameters used in computing T_{GB} are $-\log(\epsilon) = 54$, $\beta = 75$, and $\omega = \gamma = 0.1$. These values correspond to double precision accuracy on the Delta. The figure shows good agreement between the theoretical and the actual timings in the range of available processors.

5 On Vector Processors

In this section, we compare the cost function T_c for the hypercube, defined in equation (3), to the cost function C for vector processors obtained by Simon [13]:

$$C = (s + rp)\lceil -\log \epsilon / \log(p + 1)\rceil,$$

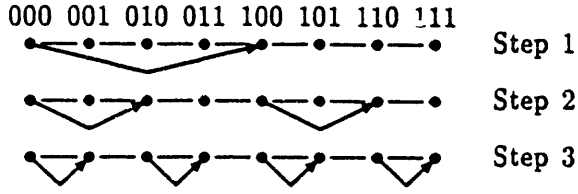

Figure 3: Broadcasting a message from node 0 on a linear array (most significant bit to least significant bit). Source: [1].

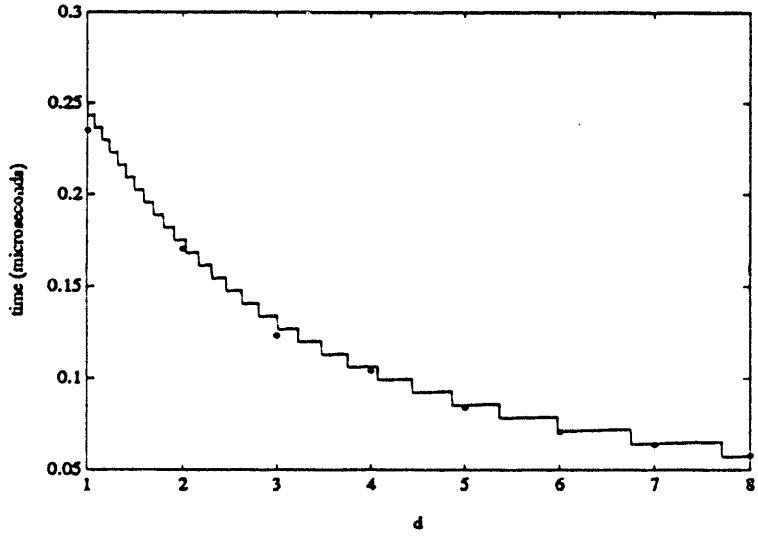

Figure 4: Theoretical and actual values of the cost function for computing an eigenvalue of the $[1, 2, 1]$ matrix of order 1000 on the Delta.

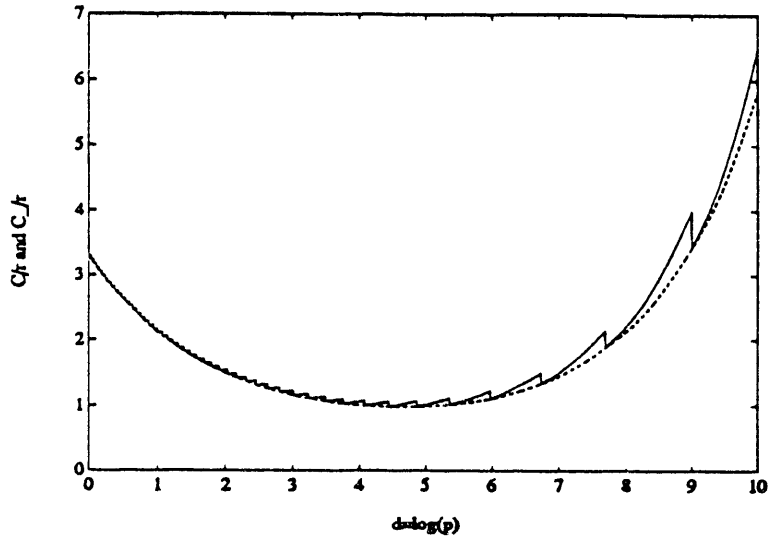

Figure 5: The scaled vector cost functions $\frac{1}{r}C$ and $\frac{1}{r}C_{-}$. These dimensionless quantities show the optimal number of sections to use on a vector processor.

where s is the startup cost of the vector loop, r the asymptotic rate, and p the number of multisection points. We confirm that the analysis is considerably more straightforward for a vector processor than for a distributed-memory multiprocessor.

