Electronic Transactions on Numerical Analysis.
Volume 3, pp. 116-149, December 1995.
Copyright © 1995, Kent State University.
ISSN 1068-9613.

ETNA
Kent State University
etna@mcs.kent.edu

ON THE CORRECTNESS OF SOME BISECTION-LIKE PARALLEL EIGENVALUE ALGORITHMS IN FLOATING POINT ARITHMETIC*

JAMES W. DEMMEL †, INDERJIT DHILLON‡, AND HUAN REN§

Abstract. Bisection is a parallelizable method for finding the eigenvalues of real symmetric tridiagonal matrices, or more generally symmetric acyclic matrices.

Ideally, one would like an implementation that was simultaneously parallel, load balanced, devoid of communication, capable of running on networks of heterogenous workstations, and of course correct. But this is surprisingly difficult to achieve.

The reason is that bisection requires a function Count(x) which counts the number of eigenvalues less than x. In exact arithmetic Count(x) is a monotonic increasing function of x, and the logic of the algorithm depends on this. However, monotonicity can fail, and incorrect eigenvalues may be computed, because of roundoff or as a result of using networks of heterogeneous parallel processors. We illustrate this problem, which even arises in some serial algorithms like the EISPACK routine `bisect`, and show several ways to fix it. One of these ways has been incorporated into the ScaLAPACK library.

Key words. symmetric eigenvalue problem, parallel algorithms, monotonicity, correctness, floating point.

AMS subject classifications. 65F15, 65Y05.

1. Introduction.

Let T be an n-by-n real symmetric tridiagonal matrix with diagonals $a_1, ..., a_n$ and offdiagonals $b_1, ..., b_{n-1}$; we let $b_0 \equiv 0$. Let $\lambda_1 \le \cdots \le \lambda_n$ be T's eigenvalues. It is known that the function Count(x) defined below returns the number of eigenvalues of T that are less than x (for all but the finite number of x resulting in a divide by zero, which are called singular points) :

Algorithm 1: Count(x) **returns the number of eigenvalues of a real symmetric tridiagonal matrix T that are less than x.**

```
Count = 0;
$d = 1$;
for $i = 1$ to $n$
    $d = a_i - x - b_{i-1}^2/d$
    if $d < 0$ then Count = Count + 1
endfor
```

---

* Received September 26, 1995. Accepted for publication December 7, 1995. Communicated by J. J. Dongarra.

† Computer Science Division and Department of Mathematics, University of California, Berkeley, California 94720. This material is based in part upon work supported by the Advanced Research Projects Agency contract No. DAAL03-91-C-0047 (via subcontract No. ORA4466.02 with the University of Tennessee), the Department of Energy grant No. DE-FG03-94ER25219, and contract No. W-31-109-Eng-38 (via subcontract Nos. 20552402 and 941322401 with Argonne National Laboratory), the National Science Foundation grant Nos. ASC-9313958, ASC-9005933, CCR-9196022, and NSF Infrastructure Grant Nos. CDA-8722788 and CDA-9401156. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

‡ Computer Science Division, University of California, Berkeley, California 94720. The author was supported by DARPA grant DAAL03-91-C-0047 via a subcontract from the University of Tennessee.

§ Department of Mathematics, University of California, Berkeley, California 94720. The author was supported by DARPA grant DAAL03-91-C-0047 via a subcontract from the University of Tennessee.

(Algorithm 1 can be derived by doing Gaussian Elimination to get $T - xI = LDL^T$, counting the number of negative diagonal entries of D, and applying Sylvester's Law of Inertia [22]. If we wish to emphasize that T is the argument, we will write $\mathrm{Count}(x, T)$ instead.)

We define $\mathrm{Count}(x)$ at singular points so that it is left continuous. This means that for all x, $\mathrm{Count}(x)$ is the number of eigenvalues less than x.

It is easy to see that the number of eigenvalues in the half-open interval $[\sigma_1, \sigma_2)$ is $\mathrm{Count}(\sigma_2) - \mathrm{Count}(\sigma_1)$. This observation may be used as the basis for a "bracketing" algorithm to find all the eigenvalues of T, or just those in an interval $[\sigma_1, \sigma_2)$ or $[\lambda_j, \lambda_k)$. A bracketing algorithm refers to any algorithm which involves dividing an interval containing at least one eigenvalue into smaller subintervals of any size, and recomputing the numbers of eigenvalues in the subintervals. The algorithm terminates when the intervals are narrow enough. Bisection is one such bracketing algorithm where each interval is divided into two subintervals of equal size.

The logic of such a bracketing algorithm would seem to depend on the simple fact that $\mathrm{Count}(x)$ is a monotonic increasing step function of x. If its computer implementation, call it $\mathrm{FloatingCount}(x)$, were not also monotonic, so that one could find $\sigma_1 < \sigma_2$ with $\mathrm{FloatingCount}(\sigma_1) > \mathrm{FloatingCount}(\sigma_2)$, then the computer implementation might well report that the interval $[\sigma_1, \sigma_2)$ contains a negative number of eigenvalues, namely $\mathrm{FloatingCount}(\sigma_2) - \mathrm{FloatingCount}(\sigma_1)$. This result is clearly incorrect. In section 4.1 below, we will see that this can indeed occur using the EISPACK routine `bisect` (using IEEE floating point standard arithmetic [2, 3], and without over/underflow).

This paper explains how to design correct bracketing algorithms in the face of nonmonotonicity. There are at least four reasons why $\mathrm{FloatingCount}(x)$ might not be monotonic:

1. the floating point arithmetic is too inaccurate,
2. over/underflow occurs, or is avoided improperly,
3. $\mathrm{FloatingCount}(x)$ is implemented using a fast parallel algorithm called parallel prefix, or
4. heterogeneity — processors in a parallel environment may have differing floating point arithmetics, or may just compile code slightly differently.

Ideally, we would like a bracketing algorithm that was simultaneously parallel, load balanced, devoid of communication, and correct in the face of nonmonotonicity. We still do not know how to achieve this completely; in the most general case, when different parallel processors do not even possess the same floating point format, we do not know how to implement a correct and reasonably fast algorithm at all. Even when floating point formats are the same, we do not know how to avoid some global communication. Still, we have discovered several efficient and correct implementations, one of which (see PAR_ALLEIG4 below) we have incorporated in the ScaLAPACK [8] library routine for the symmetric tridiagonal eigenproblem.

A major goal of this paper is to explain and justify the algorithm we have chosen for use in ScaLAPACK. Many of the apparently arcane failure modes discussed in this paper have arisen in practice. We present a number of other interesting results along the way.

Our first result is to give examples showing how monotonicity can fail, and cause incorrect eigenvalues to be computed, for all the reasons listed above. See sections 4 and 6.

Our second result is to show that as long as the floating point arithmetic is monotonic (we define this in section 2.1), and FloatingCount(x) is implemented on a single processor in a reasonable way, then FloatingCount(x) is also monotonic. A sufficient condition for floating point to be monotonic is that it be correctly rounded or correctly chopped; thus IEEE floating point arithmetic is monotonic. This result was first proven but not published by Kahan in 1966 for symmetric tridiagonal matrices [18]; in this paper we extend this result to *symmetric acyclic matrices*, a larger class including tridiagonal matrices, arrow matrices, and exponentially many others [10]; see section 6.

Our third result is to formalize the notion of a correct implementation of a bracketing algorithm, and use this characterization to identify correct and incorrect serial and parallel implementations of such algorithms. We illustrate with several simple, natural but wrong implementations of parallel bracketing algorithms, and show how to implement them correctly in the absence of monotonicity; See sections 4 and 7. Nonmonotonic implementations of FloatingCount(x) remain of interest, even though nonmonotonic arithmetics are a dying breed, because the fastest known parallel prefix implementations of FloatingCount(x) appear unavoidably nonmonotonic. Also, networks of heterogeneous processors, or even just machines that compile code differently, lead to issues that are similar to those arising due to a nonmonotonic FloatingCount(x).

We feel this paper is also of interest because it is an example of a rigorous correctness proof of an algorithm using floating point arithmetic. We make clear exactly which properties of floating point are necessary to prove correctness.

The rest of this paper is organized as follows. Section 2 gives the definitions and assumptions. Section 3 gives tables to summarize the results of this paper and the assumptions needed to prove the results. Section 4 gives some examples of incorrect bracketing algorithms, and also gives some serial and parallel algorithms that are provably correct subject to some assumptions about the computer arithmetic and FloatingCount(x). Section 5 reviews the roundoff error analysis of FloatingCount(x), and how to account for over/underflow; this material may also be found in [10, 18]. Section 6 illustrates how monotonicity can fail, and proves that a natural serial implementation of FloatingCount(x) must be monotonic if the arithmetic is. Section 7 gives formal proofs for the correctness of the bracketing algorithms given in section 4. Section 8 discusses some practical implementation issues and Section 9 concludes the paper.

2. Definitions and Assumptions.

Section 2.1 defines the kinds of matrices whose eigenvalue problems we will consider, what monotonic arithmetic is, and what "jump points" of the functions Count(x) and FloatingCount(x) are. Section 2.2 presents our (mild) assumptions about floating point arithmetic, the input matrices our algorithms will accept and the way the next iterate in the enclosing interval may be chosen. Section 2.3 lists the criteria a bracketing algorithm must satisfy to be correct.

2.1. Preliminary Definitions.

Algorithm 1 was recently extended to the larger class of *symmetric acyclic matrices* [10], i.e. those matrices whose graphs are acyclic (trees). The undirected graph $G(T)$ of a symmetric n-by-n matrix T is defined to have n nodes and an edge (i, j), $i < j$, if and only if $T_{ij} \neq 0$. A symmetric tridiagonal matrix is one example of a symmetric acyclic matrix; its graph is a chain. An "arrow matrix" which is nonzero only on the diagonal, in the last row and in the last column, is another example; its graph is a star. From now on, we will assume T is a symmetric acyclic matrix unless we state explicitly otherwise. Also we will number the rows and columns of T in preorder such that node 1 is the root of the tree and so accessed first; node j is called a *child* of node i if $T_{ij} \neq 0$ and node j has not yet been visited by the algorithm (see Algorithm 6 in section 6, TreeCount, for details). We let C denote the maximum number of children of any node in the acyclic graph $G(T)$(C is never larger than the degree of $G(T)$).

To describe the monotonicity of FloatingCount(x), we need to define *monotonic arithmetic*: An implementation of floating point arithmetic is monotonic if, whenever a, b, c and d are floating point numbers, $\otimes$ is any binary operation, and the floating point results $fl(a \otimes b)$ and $fl(c \otimes d)$ do not overflow, then $a \otimes b \geq c \otimes d$ implies $fl(a \otimes b) \geq fl(c \otimes d)$. This is satisfied by any arithmetic that rounds or truncates correctly. In section 6, we will prove that the FloatingCount function (Floating TreeCount) for a symmetric acyclic matrix is monotonic if the floating point arithmetic is monotonic.

We now define a *jump-point* of the function Count(x). λ_i is the i^{th} jump-point of the function Count(x) if

$$\lim_{x \to \lambda_i^-} \mathrm{Count}(x) < i \leq \lim_{x \to \lambda_i^+} \mathrm{Count}(x)$$

Note that if $\lambda_i = \lambda_j$, then λ_i is simultaneously the i^{th} and j^{th} jump point. Analogous to the above definition, we define an i^{th} jump-point of a possibly nonmonotonic function FloatingCount(x) as a floating point number λ_i'' such that

$$\mathrm{FloatingCount}(\lambda_i'') < i \leq \mathrm{FloatingCount}(\mathit{nextafter}(\lambda_i''))$$

where *nextafter*(λ_i'') is the smallest floating point number greater than λ_i''. For a nonmonotonic FloatingCount(x) function, there may be more than one such jump-point.

2.2. Assumptions. In order to prove correctness of our algorithms, we need to make some assumptions about the computer arithmetic, the inputs, the bracketing algorithm and the FloatingCount function. The following is a list of all the assumptions we will make; not all our results require all the assumptions, so we must be explicit about which assumptions we need.