A comparison of Figures 1 and 5 shows that T_c differs dramatically from C. The reason is that, on vector processors, the vector loop startup cost is independent of the number of multisection points while the asymptotic arithmetic cost increases linearly with this number.

In the vector case, a minimum always occurs for some number of division points $p > 1$. Thus, multisection is always optimal on vector processors. In contrast, the distributed-memory cost function T_c does not have a global minimum but instead a global maximum. Therefore, a study of the function in the interval of available processors is necessary to find the local minimum. Our analysis shows different behaviors of T_c depending on the values of the different parameters involved. Thus, in the hypercube case, the optimal method depends on the problem order, the ratio between communication and computation costs, and on the number of processors available.

Although it is clear from [13] that multisection into $p + 1$ intervals is the best choice in the vector case, it is not straightforward which value of p produces the minimum cost. For instance, for a machine with ratio $s/r = 60$, the value of p_{min} obtained by Simon from running his multisection code on a random matrix of order 3000 is 37, whereas the predicted p is 25. Simon attributes this difference to the fact that the predicted value is obtained considering only the multisection loop, whereas the experimental p_{min} comes from taking into account the execution time of the whole multisection code.

In his analysis, Simon actually uses the function

$$C_{-} = (s + rp)(-\log \epsilon / \log(p + 1))$$

analogous to T_{-} defined in equation (4) for distributed-memory multiprocessors. Figure 5 depicts the functions C and C_{-} plotted against the continuous variable $d = \log(p)$ for $s/r = 60.0$ and $-\log(\epsilon) = 54$.

In this example, the optimal number of processors, p_{min}, obtained by neglecting and using the ceiling function are 25 and 22 respectively, whereas the value obtained experimentally by Simon is 37. Observe that although the actual minimum of C occurs at $p \approx 29.1$, we consider only the integer values of p_{min}. That is the reason why we take $p = 22$ as the minimum value.

Because the approximation of T_c with T_- may incur significant errors, we first thought, by analogy, that the discrepancies between the predicted p and the one obtained numerically by Simon could be partially attributed to the neglecting of the ceiling function. However, the results obtained by using C are only slightly different from those obtained by using C_- and consequently do not justify this argument. Ignoring the ceiling in the vector cost function C does not affect the final conclusions about the optimal value of p. See Figure 5.

Finally, because the location of the extreme value of C may vary with ϵ, we compute the value of p_{min} for $\epsilon = 10^{-8}$. The minimum for this single precision example also occurs at $p = 22$ and does not affect our conclusions.

6 Time Complexity Analysis for Swarztrauber's Parallel Bisection (SPB)

In this section, we develop an analytical expression to estimate the time required to compute a single eigenvalue by the parallel bisection approach proposed by Swarztrauber [15]. This method is based on the parallel evaluation of the Sturm sequence $\{d_{1,j}(\lambda)\}$ at each bisection point. If we define

$$Q_{i,j}(\lambda) = \prod_{k=i}^{j} \begin{bmatrix} \zeta_k - \lambda & \eta_k \\ -\eta_{k-1} & 0 \end{bmatrix}, \tag{9}$$

it can be shown that

$$Q_{i,j}(\lambda) = \begin{bmatrix} d_{i,j}(\lambda) & \eta_j d_{i,j-1}(\lambda) \\ -\eta_{i-1} d_{i+1,j}(\lambda) & -\eta_{i-1}\eta_j d_{i+1,j-1}(\lambda) \end{bmatrix}. \tag{10}$$

Thus, the terms of the Sturm sequence $\{d_{1,j}(\lambda)\}$, $j = 1, \ldots, n$ are given as the upper left elements of $\{Q_{1,j}(\lambda)\}$, $j = 1, \ldots, n$ respectively. The sequence $\{Q_{1,j}(\lambda)\}_{j=1,n}$ can be evaluated in parallel by using the associative property of matrix multiplication according to the formula

$$Q_{i,j} = Q_{i,k} Q_{k+1,j}, \tag{11}$$

for any $i \leq k \leq j-1$.

Using this splitting formula, the computation can be performed in $\log n$ steps following a binary tree scheme. Figure 6 illustrates the data dependency graph for the case $n = 4$.

The parallel algorithm to compute the Sturm sequence proceeds as follows:
Step (1) - In parallel, compute:

$$Q_{i,i+1} = Q_{i,i} Q_{i+1,i+1}, \qquad i = 1, \ldots n-1, \tag{12}$$

where $Q_{i,i}$ are 2×2 matrices obtained by setting $i = j$ in equation (9).

Step (2) - In parallel, compute:

$$\begin{aligned} Q_{1,3} &= Q_{1,1} Q_{2,3} \\ Q_{1,4} &= Q_{1,2} Q_{3,4} \\ Q_{i,i+3} &= Q_{i,i+1} Q_{i+2,i+3} \qquad i = 2, \ldots n-3. \end{aligned} \tag{13}$$

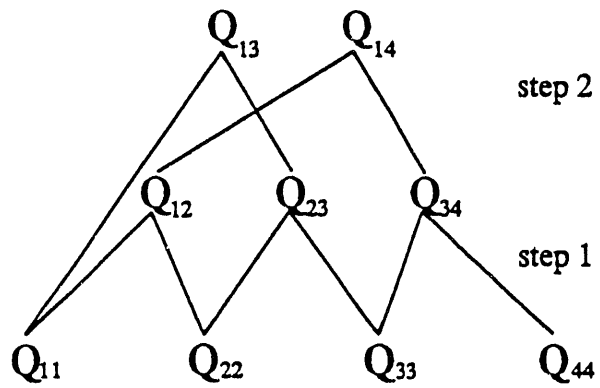

Figure 6: Data dependency graph for $n = 4$.

Step (k) - In parallel, compute:

$$\begin{aligned} Q_{1,2^{k-1}+i} &= Q_{1,i}Q_{i+1,2^{k-1}+i} \qquad i = 1,\ldots 2^{k-1} \\ Q_{i,i+2^k-1} &= Q_{i,i+2^{k-1}-1}Q_{i+2^{k-1},i+2^k-1} \qquad i = 2,\ldots n-2^k+1. \end{aligned} \tag{14}$$

Step $(\log n)$ - In parallel, compute:

$$Q_{1,2^{\log n-1}+1},\ldots,Q_{1,n}.$$

We analyze in the following subsections, the computation and communication costs of this procedure.

6.1 Computation Cost

We first analyze the arithmetic cost that each Sturm sequence evaluation incurs. By looking at equations (12–14), it can be seen that there are $n-1$ matrix multiplications in the first level of the computational tree, $n-2$ in the second and $n-2^{k-1}$ in the k^{th} level. The total number of 2×2 matrix multiplications involved in the $\log n$ levels of the computational tree is then

$$\sum_{i=0}^{\log n-1} (n-2^i) = n\log n + 1 - n.$$

Furthermore, each matrix multiplication requires 8 floating point multiplications and 4 floating point additions, except for the matrices at the first level which only take 6 multiplications and 2 additions because they have a zero element. Thus, the serial computation cost to compute the Sturm sequence at each bisection step of this algorithm is

$$(4\omega_1 + 8\omega_2) * (n\log n - n) + 2\omega_1 + 6\omega_2.$$

Assuming that we can distribute this cost evenly among the p processors, the parallel computation cost per Sturm sequence evaluation becomes roughly

$$\frac{1}{p} * (4\omega_1 + 8\omega_2) * (n\log n + 1 - n). \tag{15}$$

Adding the cost of computing the bisection points and counting the sign agreements in the Sturm sequence to (15) and considering all the steps in the bisection iteration gives the total arithmetic cost per processor for computing a single eigenvalue by Swarztrauber's parallel bisection algorithm as

$$T_{SPB} = k_1 * \left[\frac{1}{p} * (4\omega_1 + 8\omega_2) * (n \log n + 1 - n) + n\gamma_2 + \omega_1 + \omega_2\right]. \tag{16}$$

6.2 Communication Cost

As illustrated in Figure 6, it is necessary to carry out $\log n$ steps to evaluate the Sturm sequence at each bisection point. For our analysis, we will ignore contention problems and assume that each processor is involved in at most one 2×2 matrix send or receive per step. The communication cost is then

$$\log n(\beta + 4 * 8\tau), \tag{17}$$

for real*8 matrix elements. Once the Sturm sequence is evaluated, its terms reside on different processors, and it is necessary to accumulate them in order to count the number of sign agreements. This interchange can be accomplished using the ADE approach described in [11] and reviewed in section 2.2.2 of this paper.