The first set of assumptions, Assumption 1, concerns the floating point arithmetic. Not all parts of Assumption 1 are necessary for all later results, so we will later refer to Assumptions 1A, 1B, etc. Assumption 2 is about the input matrix, and includes a mild restriction on its size, and an easily enforceable assumption on its scaling. Assumption 3 is about the choice of the iteration point in an interval; it is also easily enforced. Assumption 4 consists of two statements about the implementation of the FloatingCount function, which can be proven for most current implementations provided appropriate parts of Assumptions 1 and 2 are true (see section 5). We still call these two statements assumptions, rather than theorems, because they are the most convenient building blocks for the ultimate correctness proofs.

Assumption 1 (Properties of Floating Point Arithmetic).

1A. Barring overflow, the usual expression for roundoff is extended to include underflow as follows [9]:

$$fl(a \otimes b) = (a \otimes b)(1 + \delta) + \eta \tag{2.1}$$

where $\otimes$ is a binary arithmetic operation, $|\delta|$ is bounded by machine precision ε, $|\eta|$ is bounded by a tiny number $\bar{\omega}$, typically the underflow threshold ω (the smallest normalized number which can safely participate in, or be a result of, any floating point operation)[1], and at most one of δ and η can be nonzero. In IEEE arithmetic, gradual underflow lets us further assert that $\bar{\omega} = \varepsilon\omega$, and that if $\otimes$ is addition or subtraction, then η must be zero. We denote the overflow threshold of the computer (the largest number which can safely participate in, or be a result of, any floating point operation) by Ω.

In this paper, we will consider the following three variations on this basic floating point arithmetic model:

Model 1. $fl(a \otimes b) = (a \otimes b)(1+\delta) + \eta$ as above, and overflows terminate, i.e. the machine stops executing the running program.

Model 2. IEEE arithmetic with $\pm\infty$, ± 0 and NaN, and with gradual underflow.

Model 3. IEEE arithmetic with $\pm\infty$, ± 0 and NaN, but with underflow flushing to zero.

1B. $\sqrt{\omega} \le \varepsilon \le 1 \le 1/\varepsilon \le \sqrt{\Omega}$. This mild assumption is satisfied by all commercial floating point arithmetics.

1C. Floating point arithmetic is monotonic. This is true of IEEE arithmetic (Models 2 and 3) but may not be true of Model 1.

1D. All processors have essentially identical floating point formats so that **send** and **receive** operations between processors just copy bits. Processors that follow the IEEE format but have different byte orderings (big endian and little endian) for a floating point number satisfy this assumption.

1E. Different processors have *identical* floating point arithmetic so that the result of the same floating point operation is bitwise identical on all processors.

ASSUMPTION 2 (PROPERTIES OF THE INPUT MATRIX).

2A. Assumption on the problem size n: $n\varepsilon \le 0.1$. Virtually all numerical algorithms share a restriction like this.

2B. Assumptions on the scaling of the input matrix. Let $\bar{B} \equiv \min_{i \ne j} T_{ij}^2$ and $\bar{M} \equiv \max_{i,j} |T_{ij}|$.

1. $\bar{B} \ge \omega$.
2. $\bar{M} \le \sqrt{\Omega}$.

These assumptions may be achieved by explicitly scaling the input matrix (multiplying it by an appropriate scalar), and then setting small off-diagonal elements $T_{ij}^2 < \omega$ to zero and so splitting the matrix into *unreduced* blocks [4]; see section 5.8 for details. By Weyl's Theorem [22], this may introduce a tiny error of amount no more than $\sqrt{\omega}$ in the computed eigenvalues.

2C. More assumptions on the scaling of the input matrix. These are used to get refined error bounds in section 5.

1. $\bar{M} \ge \omega/\varepsilon$.
2. $\bar{M} \ge 1/(\varepsilon\Omega)$.

ASSUMPTION 3 (HOW TO SPLIT AN INTERVAL).

All the algorithms we will consider try to bound an eigenvalue within an interval. A fundamental operation in these algorithms is to compute a point which lies in an interval (α, β) — we denote this point by $inside(\alpha, \beta, T)$. This point may be computed by simply bisecting the interval or by applying Newton's or Laguerre's iteration. *We assume that $fl(inside(\alpha, \beta, T)) \in (\alpha, \beta)$, for all attainable values of α and β.* For

---

[1] These caveats about "safe participation in *any* floating point operation" take machines like some Crays into account, since they have "partial overflow".

example, if we bisect the interval, $inside(\alpha,\beta,T) = \frac{\alpha+\beta}{2}$ and we will assume that $fl(\frac{\alpha+\beta}{2}) \in (\alpha,\beta)$, for all $\alpha < \beta$ such that $\beta - \alpha > \max(|\alpha|,|\beta|)\epsilon$ (i.e., there is at least one floating point number in (α,β)), where ϵ is the machine precision. Barring overflow, this assumption always holds in IEEE arithmetic, and for any model of binary arithmetic which rounds correctly, i.e., rounds a result to the nearest floating point number. For a detailed treatment on how to compute $\frac{\alpha+\beta}{2}$ correctly on various machines, see [19].

An easy way to enforce this assumption given an arbitrary $inside(\alpha,\beta,T)$ is to replace it by

$$\min(\max(inside(\alpha,\beta,T), nextafter(\alpha)), nextbefore(\beta))),$$

where $nextafter(\alpha)$ is the smallest floating point number greater than α, and $nextbefore(\beta)$ is the greatest floating point number smaller than β.

Assumption 4 (Correctness Properties of FloatingCount).

4A. FloatingCount(x) does not abort.

4B. Let $\lambda_i^{(1)''}, \lambda_i^{(2)''}, \ldots, \lambda_i^{(k)''}$ be the i^{th} jump-points of FloatingCount(x). We assume that FloatingCount(x) satisfies the error bound,

$$|\lambda_i^{(j)''} - \lambda_i| \le \xi_i, \ \ \forall j = 1, \ldots, k$$

for some $\xi_i \ge 0$. We have assumed that FloatingCount(x) has a bounded region of possible nonmonotonicity, and ξ_i is a bound on the nonmonotonicity around eigenvalue λ_i. Different implementations of FloatingCount(x) result in different values of ξ_i (see section 5).

Differing floating point arithmetic may result in ξ_i being different on different processors. In such a case, we define $\xi_i = \max_{j=0}^{p-1} \xi_i^{(j)}$, where $\xi_i^{(j)}$ is the corresponding bound on the nonmonotonicity on processor j.

For some of the practical FloatingCount functions in use, we will prove Assumption 4 in section 5.

2.3. When is a Bracketing Algorithm Correct? We now describe the functional behaviour required of any correct implementation of a bracketing algorithm. Let τ_i be a user-supplied upper bound on the desired error in the i^{th} computed eigenvalue; this means the user wants the computed eigenvalue λ_i' to differ from the true eigenvalue λ_i by no more than τ_i. Note that not all values of τ_i are attainable, and the attainable values of τ_i depend on the FloatingCount function, the underlying computer arithmetic and the input matrix T. For example, when using Algorithm 1, τ_i can range from $\max_i |\lambda_i|$ down to $O(\varepsilon)\max_i |\lambda_i|$, or perhaps smaller for special matrices [5]. Let U be a non-empty set of indices that correspond to the eigenvalues that the user wants to compute, e.g., $U = \{1, \ldots, n\}$ if the user wants to compute all the eigenvalues of a matrix of dimension n. The output of the algorithm should be a sorted list of the computed eigenvalues, i.e. a list (i, λ_i') where each $i \in U$ occurs exactly once, and $\lambda_i' \le \lambda_j'$ when $i \le j$. In a parallel implementation, this list may be distributed over the processors in any way at the end of the computation. Thus the algorithm must compute the eigenvalues and also present the output in sorted order. Beyond neatness, the reason that we require the eigenvalues to be returned in sorted order is that it facilitates subsequent uses that require scanning through the eigenvalues in order, such as reorthogonalization of eigenvectors in inverse iteration.

In summary, when we say that an implementation of a bracketing algorithm is *correct*, we assert that it terminates and all of the following hold:

- Every desired eigenvalue is computed exactly once.
- The computed eigenvalues are correct to within the user specified error tolerance, i.e. for all desired $i > 0$, $|\lambda_i - \lambda'_i| \leq \tau_i + 2\xi_i$ (in case $\tau_i > 2\xi_i$, the implementation guarantees that $|\lambda_i - \lambda'_i| \leq 2\tau_i$). See section 2.2, Assumption 4B for a definition of ξ_i.
- The computed eigenvalues are in sorted order.

We say that an implementation of a bracketing algorithm is *incorrect* when any of the above fails to hold.

3. Outline of the Paper. In this section, we outline all our results in four tables. Tables 3.1 and 3.2 describe the algorithms, and Tables 3.3 and 3.4 describe their properties. Table 3.2 lists the six bracketing algorithms we consider, both serial and parallel. Each such algorithm uses a version of FloatingCount internally; the possible versions are listed in Table 3.1. Correctness of a bracketing algorithm depends on whether FloatingCount has the correctness properties in Assumption 4, and possibly also whether it is monotonic. For each version of FloatingCount, Table 3.3 lists which parts of Assumptions 1–3 are needed for Assumption 4, and possibly monotonicity, to hold. Finally, Table 3.4 summarizes by listing exactly which assumptions are needed for each bracketing algorithm in Table 3.2 to be correct; note that each bracketing algorithm may work with one or more versions of FloatingCount, depending on which assumptions hold.

The reason we consider six bracketing algorithms is to illustrate the complexities necessitated by dealing with parallelism, nonmonotonic floating point, and heterogeneous floating point. All these complexities arose naturally in the course of implementing ScaLAPACK [8]. For example, Algorithm PAR_ALLEIG1, perhaps the simplest and most natural parallel algorithm, cannot correctly use the EISPACK algorithm FlCnt_bisect because of its nonmonotonicity. The current ScaLAPACK implementation uses Algorithm PAR_ALLEIG4. Thus, it depends on Assumption 1D, i.e., all processors have the same floating point format.

TABLE 3.1
Different implementations of FloatingCount

Algorithms	Description	Where
FlCnt_bisect	algorithm used in EISPACK's `bisect` routine; most floating point exceptions avoided by tests and branches	See section 5.2 and [23]
FlCnt_IEEE	IEEE standard floating point arithmetic used to accommodate possible exceptions; tridiagonals only	See section 5.3 and [4, 18]
FlCnt_stebz	algorithm used in LAPACK's `dstebz` routine; floating point exceptions avoided by tests and branches	See section 5.4 and [1]
FlCnt_Best_Scaling	like FlCnt_stebz, but prescales for optimal error bounds	See section 5.5 and [4, 18]

The following sections prove the results outlined in the above tables. Sections 4, 5 and 6 may be read independently of each other although parts of sections 5 and 6 justify the assumptions we make in the various algorithms given in section 4.

TABLE 3.2
Different implementations of Bracketing algorithms

Algorithms	Description	Where
SER_BRACKET	Serial bracketing algorithm that finds all the eigenvalues of T in a user-specified interval	See section 4.3
SER_ALLEIG	Serial bracketing algorithm that finds all the eigenvalues of T	See section 4.3
PAR_ALLEIG1	Parallel bracketing algorithm that finds all the eigenvalues of T, load distributed by equally dividing the Gerschgorin interval into p equal subintervals; needs monotonic FloatingCount; incorrect on heterogeneous parallel machines	See section 4.4
PAR_ALLEIG2	Similar to PAR_ALLEIG1, but monotonic FloatingCount not needed; correct on heterogeneous parallel machines	See section 4.4
PAR_ALLEIG3	Parallel bracketing algorithm that finds all the eigenvalues of T, load balanced by making each processor find an equal number of eigenvalues; monotonic FloatingCount not needed; incorrect on heterogeneous parallel machines	See section 4.4
PAR_ALLEIG4	Similar to PAR_ALLEIG3, but correct on heterogeneous parallel machines	See section 4.4

TABLE 3.3
Results of Roundoff Error Analysis and Monotonicity

Assumptions about Arithmetic and Input Matrix	Results	Proofs
T is symmetric tridiagonal $\wedge$ (1A(Model 2) $\vee$ 1A(Model 3)) $\wedge$ 1B $\wedge$ 2A $\wedge$ 2B(ii)	For FlCnt_bisect, Assumption 4 holds but FloatingCount(x) can be nonmonotonic	See section 4.3 and section 5.2
T is symmetric tridiagonal $\wedge$ (1A(Model 2) $\vee$ 1A(Model 3)) $\wedge$ 1B $\wedge$2A $\wedge$ 2B	For FlCnt_IEEE, Assumption 4 holds and FloatingCount(x) is monotonic	See section 5.3 and section 6
T is symmetric acyclic $\wedge$ 1A $\wedge$ 1B $\wedge$ 1C $\wedge$ 2A $\wedge$ 2B(ii)	For FlCnt_stebz, Assumption 4 holds and FloatingCount(x) is monotonic	See section 5.4 and section 6
T is symmetric acyclic $\wedge$ 1A $\wedge$ 1B $\wedge$ 1C $\wedge$ 2A	For FlCnt_Best_Scaling, Assumption 4 holds and FloatingCount(x) is monotonic	See section 5.5 and section 6

4. Correctness of Bracketing Algorithms.

We now investigate the correctness of bracketing algorithms and exhibit some existing "natural" implementations of bracketing that are incorrect. Table 3.4 summarizes all the correctness results that we prove in this paper.

In the rest of the paper, whenever we refer to a variable x used in an algorithm, we will mean it to be the computed value of x. Also, we will frequently refer to the value of the variable x on processor i as $x^{(i)}$.

4.1. An Incorrect Serial Implementation of Bisection.

We give an example of the failure of EISPACK's `bisect` routine in the face of a nonmonotonic FloatingCount(x). Suppose we use IEEE standard double precision floating point arithmetic with $\varepsilon = 2^{-53} \approx 1.1 \cdot 10^{-16}$ and we want to find the eigenvalues of the following 2×2 matrix:

$$A = \begin{pmatrix} 0 & 2\varepsilon \\ 2\varepsilon & 1 \end{pmatrix}$$

A has eigenvalues near 1 and $-4\varepsilon^2 \approx -4.93 \cdot 10^{-32}$. But `bisect` reports that the interval $[-10^{-32}, 0)$ contains -1 eigenvalues. No overflow or underflow occurs in this

Table 3.4
Correctness Results

Assumptions	Results	Proofs
1A, 3, 4	Algorithm Ser_Bracket is correct	See section 7
1A, 3, 4	Algorithm Ser_AllEig is correct	See section 4.3
1A, 1E, 3, 4, and FloatingCount(x) is monotonic	Algorithm Par_AllEig1 is correct	See section 7.2
1A, 1D, 3, 4	Algorithm Par_AllEig2 is correct	See section 7.2
1A, 1E, 3, 4	Algorithm Par_AllEig3 is correct	See section 7.2
1A, 1D, 3, 4	Algorithm Par_AllEig4 is correct	See section 7.2

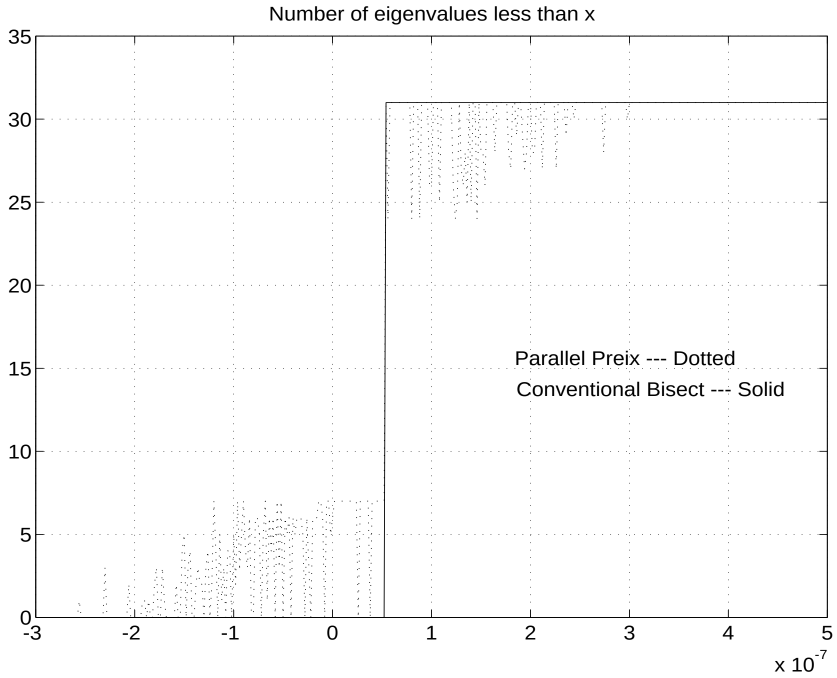

Fig. 4.1. *Parallel Prefix vs Conventional Bisection*

case. The reason for this is `bisect`'s incorrect provision against division by zero (See Algorithm 3 in section 5.2). In section 6, by the proof of Theorem 6.1, we will show that this cannot happen for the LAPACK routine `dstebz` even for general symmetric acyclic matrices.

4.2. Nonmonotonicity of Parallel Prefix Algorithm. We now give another example of a nonmonotonic FloatingCount(x) when Count(x) is implemented using a fast parallel algorithm called parallel prefix [11]. Figure 4.1 shows the FloatingCount(x) for a 64×64 matrix of norm near 1 with 32 eigenvalues very close to $5 \cdot 10^{-8}$ computed both by conventional bisection (FlCnt_IEEE) and the parallel prefix algorithm in the neighborhood of the eigenvalues; see [21, 14] for details.

4.3. A Correct Serial Implementation of the Bracketing Algorithm. As we saw in section 4.1, the Eispack implementation of the bisection algorithm fails in the face of a nonmonotonic FloatingCount. We now present an implementation which works correctly even if FloatingCount is nonmonotonic.

```
    subroutine SER_BRACKET($n$,$T$,$left$,$right$,$n_{left}$,$n_{right}$,$\tau$) /* computes the eigenvalues
            of $T$ in the interval $[left, right]$ to the desired accuracy $\tau$ given the initial
            task $(left, right, n_\alpha, n_\beta, I_{n_{left}}^{n_{right}})$ */
1:      if ($n_{left} \geq n_{right}$ or $left > right$) return;
2:      enqueue $(left, right, n_{left}, n_{right}, I_{n_{left}}^{n_{right}})$ to $Worklist$;
3:      while ($Worklist$ is not empty)
4:          dequeue $(\alpha, \beta, n_\alpha, n_\beta, I_{n_\alpha}^{n_\beta})$ from $Worklist$;
5:          if ($\beta - \alpha < \min(\min_{i=n_\alpha+1}^{n_\beta} \tau_i, \max(|\alpha|, |\beta|)\varepsilon)$) then
6:              print "Eigenvalue $\min(\max((\alpha+\beta)/2, \alpha), \beta)$ has multiplicity $n_\beta - n_\alpha$";
7:          else
8:              $mid = inside(\alpha, \beta, T)$;
9:              $n_{mid} = \min(\max(\text{FloatingCount}(mid), n_\alpha), n_\beta)$;
10:             if ($n_{mid} > n_\alpha$) then
11:                 enqueue $(\alpha, mid, n_\alpha, n_{mid}, I_{n_\alpha}^{n_{mid}})$ to $Worklist$;
12:             end if
13:             if ($n_{mid} < n_\beta$) then
14:                 enqueue $(mid, \beta, n_{mid}, n_\beta, I_{n_{mid}}^{n_\beta})$ to $Worklist$;
15:             end if
16:         end if
17:     end while
18: end subroutine
```

FIG. 4.2. **Algorithm** SER_BRACKET

We define a *task* to be a 5-tuple $T = (\alpha, \beta, n_\alpha, n_\beta, O)$, where $[\alpha, \beta]$ is a non-empty interval, n_α and n_β are counts associated with α and β respectively, and O is the set of indices corresponding to the eigenvalues being searched for in this interval. We require $O \subseteq I_{n_\alpha}^{n_\beta}$, where $I_{n_\alpha}^{n_\beta} = \{n_\alpha + 1, \ldots, n_\beta\}$ ($I_{n_\alpha}^{n_\beta} = \emptyset$ when $n_\alpha \geq n_\beta$). We do not insist that $n_\alpha = \text{FloatingCount}(\alpha)$, and $n_\beta = \text{FloatingCount}(\beta)$, only that $\exists\ x_l \leq \beta, x_u \geq \alpha$ such that $\text{FloatingCount}(x_u) \leq n_\alpha$ and $n_\beta \leq \text{FloatingCount}(x_l)$ if $I_{n_\alpha}^{n_\beta} \neq \emptyset$. In most implementations $O = I_{n_\alpha}^{n_\beta}$ and the index set O is not explicitly maintained by the implementation. A task that satisfies the above conditions is said to be *valid*.

Algorithm SER_BRACKET (see figure 4.2) is a correct serial implementation of a bracketing algorithm that finds all the eigenvalues specified by the given initial task ($left$,$right$,n_{left},n_{right}, $I_{n_{left}}^{n_{right}}$), the i^{th} eigenvalue being found to the desired accuracy τ_i (note that we allow different tolerances to be specified for different eigenvalues, τ_i being the i^{th} component of the input tolerance vector τ). The difference between this implementation and the EISPACK implementation is the initial check to see if $n_{left} \geq n_{right}$, and the *forcing* of monotonicity on FloatingCount(x) by executing statement 9 at each iteration. Statement 9 has no effect if FloatingCount(x) is monotonic, whereas it forces n_{mid} to lie between n_α and n_β if FloatingCount(x) is nonmonotonic.

THEOREM 4.1. *Algorithm* SER_BRACKET *is correct if the input task is valid and Assumptions 1A, 3 and 4 hold.*

Proof. The theorem is proved in section 7. □

Note that in all our theorems about the correctness of various bracketing algorithms, unless stated otherwise, we do not require FloatingCount(x) to be monotonic.

Algorithm SER_ALLEIG (see figure 4.3) is designed to compute all the eigenvalues

```
    subroutine Ser_Alleig($n$,$T$,$\tau$) /* computes all the eigenvalues of $T$ */
        $(gl, gu)$ = Compute_Gerschgorin($n$,$T$);
        call Ser_Bracket($n$,$T$,$gl$,$gu$,0,$n$,$\tau$);
    end subroutine
    function Compute_Gerschgorin($n$,$T$) /* returns the Gerschgorin Interval [$gl$,$gu$] */
1:      $gl = \min_{i=1}^{n}(T_{ii} - \sum_{j \neq i} |T_{ij}|)$;    /* Gerschgorin left bound */
2:      $gu = \max_{i=1}^{n}(T_{ii} + \sum_{j \neq i} |T_{ij}|)$;    /* Gerschgorin right bound */
3:      $bnorm = \max(|gl|, |gu|)$;
4:      $gl = gl - bnorm * 2n\varepsilon - \xi_0$; $gu = gu + bnorm * 2n\varepsilon + \xi_n$; /* see Table 5.5 */
5:      $gu = \max(gl, gu)$;
6:      return($gl$,$gu$);
    end function
```

FIG. 4.3. **Algorithm** Ser_Alleig *computes all the eigenvalues of* T

of T to the desired accuracy. Compute_Gerschgorin is a subroutine that computes the Gerschgorin interval $[gl, gu]$ guaranteed to contain all the eigenvalues of the symmetric acyclic matrix T. Note that in line 4 of the pseudocode, the Gerschgorin interval is widened to ensure that FloatingCount$(gl) = 0$ and FloatingCount$(gu) = n$ when Assumptions 1A and 1C hold (this is proved in section 5.7). Due to the correctness of Algorithm Ser_Bracket, we have the following corollary :

COROLLARY 4.2. *Algorithm* Ser_Alleig *is correct if Assumptions 1A, 1C, 3 and 4 hold.*

4.4. Parallel Implementation of the Bracketing Algorithm.

We now discuss parallel implementations of the bracketing algorithm. The bisection algorithm offers ample opportunities for parallelism, and many parallel implementations exist [6, 15, 20, 16]. We first discuss a natural parallel implementation which gives the false appearance of being *correct*. Then, we give a *correct* parallel implementation that has been tested extensively on many parallel platforms, such as the CM-5, Intel Paragon, IBM SP2 and various PVM platforms, and is a part of ScaLAPACK [8].