The ADE algorithm is used to accumulate a vector whose components are evenly distributed among the processors. If the machine allows simultaneous bidirectional exchange across the same communication channel, the cost of the vector accumulation becomes

$$d\beta + 8\tau * (2^d - 1), \tag{18}$$

for real*8 vector elements.

Thus, adding expressions (17) and (18) to equation (16), the total time to compute one eigenvalue to accuracy $\epsilon/2$ by Swarztrauber's parallel bisection is

$$\begin{aligned} T_{SPB} &= \{\text{computation cost for each Sturm sequence evaluation} \\ &\quad + \text{communication cost for each Sturm sequence evaluation} \\ &\quad + \text{time to interchange the terms by ADE}\} \\ &\quad\quad * \text{number of iterations} \\ &= \{\frac{1}{p}(4\omega_1 + 8\omega_2) * (n \log n + 1 - n) + n\gamma_2 + \omega_1 + \omega_2 + \\ &\quad \log n * (\beta + 4 * 8\tau) + \\ &\quad d\beta + 8\tau * (p - 1)\} * \lceil -\log \epsilon \rceil, \end{aligned}$$

where $d = \log(p)$.

Therefore, even in the best situation when each processor sends or receives at most one 2×2 matrix per step, the communication cost for this method is higher than for the other two, i.e., multisection based on BG or ADE, for any number of processors. Because the computation cost for each Sturm sequence evaluation is divided by p, we first thought that SPB could be competitive for a large number of processors. However, this is not the case, as illustrated in figure 7 where it can be seen that multisection is more efficient than SPB even for large number of processors.

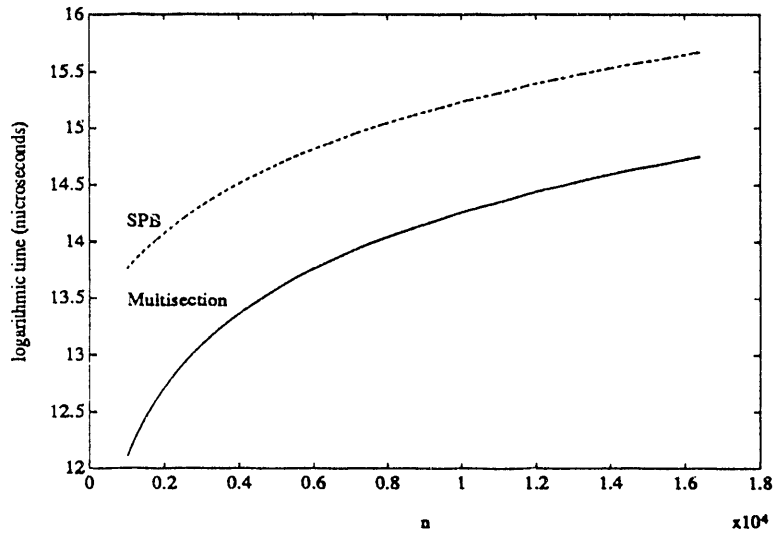

Figure 7: The cost functions T_{SPB} and T_{ADE} for $p = 1024$.

Figure 7 shows a comparison between theoretical values obtained using the functions T_{ADE}, i.e., the cost of multisection using ADE, and T_{SPB}, i.e., the cost of SPB, for $p = 1024$. The values of the parameters considered correspond to those for the iPSC/2 given in [4], and ϵ is the double precision machine error. Hence, $\beta = 390$, τ, ω_1, and ω_2 are as described in section 2.2, $\gamma_2 = 5.4$, and $-\log(\epsilon) = 54$.