In the following sections, all the algorithms are presented in the SPMD (Single Program Multiple Data Stream) style of programming.

4.4.1. A Simple, Natural and Incorrect Parallel Implementation.

A natural way to divide the work arising in the bracketing algorithm among p processors is to partition the initial Gerschgorin interval into p equal subintervals, and assign to processor i the task of finding all the eigenvalues in the i^{th} subinterval. Algorithm Par_Alleig0 (see figure 4.4) is a simple and natural parallel implementation based on this idea that attempts to find all the eigenvalues of T (the p processors are assumed to be numbered $0, 1, 2, \ldots, p-1$). However, algorithm Par_Alleig0 fails to find the eigenvalue of a 1×1 identity matrix when implemented on a 32 processor CM-5! The reason is that when $p = 32$, *average_width* is so small that $\alpha = \beta$ on all the processors. Thus, none of the processors are able to find the only eigenvalue. (This would happen on any machine with IEEE arithmetic, not just the CM-5.)

The error in algorithm Par_Alleig0 is in the way β is computed. Algorithm Par_Alleig1 (see figure 4.5) fixes the problem by computing $\beta^{(i)}$ as $gl + (i+1) *$ *average_width*. This results in the following theorem, which we prove in section 7.2.

```
subroutine PAR_ALLEIG0($n$,$T$,$\tau$) /* computes all the eigenvalues of $T$ in parallel */
    $i$ = MY_PROCESSOR_NUMBER();
    $(gl, gu)$ = COMPUTE_GERSCHGORIN($n$,$T$);
    $average\_width = (gu - gl)/p$;
    $\alpha = gl + i * average\_width$;
    $\beta = \alpha + average\_width$;
    $n_\alpha$ = FloatingCount($\alpha$);
    $n_\beta$ = FloatingCount($\beta$);
    call SER_BRACKET($n$,$T$,$\alpha$,$\beta$,$n_\alpha$,$n_\beta$,$\tau$);
end subroutine
```

FIG. 4.4. **Algorithm** PAR_ALLEIG0 — *An* Incorrect *Parallel Implementation*

```
subroutine PAR_ALLEIG1($n$,$T$,$\tau$) /* computes all the eigenvalues of $T$ in parallel */
    $i$ = MY_PROCESSOR_NUMBER();
    $(gl, gu)$ = COMPUTE_GERSCHGORIN($n$,$T$);
    $average\_width = (gu - gl)/p$;
    $\alpha = gl + i * average\_width$;
    $\beta = \max(gl + (i+1) * average\_width, \alpha)$;
    if $(i = p - 1)$ $\beta = gu$;
    $n_\alpha$ = FloatingCount($\alpha$);
    $n_\beta$ = FloatingCount($\beta$);
    call SER_BRACKET($n$,$T$,$\alpha$,$\beta$,$n_\alpha$,$n_\beta$,$\tau$);
end subroutine
```

FIG. 4.5. **Algorithm** PAR_ALLEIG1 — Correct *when* FloatingCount(x) *is monotonic*

THEOREM 4.3. *Algorithm* PAR_ALLEIG1 *is correct if Assumptions 1A, 1C, 1E, 3 and 4 hold and* FloatingCount(x) *is monotonic.*

However, when FloatingCount(x) is nonmonotonic, Algorithm PAR_ALLEIG1 is still incorrect! The error in the algorithm is that when FloatingCount(x) is nonmonotonic, a desired eigenvalue may be computed more than once. For example, suppose $n = p = 3$, $n_\alpha^{(0)} = 0$, $n_\beta^{(0)} = n_\alpha^{(1)} = 2$, $n_\beta^{(1)} = n_\alpha^{(2)} = 1$, and $n_\beta^{(2)} = 3$. In this case, the second eigenvalue will be computed both by processors 0 and 2.

Algorithm PAR_ALLEIG2 (see figure 4.6) corrects this problem by making processor 0 find all the initial tasks that are input to algorithm SER_BRACKET. These initial tasks are formed such that $n_\alpha^{(i)} \leq n_\beta^{(i)}$, for $i = 0, \ldots, p-1$, and are then communicated to the other processors. Note that these initial tasks may also be formed by computing $n_\alpha^{(i)}$ and $n_\beta^{(i)}$ in parallel on each processor, making sure $n_\alpha^{(i)} \leq n_\beta^{(i)}$, and then doing two max scans replacing $n_\alpha^{(i)}$ by $\max_{j \leq i} n_\alpha^{(i)}$ and $n_\beta^{(i)}$ by $\max_{j \leq i} n_\beta^{(i)}$. This would take $\log(p)$ steps on p processors. The function send(i,(γ_0,γ_1,n_{γ_0},n_{γ_1})) sends the task (γ_0,γ_1,n_{γ_0},n_{γ_1}) to processor i, while receive(0,(α,β,n_α,n_β)) receives the task sent by processor 0. Thus, we have the following theorem:

THEOREM 4.4. *Algorithm* PAR_ALLEIG2 *is correct if Assumptions 1A, 1C, 1D, 3 and 4 hold.*

Proof. The theorem is proved in section 7.2. □

```
subroutine Par_Alleig2($n$,$T$,$\tau$) /* computes all the eigenvalues of $T$ in parallel */
    $i$ = My_Processor_Number();
    $(gl, gu)$ = Compute_Gerschgorin($n$,$T$);
    $average\_width = (gu - gl)/p$;
    if ($i = 0$) then
        $n_{\gamma_0} = 0$; $\gamma_0 = gl$; $\gamma_1 = \gamma_0$;
        for ($j = 0; j < p; j = j + 1$) do      /* does a max scan */
            $\gamma_1 = \max(\gamma_0 + average\_width, \gamma_0)$;
            $n_{\gamma_1} = \max(\mathrm{FloatingCount}(\gamma_1), n_{\gamma_0})$;
            send($j$,($\gamma_0$,$\gamma_1$,$n_{\gamma_0}$,$n_{\gamma_1}$));
            $\gamma_0 = \gamma_1$; $n_{\gamma_0} = n_{\gamma_1}$;
        end for
    end if
    receive(0,($\alpha$,$\beta$,$n_\alpha$,$n_\beta$));
    if ($i = p - 1$) then
        $\beta = gu$;
        $n_\beta = n$;
    end if
    call Ser_Bracket($n$,$T$,$\alpha$,$\beta$,$n_\alpha$,$n_\beta$,$\tau$);
end subroutine
```

Fig. 4.6. **Algorithm** Par_Alleig2 — *A* Correct *Parallel Implementation*

Note that algorithm Par_Alleig2 does not require different processors to have identical arithmetic.

4.4.2. A Practical Correct Parallel Implementation. Although correct, Algorithm Par_Alleig2 is very sensitive to the eigenvalue distribution in the Gerschgorin interval, and does not result in high speedups on massively parallel machines when the eigenvalues are not distributed uniformly. We now give a more practical parallel implementation, and prove its correctness. Algorithm Par_Alleig3 (see figure 4.7) partitions the work among the p processors by making each processor find an almost equal number of eigenvalues (for ease of presentation, we assume that p divides n). This static partitioning of work has been observed to give good performance on various parallel machines for almost all eigenvalue distributions.

In algorithm Par_Alleig3, processor i attempts to find eigenvalues $i(n/p) + 1$ through $(i+1)n/p$. The function Find_Init_Task invoked on processor i finds a floating point number $\beta^{(i)}$ such that $\mathrm{FloatingCount}(\beta^{(i)}) = (i + 1)n/p$, unless $\lambda_{(i+1)n/p}$ is part of a cluster of eigenvalues. In the latter case, Find_Init_Task finds a floating point number $\beta^{(i)}$ such that $\mathrm{FloatingCount}(\beta^{(i)})$ is bigger than $(i + 1)n/p$ and $\lambda_{(i+1)n/p}, \ldots, \lambda_{\mathrm{FloatingCount}(\beta^{(i)})}$ form a cluster relative to the user specified error tolerance, τ. Each processor i computes β, and communicates with processor $i - 1$ to receive α, and then calls algorithm Ser_Bracket with the initial task (α, β,n_α,n_β,I_α^β) returned by the function Find_Init_Task. When the eigenvalues are well separated, each processor finds an equal number of eigenvalues.

Theorem 4.5. *Algorithm* Par_Alleig3 *is correct if Assumptions 1A, 1C, 1E, 3 and 4 hold.*

Proof. The theorem is proved in section 7.2. □

```
subroutine $\text{PAR\_ALLEIG3}(n,T,\tau)$ /* computes all the eigenvalues of $T$ in parallel */
    $i = \text{MY\_PROCESSOR\_NUMBER}()$;
    $[gl,gu] = \text{COMPUTE\_GERSCHGORIN}(n,T)$;
    $(\alpha,\beta,n_\alpha,n_\beta) = \text{FIND\_INIT\_TASK}(n,T,gl,gu,0,n,\tau)$;
    call $\text{SER\_BRACKET}(n,T,\alpha,\beta,n_\alpha,n_\beta,\tau)$;
end subroutine

function $\text{FIND\_INIT\_TASK}(n,T,\alpha,\beta,n_\alpha,n_\beta,\tau)$ /* returns initial task */
    $save = \alpha$; $n_{save} = n_\alpha$;
    while ( $(n_\beta \neq (i+1)n/p)$ and $(\beta - \alpha > \min(\min_{i=n_\alpha+1}^{n_\beta} \tau_i, \max(|\alpha|,|\beta|)\varepsilon)))$
        $mid = inside(\alpha,\beta,T)$;
        $n_{mid} = \min(\max(\text{FloatingCount}(mid), n_\alpha), n_\beta)$;
        if $(n_{mid} \geq (i+1)n/p)$ then
            $\beta = mid$; $n_\beta = n_{mid}$;
        else
            $\alpha = mid$; $n_\alpha = n_{mid}$;
        end if
    end while
    if $(i \neq p-1)$ then
        send$(i+1,\beta)$;
        send$(i+1,n_\beta)$;
    end if
    if $(i \neq 0)$ then
        receive$(i-1,\alpha)$;
        receive$(i-1,n_\alpha)$;
    else
        $\alpha = save$; $n_\alpha = n_{save}$;
    end if
    return$(\alpha,\beta,n_\alpha,n_\beta)$;
end function
```

FIG. 4.7. **Algorithm** PAR_ALLEIG3 — *each processor finds an (almost) equal number of eigenvalues*

However, the above algorithm can be incorrect if the arithmetic differs on the different processors. Due to the differing arithmetics, the function FIND_INIT_TASK may return tasks such that $\beta^{(i)} > \beta^{(i+1)}$. For example, suppose $n = p = 3$, $n_\alpha^{(0)} = 0$, $n_\beta^{(0)} = n_\alpha^{(1)} = 1$, $n_\beta^{(1)} = n_\alpha^{(2)} = 2$, and $n_\beta^{(2)} = 3$ but $\beta^{(1)} > \beta^{(2)}$. In this case, processor 2 will not compute any eigenvalue since the task input to Algorithm SER_BRACKET is not a *valid* task. The fix is to explicitly ensure that $\alpha^{(i)} \leq \beta^{(i)}$ and $n_\alpha^{(i)} \leq n_\beta^{(i)}$ by doing a few max_scans [7]. This fix results in algorithm PAR_ALLEIG4 (see figure 4.8). The call $\alpha = \text{max\_scan}(\alpha)$ sets $\alpha^{(i)} = \max_{j \leq i} \alpha^{(j)}$ for $i = 1, 2, \ldots, p-1$.

THEOREM 4.6. *Algorithm* PAR_ALLEIG4 *is correct if Assumptions 1A, 1C, 1D, 3 and 4 hold.*

Proof. The theorem is proved in section 7.2. ☐

Note that in order to deal with nonmonotonicity we used some communication between processors to guarantee correctness of the parallel algorithms. We have not succeeded in devising a correct, load balanced algorithm that does not require com-

```
subroutine PAR_ALLEIG4($n$,$T$,$\tau$) /* computes all the eigenvalues of $T$ in parallel */
    $i$ = MY_PROCESSOR_NUMBER();
    [$gl$,$gu$] = COMPUTE_GERSCHGORIN($n$,$T$);
    ($\alpha$,$\beta$,$n_\alpha$,$n_\beta$) = FIND_INIT_TASK($n$,$T$,$gl$,$gu$,0,$n$,$\tau$);
    call SER_BRACKET($n$,$T$,$\alpha$,$\beta$,$n_\alpha$,$n_\beta$,$\tau$);
end subroutine

function FIND_INIT_TASK($n$,$T$,$\alpha$,$\beta$,$n_\alpha$,$n_\beta$,$\tau$) /* returns initial task */
    $save = \alpha$; $n_{save} = n_\alpha$;
    while ( $(n_\beta \neq (i+1)n/p)$ and $(\beta - \alpha > \min(\min_{i=n_\alpha+1}^{n_\beta} \tau_i, \max(|\alpha|, |\beta|)\varepsilon))$)
        $mid = inside(\alpha, \beta, T)$;
        $n_{mid} = \min(\max(\text{FloatingCount}(mid), n_\alpha), n_\beta)$;
        if $(n_{mid} \geq (i+1)n/p)$ then
            $\beta = mid$; $n_\beta = n_{mid}$;
        else
            $\alpha = mid$; $n_\alpha = n_{mid}$;
        end if
    end while
    if $(i \neq p-1)$ then
        send($i+1$,$\beta$);
        send($i+1$,$n_\beta$);
    end if
    if $(i \neq 0)$ then
        receive($i-1$,$\alpha$);
        receive($i-1$,$n_\alpha$);
    else
        $\alpha = save$; $n_\alpha = n_{save}$;
    end if
    $\alpha = \min(\alpha, \beta)$; $\beta = \max(\alpha, \beta)$;
    $\alpha = \text{max\_scan}(\alpha)$; $\beta = \text{max\_scan}(\beta)$;
    $n_\alpha = \min(n_\alpha, n_\beta)$; $n_\beta = \max(n_\alpha, n_\beta)$;
    $n_\alpha = \text{max\_scan}(n_\alpha)$; $n_\beta = \text{max\_scan}(n_\beta)$;
    return($\alpha$,$\beta$,$n_\alpha$,$n_\beta$);
end function
```

FIG. 4.8. **Algorithm** PAR_ALLEIG4 — *each processor finds an (almost) equal number of eigenvalues*

munication; this remains an open problem.

5. Roundoff Error Analysis.

As we mentioned before, Algorithm 1 was recently extended to the symmetric acyclic matrices. In [10] the implementation of Count(x) for acyclic matrices was given, see Algorithm 2. The algorithm refers to the tree $G(T)$, where node 1 is chosen (arbitrarily) as the root of the tree, and node j is called a *child* of node i if $T_{ij} \neq 0$ and node j has not yet been visited by the algorithm.

Algorithm 2: Count(x) **returns the number of eigenvalues of the symmetric acyclic matrix** T **that are less than** x**.**

```
call TreeCount($1, x, d, s$)
```

```
Count = $s$

procedure TreeCount($i, x, d, s$)        /* $i$ and $x$ are inputs, $d$ and $s$ are outputs */
    $s = 0$
    $sum = 0$
    for all children $j$ of $i$ do
        call TreeCount($j, x, d', s'$)
        $sum = sum + T_{ij}^2/d'$
        $s = s + s'$
    endfor
    $d = (T_{ii} - x) - sum$
    if $d < 0$ then $s = s + 1$
end TreeCount
```

In [10] it is also shown that barring over/underflow, the floating point version of Algorithm 2 has the same attractive backward error analysis as the floating point version of Algorithm 1: Let FloatingCount(x, T) denote the value of Count(x, T) computed in floating point arithmetic. Then FloatingCount(x, T) = Count(x, T'), where T' differs from T only slightly:

$$|T_{ij} - T'_{ij}| \leq f(C/2+2, \varepsilon)|T_{ij}| \text{ if } i \neq j \text{ and } T_{ii} = T'_{ii}, \tag{5.1}$$

where ε is the machine precision, C is the maximum number of children of any node in the graph $G(T)$ and $f(n, \varepsilon)$ is defined by

$$f(n, \varepsilon) = (1+\varepsilon)^n - 1.$$

By Assumption 2A ($n\varepsilon \leq .1$), we have [24]:

$$f(n, \varepsilon) \leq 1.06 n\varepsilon.$$

(Strictly speaking, the proof of this bound is a slight modification of the one in [10], and requires that d be computed exactly as shown in TreeCount. The analysis in [10] makes no assumption about the order in which the sum for d is evaluated, whereas the bound (5.1) for TreeCount assumes the parentheses in the sum for d are respected. Not respecting the parentheses weakens the bounds just slightly, and complicates the discussion below, but does not change the overall conclusion.)

This tiny componentwise backward error permits us to compute the eigenvalues accurately, as we now discuss. Suppose the backward error in (5.1) can change eigenvalue λ_k by at most ξ_k. For example, Weyl's Theorem [22] implies that $\xi_k \leq \|T - T'\|_2 \leq 2f(C/2+2, \varepsilon)\|T\|_2$, i.e. that each eigenvalue is changed by an amount small compared with the largest eigenvalue. If $T_{ii} = 0$ for all i, then $\xi_k \leq (1-(C+4)\varepsilon)^{1-n}|\lambda_k|$, i.e. each eigenvalue is changed by an amount small relative to itself. See [5, 12, 18] for more such bounds.

Now suppose that at some point in the algorithm we have an interval $[x, y)$, $x < y$, where

$$i = \text{FloatingCount}(x, T) < \text{FloatingCount}(y, T) = j \ . \tag{5.2}$$

Let T'_x be the equivalent matrix for which FloatingCount(x, T) = Count(x, T'_x), and T'_y be the equivalent matrix for which FloatingCount(y, T) = Count(y, T'_y), Thus $x \leq \lambda_{i+1}(T'_x) \leq \lambda_{i+1}(T) + \xi_{i+1}$, or $x - \xi_{i+1} \leq \lambda_{i+1}(T)$. Similarly, $y > \lambda_j(T'_y) \geq \lambda_j(T) - \xi_j$, or $\lambda_j(T) < y + \xi_j$. Altogether,

$$x - \xi_{i+1} \leq \lambda_{i+1}(T) \leq \lambda_j(T) < y + \xi_j \ . \tag{5.3}$$

If $j = i + 1$, we get the simpler result

$$x - \xi_j \le \lambda_j(T) < y + \xi_j \ . \tag{5.4}$$

This means that by making x and y closer together, we can compute $\lambda_j(T)$ with an accuracy of at best about $\pm\xi_j$; this is when x and y are adjacent floating point numbers and $j = i + 1$ in (5.2). Thus, in principle $\lambda_j(T)$ can be computed nearly as accurately as the inherent uncertainty ξ_j permits.

We now describe the impact of over/underflow, including division by zero. We first discuss the way division by zero is avoided in EISPACK's `bisect` routine [23], then the superior method in LAPACK's `dstebz` routine [1, 18], and finally how our alternative Algorithm 4 (Flcnt_IEEE) works (Algorithm 4 assumes IEEE arithmetic). The difficulty arises because if d' is tiny or zero, the division T_{ij}^2/d' can overflow. In addition, T_{ij}^2 can itself over/underflow.

We denote the pivot d computed when visiting node i by d_i. The floating point operations performed while visiting node i are then

$$d_i = fl((T_{ii} - x) - (\sum_{\substack{\text{all children}\\ j \text{ of } i}} \frac{T_{ij}^2}{d_j})). \tag{5.5}$$

To analyze this formula, we will let subscripted ε's and η's denote independent quantities bounded in absolute value by ε (machine precision) and $\bar\omega$ (underflow threshold). We will also make standard substitutions like $\prod_{i=1}^{n}(1+\varepsilon_i) \to (1+\tilde\varepsilon)^n$ where $|\tilde\varepsilon| \le \varepsilon$, and $(1+\varepsilon_i)^{\pm 1}\eta_j \to \eta_j$.

5.1. Model 1: Barring Overflow, Acyclic Matrix. Barring overflow, (5.5) and Assumption 2B(i) leads to

$$d_i = \{(T_{ii} - x)(1+\varepsilon_a^i) + \eta_{1i} - \sum_{\substack{\text{all children}\\ j \text{ of } i}} \frac{T_{ij}^2}{d_j}(1+\tilde\varepsilon_{ij})^{C+1} - (2C-1)\eta_{2i}\}(1+\varepsilon_b^i) + \eta_{3i}.$$

or

$$\frac{d_i}{1+\varepsilon_b^i} = (T_{ii} - x)(1+\varepsilon_a^i) - \sum_{\substack{\text{all children}\\ j \text{ of } i}} \frac{T_{ij}^2}{d_j}(1+\tilde\varepsilon_{ij})^{C+1} + 2C \cdot \eta'_{2i} + \eta'_{3i}.$$

or

$$\frac{d_i}{(1+\varepsilon_c^i)^2} = T_{ii} - x - \sum_{\substack{\text{all children}\\ j \text{ of } i}} \frac{T_{ij}^2}{d_j}(1+\hat\varepsilon_{ij})^{C+2} + (2C+1)\eta_i.$$

Let $\tilde d_i = d_i/(1+\varepsilon_c^i)^2$, finally,

$$\tilde d_i = T_{ii} + (2C+1)\eta_i - x - \sum_{\substack{\text{all children}\\ j \text{ of } i}} \frac{T_{ij}^2}{\tilde d_j}(1+\varepsilon_{ij})^{C+4}. \tag{5.6}$$

REMARK 5.1. Under Model 2, IEEE arithmetic with gradual underflow, the underflow error $(2C+1)\eta_i$ of the above equation can be replaced by $C\eta_i$ because addition and subtraction never underflow. If there is no underflow during the computations of d_i either, then (5.6) simplifies to:

$$\tilde{d}_i = T_{ii} - x - \sum_{\substack{\text{all children} \\ j \text{ of } i}} \frac{T_{ij}^2}{\tilde{d}_j}(1+\varepsilon_{ij})^{C+4}.$$

This proves (5.1), since the $\tilde{d}_i$ are the exact pivots correspond to T' where T' satisfies (5.1) and $\text{sign}(\tilde{d}_i) = \text{sign}(d_i)$.

REMARK 5.2. We need to bar overflow *in principle* for symmetric acyclic matrix with IEEE arithmetic, because if in (5.5), there are two children j_1 and j_2 of i such that $T_{ij_1}^2/d_{j_1}$ overflows to ∞ and $T_{ij_2}^2/d_{j_2}$ overflows to $-\infty$; then d_i will be NaN, not even well-defined.

5.2. Models 2 and 3: EISPACK's FlCnt_bisect, Tridiagonal Matrix.

EISPACK's FlCnt_bisect can overflow for symmetric tridiagonal or acyclic matrices with Model 1 arithmetic, and return NaN's for symmetric acyclic matrices and IEEE arithmetic since it makes no provision against overflow (see Remark 5.2). In this section, we assume T is a symmetric tridiagonal matrix whose graph is just a chain, i.e. $C = 1$. Therefore, to describe the error analysis for FlCnt_bisect, we need the following assumptions:

Assumption 2B(ii): $\bar{M} \equiv \max_{i,j} |T_{ij}| \le \sqrt{\Omega}$, and one of

Assumption 1A: Model 2. Full IEEE arithmetic with ∞ and NaN arithmetic, and with gradual underflow, or

Assumption 1A: Model 3. Full IEEE arithmetic with ∞ and NaN arithmetic, but with underflow flushing to zero.