The situation illustrated in figure 7 remains unchanged even for larger number of processors —i.e., $p > 1024$— due to the high communication cost incurred by this method. Therefore, the parallel Sturm sequence evaluation proposed by Swarztrauber is not a competitive method for computing one eigenvalue on the distributed-memory multiprocessors considered in this paper.

7 Summary

In this paper, we have studied and compared the costs of computing a single eigenvalue by serial bisection, parallel multisection, and Swarztrauber's parallel bisection, on different distributed-memory MIMD multiprocessors. We have shown that parallel multisection outperforms Swarztrauber's parallel bisection. We have also shown that the optimal number of processors (and sections) depends on such parameters as the ratio between communication and computation costs and the matrix size.

In general, multisection performs better than bisection on the iPSC/2 and Delta machines, and the maximum number of processors available is a good, practical approximation to the optimal number of sections to use. Bisection is the best choice only for small size matrices. An example in section 3 shows that bisection is preferred for computing an eigenvalue to double precision on the iPSC/2 when the matrix order is no greater than 27.

Acknowledgements

We thank Bill Gropp for his aid in running experiments on the Delta, Rik Littlefield for his helpful comments on an early draft of this work, and Robert van de Geijn for providing Figure 3.

References

[1] M. Barnett, D.G. Payne, and R. van de Geijn, *Optimal broadcasting in mesh-connected architectures*, Dept. of Computer Science, University of Texas, Technical Report TR-91-38, 1991.

[2] W. Barth, R.S. Martin and J.H. Wilkinson, *Calculation of the eigenvalues of a symmetric tridiagonal matrix by the method of bisection, Handbook for automatic computation: Linear Algebra*, Springer Verlag, 1971, pp. 249–256.

[3] H. Bernstein and M. Goldstein, *Optimizing Givens' algorithm for multiprocessors*, SIAM J. Sci. Stat. Comput., 9, 1988, pp. 601–602.

[4] T. Dunigan, *Performance of the Intel iPSC/860 hypercube*, Oak Ridge National Laboratory, Technical Report ORNL/TM-11491, 1990.

[5] W. Givens, *Numerical computation of the characteristic values of a real symmetric matrix*, Oak Ridge National Laboratory, Technical Report ORNL-1574, 1954.

[6] H. Huang, *A parallel algorithm for symmetric tridiagonal eigenvalue problems*, Center for Advanced Computation, University of Illinois, CAC Document 109, 1974.

[7] I. Ipsen and E. Jessup, *Solving the symmetric tridiagonal eigenvalue problem on the hypercube*, SIAM J. Sci. Stat. Comput., 11, 1990, pp. 203–229.

[8] E. Jessup, *Parallel solution of the symmetric tridiagonal eigenproblem*, Dept. of Computer Science, Yale University, Research Report 728, 1989.

[9] W. Kahan, *Accurate eigenvalue of a symmetric tridiagonal matrix*, Dept. of Computer Science, Stanford University, Technical Report CS41, 1966 (revised June 1968).

[10] S. Lo, B. Phillipe, and A. Sameh, *A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem*, SIAM J. Sci. Stat. Comput., 8, 1987, pp. 155–165.

[11] Y. Saad and M. Schultz, *Data communication in hypercubes*, Dept. of Computer Science, Yale University, Research Report 428, 1985.

[12] S. Seidel, M.H. Lee, and S. Fotedar, *Concurrent bidirectional communication on the Intel iPSC/860 and iPSC/2*, Dept. of Computer Science, Michigan Technological University, Technical Report CS-TR 90-06, 1990.

[13] H. Simon, *Bisection is not optimal on vector processors*, SIAM J. Sci. Stat. Comput., 10, 1989, pp. 205–209.

[14] B. SMITH, J. BOYLE, J. DONGARRA, B. GARBOW, Y. IKEBE, V. KLEMA, AND C. MOLER, *Matrix eigensystem routines-EISPACK Guide, Lecture Notes in Computer Science, Vol. 6, 2nd edition*, Springer-Verlag, 1976.

[15] P. SWARZTRAUBER, *A parallel algorithm for computing the eigenvalues of a symmetric tridiagonal matrix*, unpublished manuscript, 1991.

END

DATE
FILMED
9/30/93