Algorithm 3: EISPACK **FlCnt_bisect.** FloatingCount(x) **returns the number of eigenvalues of a real symmetric tridiagonal matrix T that are less than x.**

```
FloatingCount = 0;
$d_0 = 1$;
for $i = 1$ to $n$
    if ($d_i = 0$) then
        $v = |b_{i-1}|/\varepsilon$
    else
        $v = b_{i-1}^2/d_i$
    endif
    $d_i = a_i - x - v$
    if $d_i < 0$ then FloatingCount = FloatingCount + 1
endfor
```

Under Models 2 and 3, our error expression (5.6) simplifies to

$$\tilde{d}_i = a_i + 3\eta_i - x - \frac{b_{i-1}^2(1+\varepsilon_{ij})^5}{\tilde{d}_{i-1}}.$$

where $a_i = T_{ii}$ and $b_{i-1} = T_{i-1,i}$.

However, FlCnt_bisect's provision against division by zero can drastically increase the backward error bound (5.1). When $d_j = 0$ for some j in (5.5), it is easy to see that what `bisect` does is equivalent to perturbing a_j by $\varepsilon|b_j|$. This backward error is clearly small in norm, i.e. at most $\varepsilon\|T\|_2$, and so by Weyl's Theorem, can perturb computed eigenvalue by no more than $\varepsilon\|T\|_2$. If one is satisfied with absolute accuracy, this is sufficient. However, it can clearly destroy any componentwise relative accuracy, because $\varepsilon|b_j|$ may be much larger than $|a_j|$.

Furthermore, suppose there is some k such that d_k overflows, i.e. $|d_k| \geq \Omega$. Since $\bar{M} \leq \sqrt{\Omega}$, it must be b_{k-1}^2/d_{k-1} that overflows. So $\tilde{d}_k$ is $-\mathrm{sign}(b_{k-1}^2/d_{k-1}) \cdot \infty$ which has the same sign as the exact pivot corresponding to T'. But this will contribute an extra uncertainty to a_{k+1} of at most $\bar{M}^2/\Omega$, since $|b_k^2/d_k| \leq \bar{M}^2/\Omega$.

Therefore we get the following backward error for FlCnt_bisect:

$$|T_{ij} - T'_{ij}| \leq f(2.5, \varepsilon)|T_{ij}| \text{ if } i \neq j.$$

and

$$|T_{ii} - T'_{ii}| \leq \varepsilon\|T\|_2 + \frac{\bar{M}^2}{\Omega} + \begin{cases} \varepsilon\omega & \text{Model 2} \\ 3\omega & \text{Model 3} \end{cases} .$$

5.3. Models 2 and 3: FlCnt_IEEE, Tridiagonal Matrix. The following code can work only for *unreduced* symmetric tridiagonal matrices under Models 2 and 3 for the same reason as FlCnt_bisect: otherwise we could get $T_{ij_1}^2/d_{j_1} + T_{ij_2}^2/d_{j_2} = \infty - \infty = \mathrm{NaN}$. So in this section, we again assume T is a symmetric tridiagonal matrix. By using IEEE arithmetic, we can eliminate all tests in the inner loop, and so make it faster on many architectures [13]. To describe the error analysis, we again make Assumptions 1A(Model 2 or Model 3) and 2B(ii), as in section 5.2, and Assumption 2B(i), which is $\bar{B} \equiv \min_{i\neq j} T_{ij}^2 \geq \omega$.

The function `SignBit` is defined as in IEEE floating point arithmetic, i.e., `SignBit`(x) is 0 if $x > 0$ or $x = +0$, and 1 if $x < 0$ or $x = -0$.

Algorithm 4: FlCnt_IEEE. FloatingCount(x) **returns the number of eigenvalues of a real symmetric tridiagonal matrix T that are less than x.**

```
FloatingCount = 0;
$d_0 = 1$;
for $i = 1$ to $n$
    /* note that there is no provision against overflow and division by zero */
    $d_i = (a_i - x) - b_{i-1}^2/d_{i-1}$
    FloatingCount = FloatingCount + SignBit($d_i$)
endfor
```

By Assumption 2B(i), b_i^2 never underflows. Therefore when some d_i underflows, we do not have the headache of dealing with 0/0 which is NaN.

Algorithm 4 is quite similar to FlCnt_bisect except division by zero is permitted to occur, and the `SignBit`(± 0) function ($= 0$ or 1) is used to count eigenvalues (see also Section 8.1). More precisely, if $d_i = +0$, d_{i+1} would be $-\infty$, so after two steps, Count will increase by 1. On the other hand, if $d_i = -0$, d_{i+1} would be $+\infty$, hence Count also increases by 1 after two steps. Therefore, we can simply change any $d_i = -0$ to $d_i = +0$, and $d_{i+1} = +\infty$ to $d_{i+1} = -\infty$, to eliminate -0 from the analysis. Then using an analysis analogous to the last section, if we use Model 2(gradual underflow), T' differs from T by

$$|T_{ij} - T'_{ij}| \leq f(2.5, \varepsilon)|T_{ij}| \text{ if } i \neq j \text{ and } |T_{ii} - T'_{ii}| \leq \frac{\bar{M}^2}{\Omega} + \varepsilon\omega.$$

Using Model 3(flush to zero), we have the slightly weaker results that

$$|T_{ij} - T'_{ij}| \leq f(2.5, \varepsilon)|T_{ij}| \text{ if } i \neq j \text{ and } |T_{ii} - T'_{ii}| \leq \frac{\bar{M}^2}{\Omega} + 3\omega.$$

Since $\bar{M} \leq \sqrt{\Omega}$, so

$$\frac{\bar{M}^2/\Omega}{\bar{M}} \leq \frac{1}{\sqrt{\Omega}} \ll \varepsilon.$$

which tells us that $\bar{M} \leq \sqrt{\Omega}$ is a good scaling choice.

5.4. Models 1, 2 and 3: LAPACK's FlCnt_stebz routine, Acyclic Matrix. In contrast to EISPACK's FlCnt_bisect and FlCnt_IEEE, LAPACK's FlCnt_stebz can work in principle for general symmetric acyclic matrices under all three models (although its current implementation only works for tridiagonal matrices). So in this section, T is a symmetric acyclic matrix. Let $B = \max_{i \neq j}(1, T_{ij}^2) \leq \Omega$, and $\hat{p} = 2C \cdot B/\Omega$ ($\hat{p}$ is called *pivmin* in `dstebz`). In this section, we need the Assumptions 1A (Model 1, 2 or 3) and 2B(ii). Because of the Gerschgorin Disk Theorem, we can restrict our attention to those shifts x such that $|x| \leq (n+1)\sqrt{\Omega}$.

Algorithm 5: FloatingCount(x) **returns the number of eigenvalues of the symmetric acyclic matrix T that are less than x.**

```
call TreeCount($1, x, d_1, s_1$)
FloatingCount = $s_1$

procedure TreeCount($i, x, d_i, s_i$)   /* $i$ and $x$ are inputs, $d_i$ and $s_i$ are outputs */
    $d_i = fl(T_{ii} - x)$
    $s_i = 0$
    for all children $j$ of $i$ do
        call TreeCount($j, x, d_j, s_j$)
        $sum = sum + T_{ij}^2/d_j$
        $s_i = s_i + s_j$
    endfor
    $d_i = (T_{ii} - x) - sum$
    if ($|d_i| \leq \hat{p}$)   $d_i = -\hat{p}$
    if $d_i < 0$ then $s_i = s_i + 1$
end TreeCount
```

It is clear that $|d_i| \geq \hat{p}$ for each node i, so

$$|T_{ii}| + |x| + \sum_{\substack{\text{all children} \\ j \text{ of } i}} |\frac{T_{ij}^2}{d_j}| \leq (n+2)\sqrt{\Omega} + C \cdot \frac{B}{\hat{p}} \leq \frac{\Omega}{2} + C\frac{B}{2C \cdot B/\Omega} = \Omega.$$

This tells us that FlCnt_stebz never overflows and it works under all three models. For all these models, the assignment $d_i = -\hat{p}$ when $|d_i|$ is small can contribute an extra uncertainty to T_{ii} of no more than $2 \cdot \hat{p}$. Thus we have the following backward error:

$$|T_{ij} - T'_{ij}| \leq f(C/2 + 2, \varepsilon)|T_{ij}| \text{ if } i \neq j.$$

and

$$|T_{ii}-T'_{ii}| \le 2\cdot\hat{p} + \left\{ \begin{array}{ll} (2C+1)\bar{\omega} & \text{Model 1} \\ C\varepsilon\omega & \text{Model 2} \\ (2C+1)\omega & \text{Model 3} \end{array} \right. .$$

5.5. Models 1,2 and 3: FlCnt_Best_Scaling, Acyclic Matrix. Following Kahan[18], let $\xi = \omega^{1/4}\Omega^{-1/2}$ and $M = \xi\cdot\Omega = \omega^{1/4}\Omega^{1/2}$. The following code assumes that the initial data has been scaled so that

$$\bar{M} \le \frac{M}{\sqrt{2C}} \text{ and } \bar{M} \approx \frac{M}{\sqrt{2C}}.$$

This code can be used to compute the eigenvalues of general symmetric acyclic matrix, so in this section, T is a symmetric acyclic matrix. To describe the error analysis, we only need Assumption 1A. Again because of the Gerschgorin Disk Theorem, the shifts are restricted to those x such that $|x| \le (n+1)M$. The FlCnt_Best_Scaling is almost the same as the FlCnt_stebz except $\hat{p} = -\sqrt{\omega}$, so we do not repeat the code here.

Similar to FlCnt_stebz, $|d_i| \ge \sqrt{\omega}$ for any node i, so

$$|T_{ii}| + |x| + \sum_{\substack{\text{all children} \\ j \text{ of } i}} |\frac{T_{ij}^2}{d_j}| \le (n+1)M + \frac{M}{\sqrt{2C}} + C\cdot\frac{M^2/2C}{\omega^{1/2}} \le \frac{\Omega}{2} + C\cdot\frac{\omega^{1/2}\Omega/2C}{\omega^{1/2}} = \Omega$$

which tells us overflow **never** happens and the code can work fine under all the models we mentioned. For all the models, The backward error bound becomes,

$$|T_{ij}-T'_{ij}| \le f(C/2+2,\varepsilon)|T_{ij}| \text{ if } i \ne j.$$

and

$$|T_{ii}-T'_{ii}| \le 2\sqrt{\omega} + \left\{ \begin{array}{ll} (2C+1)\bar{\omega} & \text{Model 1} \\ C\varepsilon\omega & \text{Model 2} \\ (2C+1)\omega & \text{Model 3} \end{array} \right. .$$

5.6. Error Bounds For Eigenvalues. We need the following lemma to give error bounds for the computed eigenvalues.

Lemma 5.1. *Assume T is an acyclic matrix and* FloatingCount(x,T) = Count(x, T')*, where T' differs from T only slightly:*

$$|T_{ij}-T'_{ij}| \le \alpha(\varepsilon)|T_{ij}| \text{ if } i \ne j \text{ and } |T_{ii}-T'_{ii}| \le \beta.$$

where $\alpha(\varepsilon) \ge 0$ is a function of ε and $\beta \ge 0$. Then this backward error can change the eigenvalues λ_k by at most ξ_k where

$$\xi_k \le 2\alpha(\varepsilon) \parallel T \parallel_2 + \beta. \tag{5.7}$$

Proof. By Weyl's Theorem [22],

$$\xi_k \le \|T-T'\|_2 \le \||T-T'|\|_2 \le \|\alpha(\varepsilon)|T-\Lambda| + \beta I\|_2 \le \alpha(\varepsilon)\||T-\Lambda|\|_2 + \beta.$$

TABLE 5.1
Backward Error Bounds for Symmetric Tridiagonal Matrices

Algorithms	Model 1		Model 2		Model 3	
	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β
FlCnt_bisect	—	—	$f(2.5,\varepsilon)$	$\varepsilon\|T\|_2+\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$\varepsilon\|T\|_2+\frac{\bar{M}^2}{\Omega}+3\omega$
FlCnt_stebz	$f(2.5,\varepsilon)$	$2\hat{p}+3\bar{\omega}$	$f(2.5,\varepsilon)$	$2\hat{p}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$2\hat{p}+3\omega$
FlCnt_Best_Scaling	$f(2.5,\varepsilon)$	$2\sqrt{\bar{\omega}}+3\bar{\omega}$	$f(2.5,\varepsilon)$	$2\sqrt{\omega}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$2\sqrt{\omega}+3\omega$
FlCnt_IEEE	—	—	$f(2.5,\varepsilon)$	$\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$\frac{\bar{M}^2}{\Omega}+3\omega$

TABLE 5.2
Error Bounds ξ_k of Eigenvalues for Symmetric Tridiagonal Matrices

Algorithms	Model 1	Model 2	Model 3
FlCnt_bisect	—	$[2f(2.5,\varepsilon)+\varepsilon]\|T\|_2+\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$[2f(2.5,\varepsilon)+\varepsilon]\|T\|_2+\frac{\bar{M}^2}{\Omega}+3\omega$
FlCnt_stebz	$2f(2.5,\varepsilon)\|T\|_2+2\hat{p}+3\bar{\omega}$	$2f(2.5,\varepsilon)\|T\|_2+2\hat{p}+\varepsilon\omega$	$2f(2.5,\varepsilon)\|T\|_2+2\hat{p}+3\omega$
FlCnt_Best_Scaling	$2f(2.5,\varepsilon)\|T\|_2+2\sqrt{w}+3\bar{\omega}$	$2f(2.5,\varepsilon)\|T\|_2+2\sqrt{w}+\varepsilon\omega$	$2f(2.5,\varepsilon)\|T\|_2+2\sqrt{w}+3\omega$
FlCnt_IEEE	—	$2f(2.5,\varepsilon)\|T\|_2+\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$2f(2.5,\varepsilon)\|T\|_2+\frac{\bar{M}^2}{\Omega}+3\omega$

and

$$\||T-\Lambda|\|_2 = \|T-\Lambda\|_2 \le \|T\|_2 + \|\Lambda\|_2 \le 2\|T\|_2.$$

where $\Lambda = \text{diag}(d_i)$ which is the diagonal part of T. Therefore,

$$\xi_k \le 2\alpha(\varepsilon)\|T\|_2 + \beta.$$

$\square$

In Tables 5.1 through 5.4, we present the backward errors $\alpha(\varepsilon)$ and β, and the corresponding error bounds ξ_k for the various algorithms under different models of arithmetic.

5.7. Correctness of the Gerschgorin Bound. In this section, we will prove the correctness of the Gerschgorin interval returned by the function COMPUTE_GERSCHGORIN (see figure 4.3). We will need assumptions 1A and 4B.

In exact arithmetic,

$$glexact = \min_i (T_{ii} - \sum_{j\neq i} |T_{ij}|); \quad guexact = \max_i (T_{ii} + \sum_{j\neq i} |T_{ij}|).$$

So, $bnorm = \max(|glexact|, |guexact|) = \|T\|_\infty$. Notice that

$$fl((T_{ii} - \sum_{j\neq i} |T_{ij}|)) = (T_{ii}(1+\delta_i)^{k_i} - \sum_{j\neq i} |T_{ij}|(1+\delta_j)^{k_j}).$$

Therefore,

$$|fl(glexact) - glexact| \le f(C,\varepsilon)\|T\|_\infty \le 2n\varepsilon\|T\|_\infty = 2n\varepsilon * bnorm.$$

Similarly, $|fl(guexact) - guexact| \le 2n\varepsilon * bnorm$. With assumption 4B, this proves that if we let

$$gl = fl(glexact) - 2n\varepsilon * bnorm - \xi_0; \quad gu = fl(guexact) + 2n\varepsilon * bnorm + \xi_n.$$

TABLE 5.3
Backward Error Bounds for Symmetric Acyclic Matrices

Algorithms	Model 1		Model 2		Model 3	
	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β
FlCnt_bisect	—	—	—	—	—	—
FlCnt_stebz	$f(C/2{+}2,\varepsilon)$	$2\hat{p}{+}(2C{+}1)\bar{\omega}$	$f(C/2{+}2,\varepsilon)$	$2\hat{p}{+}C\varepsilon\omega$	$f(C/2{+}2,\varepsilon)$	$2\hat{p}{+}(2C{+}1)\omega$
FlCnt_Best_Scaling	$f(C/2{+}2,\varepsilon)$	$2\sqrt{\omega}{+}(2C{+}1)\bar{\omega}$	$f(C/2{+}2,\varepsilon)$	$2\sqrt{\omega}{+}C\varepsilon\omega$	$f(C/2{+}2,\varepsilon)$	$2\sqrt{\omega}{+}(2C{+}1)\omega$
FlCnt_IEEE	—	—	—	—	—	—

TABLE 5.4
Error Bounds ξ_k of Eigenvalues for Symmetric Acyclic Matrices, $g(\varepsilon) = f(C/2+2,\varepsilon)$

Algorithms	Model 1	Model 2	Model 3
FlCnt_bisect	—	—	
FlCnt_stebz	$2g(\varepsilon)\|T\|_2{+}2\hat{p}{+}(2C{+}1)\bar{\omega}$	$2g(\varepsilon)\|T\|_2{+}2\hat{p}{+}C\varepsilon\omega$	$2g(\varepsilon)\|T\|_2{+}2\hat{p}{+}(2C{+}1)\omega$
FlCnt_Best_Scaling	$2g(\varepsilon)\|T\|_2{+}2\sqrt{w}{+}(2C{+}1)\bar{\omega}$	$2g(\varepsilon)\|T\|_2{+}2\sqrt{w}{+}C\varepsilon\omega$	$2g(\varepsilon)\|T\|_2{+}2\sqrt{w}{+}(2C{+}1)\omega$
FlCnt_IEEE	—	—	

then we can claim

$$\text{FloatingCount}(gl) = 0; \;\; \text{FloatingCount}(gu) = n.$$

For the algorithms we mentioned in the previous sections, we can obtain the upper bounds for ξ_k under certain additional appropriate assumptions, which enable us to give more specific and explicit Gerschgorin bounds computed by the routine COMPUTE_GERSCHGORIN (see Table 5.5). For instance, the error bound of FlCnt_bisect for symmetric tridiagonal matrices is at most $[2f(2.5,\varepsilon)+\varepsilon]\|T\|_2 + \bar{M}^2/\Omega + 3\omega$, with Assumption 2C(i): $\bar{M} \geq \omega/\varepsilon$, we have

$$\begin{aligned} & [2f(2.5,\varepsilon)+\varepsilon]\|T\|_2 + \bar{M}^2/\Omega + 3\omega \leq (2*2.5*1.06\varepsilon+\varepsilon)\|T\|_2 + \frac{\bar{M}}{\Omega}\bar{M} + 3\varepsilon\bar{M} \\ \leq \;\; & 7\varepsilon * bnorm + \varepsilon * bnorm + 3\varepsilon * bnorm = 11\varepsilon * bnorm. \end{aligned}$$

According to Table 5.5, if we let

$$gl = fl(glexact) - (10n+6)\varepsilon * bnorm; \;\; gu = fl(guexact) + (10n+6)\varepsilon * bnorm. \tag{5.8}$$

Then we have

$$\text{FloatingCount}(gl) = 0; \;\; \text{FloatingCount}(gu) = n$$

in all situations, which shows the Gerschgorin Bound (5.8) is correct for EISPACK's FlCnt_bisect, LAPACK's FlCnt_stebz, FlCnt_IEEE and FlCnt_Best_Scaling.

5.8. The Splitting Criterion. The splitting criterion asks if an offdiagonal entry b_i is small enough in magnitude to set to zero without making unacceptable perturbations in the eigenvalues. This is useful because setting b_i to zero splits T into two independent subproblems which can be solved faster (serially or in parallel). If we are only interested in absolute accuracy, then Weyl's Theorem [22] guarantees that the test

$$\text{if } |b_i| \leq tol \text{ then } b_i = 0 \tag{5.9}$$

TABLE 5.5
Upper Bounds for ξ_k for Different Algorithms under Different Models

Algorithms	Matrix	Additional Assumptions	Bounds of ξ_k
FlCnt_bisect	Tridiagonal	Assumption 2C(i)	$11\varepsilon * bnorm$
FlCnt_stebz	Acyclic	Assumptions 2C(i), 2C(ii)	$(8n+6)\varepsilon * bnorm$
FlCnt_Best_Scaling	Acyclic	—	$(4n+8)\varepsilon * bnorm$
FlCnt_IEEE	Tridiagonal	Assumption 2C(i)	$10\varepsilon * bnorm$

will not change any eigenvalue by more than *tol*. An alternative test for setting b_i to zero is

$$\begin{array}{rcl}\delta &=& (a_{i+1}-a_i)^2/4\\ \rho &=& (1-2^{-1/2})(b_{i-1}^2+b_{i+1}^2)\\ \tau &=& \frac{b_i^2}{\delta+\rho}(2\rho+\frac{\delta b_i^2}{\delta+\rho})\\ \text{if} && \tau < tol^2 \text{ then } b_i = 0\end{array}$$

This also guarantees that no eigenvalue will change by more than *tol* (in fact it guarantees that the square root of the sum of the squares of the changes in all the eigenvalues is bounded by *tol*) [17, 22]. Although it sets b_i to zero more often than the simpler test (5.9), it is much more expensive.

To guarantee relative accuracy, we need the following new result:

LEMMA 5.2. *Let T be a tridiagonal matrix where for a fixed i*

$$|b_i| \le tol \cdot (|a_i a_{i+1}|)^{1/2}$$

Let $T' = T$ except for setting $b_i' = 0$. Then there exist other tridiagonal matrices T_1 and T_2 with the following properties:

1. *$\lambda_{1i} \le \lambda_i' \le \lambda_{2i}$, where T_j has eigenvalues $\lambda_{j1} \le \cdots \le \lambda_{jn}$.*
2. *$\lambda_{1i} \le \lambda_i \le \lambda_{2i}$,*
3. *$T_1 = T_2 = T$ except for entries (i,i) and $(i+1,i+1)$ which differ from those of T by factors $1 \pm tol$.*

In other words, the eigenvalues of T' and T lie between the eigenvalues of matrices T_1 and T_2, where the entries of T_j approximate those of T with relative accuracy *tol*. This is a nearly unimprovable backward error bound. Combined with [5, Theorem 4], this easily yields

COROLLARY 5.3. *Let T be γ-s.d.d., and suppose $tol < (1-\gamma)/(1+\gamma)$ (see [5] for definitions). Suppose $|b_i| \le tol \cdot (|a_i a_{i+1}|)^{1/2}$, and let $T' = T$ except for $b_i' = 0$. Then*

$$\exp\Big(\frac{-tol}{1-\gamma\frac{1+tol}{1-tol}}\Big) \le \frac{\lambda_i'}{\lambda_i} \le \exp\Big(\frac{tol}{1-\gamma\frac{1+tol}{1-tol}}\Big)$$

When $tol \ll 1-\gamma$ then

$$1-\frac{tol}{1-\gamma} \le \frac{\lambda_i'}{\lambda_i} \le 1+\frac{tol}{1-\gamma}$$

PROOF OF LEMMA 5.2. By the Courant Minimax Theorem [22] it suffices to construct T_1 and T_2 satisfying condition iii in the Lemma such that for all vectors x

$$x^T T_1 x \le x^T T' x \le x^T T_2 x \text{ and } x^T T_1 x \le x^T T x \le x^T T_2 x \tag{5.10}$$

Since all the matrices differ only in the i-th and $i+1$-st rows and columns, it suffices to consider two by two matrices only:

$$T = \begin{bmatrix} a_i & b_i \\ b_i & a_{i+1} \end{bmatrix} , \; T' = \begin{bmatrix} a_i & 0 \\ 0 & a_{i+1} \end{bmatrix} ,$$

We claim that the following matrices satisfy condition (5.10):

$$T_1 = \begin{bmatrix} a_i - tol|a_i| & b_i \\ b_i & a_{i+1} - tol|a_{i+1}| \end{bmatrix} , \; T_2 = \begin{bmatrix} a_i + tol|a_i| & b_i \\ b_i & a_{i+1} + tol|a_{i+1}| \end{bmatrix}$$

To prove this requires us to verify that

$$\begin{bmatrix} \pm tol|a_i| & 0 \\ 0 & \pm tol|a_{i+1}| \end{bmatrix} \text{ and } \begin{bmatrix} \pm tol|a_i| & b_i \\ b_i & \pm tol|a_{i+1}| \end{bmatrix}$$

are positive (negative) semidefinite, which is immediately implied by the assumption of the lemma.

This leads us to recommend the splitting criterion

$$\text{if } |b_i| \le tol|a_i a_{i+1}|^{1/2} \text{ then } b_i = 0 \tag{5.11}$$

since this does not change the eigenvalues any more than relative perturbations of size *tol* in the matrix entries. Note that it will never be applied to tridiagonals with zero diagonal unless b_i is exactly zero.

This criterion is more stringent than the criterion in the EISPACK code bisect [23], which essentially is

$$\text{if } |b_i| \le tol(|a_i| + |a_{i+1}|) \text{ then } b_i = 0. \tag{5.12}$$

Note that this is at least about as stringent as the absolute accuracy criterion (5.9) but less stringent than the relative accuracy criterion (5.11). To see that it can fail to deliver high relative accuracy when (5.11) will succeed, consider the example

$$\begin{bmatrix} 10^{20} & 5 \cdot 10^9 \\ 5 \cdot 10^9 & 1 \end{bmatrix}$$

which is γ-s.d.d. with $\gamma = .5$ [5]. Let $tol = 10^{-10}$. Then EISPACK would set the offdiagonals to zero, returning eigenvalues 10^{20} and 1. The true eigenvalues (to about 20 digits) are 10^{20} and .75.

Note that criterion (5.11) could possibly be used as the stopping criterion in a QR algorithm [5] in the hopes of attaing high relative accuracy. However, the rounding errors in any existing QR algorithm generally cause far more inaccuracy in the computed eigenvalues than the currently used stopping criteria (which are generally identical to (5.12)).

6. Proof of Monotonicity of FloatingCount(x).

In 1966 Kahan proved but did not publish the following result [18]: if the floating point arithmetic is monotonic, then FloatingCount(x) is a monotonically increasing function of x for symmetric tridiagonal matrices. That monotonic floating point arithmetic is *necessary* for FloatingCount(x) to be monotonic is easily seen by considering 1-by-1 matrices: if addition fails to be monotonic so that $x < x'$ but $fl(a_1 - x) < 0 < fl(a_1 - x')$, then FloatingCount($x$) =

$1 > 0 = \text{FloatingCount}(x')$. In this section, we will extend this proof of monotonicity of FloatingCount(x) to symmetric acyclic matrices.

In section 2.1, we mentioned that we will number the rows and columns of T in preorder, as they are accessed by Algorithm 2 (see section 5). This means node 1 is the root of the tree, since it is accessed first, and children are numbered higher than their parents. This lets us relabel the variables in Algorithm 2 as follows, where we also introduce roundoff:

Algorithm 6: Count(x) **returns the number of eigenvalues of the symmetric acyclic matrix T that are less than x.**

```
call TreeCount($1, x, d_1, s_1$)
Count = $s_1$

procedure TreeCount($i, x, d_i, s_i$)
    /* $i$ and $x$ are inputs, $d_i$ and $s_i$ are outputs */
    $d_i = fl(T_{ii} - x)$
    $s_i = 0$
    for all children $j$ of $i$ do
        call TreeCount($j, x, d_j, s_j$)
        $d_i = fl(d_i - fl(T_{ij}^2/d_j))$
        $s_i = s_i + s_j$
    endfor
    if $d_i < 0$ then $s_i = s_i + 1$
end TreeCount
```

(Without loss of generality we ignore roundoff in computing T_{ij}^2.) Thus s_i is the total number of negative d_j in the subtree rooted at i (including d_i). We may summarize Algorithm 6 more briefly by

$$d_i = fl(fl(T_{ii} - x) - fl(\sum_{j\in C(i)} fl(\frac{T_{ij}^2}{d_j}))) \tag{6.1}$$

$$s_i = \sum_{j\in C(i)} s_j + \begin{cases} 0 & \text{if } d_i \geq 0 \\ 1 & \text{if } d_i < 0 \end{cases} \tag{6.2}$$

where the sums are over the set $C(i)$ of all children of i.

Let x be a floating point number, and let x' denote the next floating point number larger than x. To distinguish the results of Algorithm 6 for different x we will write $s_i(x)$ and $d_i(x)$. The theorem we wish to prove is:

THEOREM 6.1. *If the floating point arithmetic used to implement Algorithm 6 is monotonic, then $s_i(x) \leq s_i(x')$.*

We introduce some more definitions. In these definitions, y is always a floating point number. The number y is a *zero* of d_i if $d_i(y) \geq 0 > d_i(y')$. The number y is a *pole* of d_i if $d_i(y) < d_i(y')$. It is called a *positive pole* if in addition to being a pole $d_i(y)d_i(y') > 0$ or $d_i(y) = 0$. It is called a *negative pole* if in addition to being a pole $d_i(y)d_i(y') < 0$ or $d_i(y') = 0$.

Suppose that some s_i is decreasing; we want to find a contradiction.

LEMMA 6.2. *Let m be the largest m such that s_m is decreasing. This means that for some y, $s_m(y) > s_m(y')$. Then in fact $d_m(y) < 0 \leq d_m(y')$, i.e. y is a negative pole of d_m.*

Proof. Since m is the largest integer for which s_m is decreasing, we must have $s_k(y) \le s_k(y')$ for all children k of m. Now write

$$\begin{aligned} 0 &> s_m(y') - s_m(y) \\ &= \{s_m(y') - \sum_{k\in C(m)} s_k(y')\} + \{\sum_{k\in C(m)} s_k(y') - \sum_{k\in C(m)} s_k(y)\} \\ &+ \{\sum_{k\in C(m)} s_k(y) - s_m(y)\} \\ &\equiv t_1 + t_2 + t_3 \ . \end{aligned}$$

From (6.2) we conclude $t_1 \ge 0$ and $t_3 \ge -1$. From the definition of m we conclude $t_2 \ge 0$. These inequalities have one solution, namely $t_1 = t_2 = 0$ and $t_3 = -1$. From $t_1 = 0$ we conclude that $d_m(y') \ge 0$, and from $t_3 = -1$ we conclude $d_m(y) < 0$. In particular, this means y is a *negative pole* of d_m. □

LEMMA 6.3. *If y is a pole of d_i, then i must have a child j for which y is either a positive pole or a zero.*

Proof. If y is a pole of d_i, then for some child j of i we must have

$$fl(\frac{T_{ij}^2}{d_j(y)}) > fl(\frac{T_{ij}^2}{d_j(y')}) \tag{6.3}$$

Otherwise all children would satisfy

$$fl(\frac{T_{ij}^2}{d_j(y)}) \le fl(\frac{T_{ij}^2}{d_j(y')})$$

and so by the monotonicity of arithmetic

$$fl(\sum_{j\in C(i)} fl(\frac{T_{ij}^2}{d_j(y)})) \le fl(\sum_{j\in C(i)} fl(\frac{T_{ij}^2}{d_j(y')}))$$

Arithmetic monotonicity further implies

$$fl(T_{ii} - y) \ge fl(T_{ii} - y')$$

and finally

$$fl(fl(T_{ii} - y) - fl(\sum_{j\in C(i)} fl(\frac{T_{ij}^2}{d_j(y)}))) \ge fl(fl(T_{ii} - y') - fl(\sum_{j\in C(i)} fl(\frac{T_{ij}^2}{d_j(y')})))$$

or $d_i(y) \ge d_i(y')$, contradicting the assumption that y is a pole. Applying arithmetic monotonicity to (6.3) we conclude

$$\frac{T_{ij}^2}{d_j(y)} > \frac{T_{ij}^2}{d_j(y')} \ .$$

This means either $d_j(y) \ge 0 > d_j(y')$ (i.e. y is a zero of d_j) or $d_j(y) < d_j(y')$ and $d_j(y) \cdot d_j(y') > 0$ (y is a positive pole of d_j) or $0 = d_j(y) < d_j(y')$ (y is a positive pole of d_j). □

REMARK 6.1. The proof of the last lemma does not depend on the order in which the additions and subtractions of T_{ii}, y, and T_{ij}^2/d_j are carried out. It is also not damaged by inserting the line "if $|d_i| < tol$ then $d_i = -tol$" just before "if $d_i < 0$ then $s_i = s_i + 1$" in Algorithm 6; this is done in practice to avoid overflow and division by zero; see Algorithm 5 and [1, 18]. However, the proof does *not* work for the algorithm used to avoid overflow in the subroutine `bisect` [23]. This is because `bisect` tests if a computed d_j is exactly zero, and increases if it is; this can increase $d_i(y')$ past $d_i(y)$ even if inequalities (6.3) are not satisfied. The example in section 4.1 shows that monotonicity can indeed fail in practice.

LEMMA 6.4. *If y is a pole of d_i, then there must be a l in the subtree rooted at i such that y is a zero of d_l and for all d_j on the path from i to l, y is a positive pole of d_j.*

Proof. We can apply Lemma 6.3 to i to find a child l which is either a zero or a positive pole. If it is a zero we are done, and otherwise we apply Lemma 6.3 again to l. This process must end in a zero since the leaves are of the form $d_l(x) = fl(T_{ll} - x)$ and so can only be zeros by arithmetic monotonicity. □

Now use Lemma 6.2 to conclude that there is an m such that y is a negative pole of d_m, and

$$\sum_{k \in C(m)} s_k(y) = \sum_{k \in C(m)} s_k(y') \quad . \tag{6.4}$$

Use Lemma 6.4 to conclude that there is some l in the tree rooted at m for which y is a zero. This means $d_l(y) \geq 0 > d_l(y')$, so that d_l contributes one more to the right hand side of (6.4) than to the left hand side. So to maintain (6.4) there must be another p in the tree rooted at m with $d_p(y) < 0 \leq d_p(y')$, i.e. y is a negative pole of d_p. By Lemma 6.4, p cannot lie on the path from m to l, since only positive poles lie on this path. Therefore, again by Lemma 6.4, there must be a $q \neq l$ in the tree rooted at p such that y is a zero of d_q. But this means d_p and d_q together contribute equally to both sides of (6.4), and so cannot balance d_l. By the same argument, any other negative pole which would balance d_l has a counterbalancing zero. Therefore (6.4) cannot be satisfied. This contradiction proves Theorem 6.1.

7. Proofs of Correctness.

We now present the proofs of the various theorems stated in section 4.

Proof of Theorem 4.1 To prove the theorem, we first prove the inductive assertion that for every task $(\alpha, \beta, n_\alpha, n_\beta, I_{n_\alpha}^{n_\beta})$ in the $Worklist$, $\exists\ x_l \leq \beta, x_u \geq \alpha$ such that FloatingCount$(x_u) \leq n_\alpha < n_\beta \leq$ FloatingCount(x_l). The inductive assertion is true for the initial task in the $Worklist$ by the assumption on the input task. Suppose that the inductive assertion holds for some task in the $Worklist$. We prove that the assertion holds for the subtasks created by this task. Statement 9 in figure 4.2 ensures that $n_\alpha \leq n_{mid} \leq n_\beta$. A new subtask is added to the $Worklist$ if

1. $n_\alpha < n_{mid}$. Statement 9 implies that $n_{mid} = \min(\text{FloatingCount}(mid), n_\beta)$ and therefore, $n_{mid} \leq$ FloatingCount(mid). Thus, FloatingCount$(x_u) \leq n_\alpha < n_{mid} \leq$ FloatingCount(mid), and the inductive assertion holds for the new subtask $(\alpha, mid, n_\alpha,$
$n_{mid}, I_{n_\alpha}^{n_{mid}})$.
2. $n_{mid} < n_\beta$, which implies that $n_{mid} \geq$ FloatingCount(mid). Thus, FloatingCount$(mid) \leq n_{mid} < n_\beta \leq$ FloatingCount(x_l), and the inductive assertion holds for the new subtask $(mid, \beta, n_{mid}, n_\beta, I_{n_{mid}}^{n_\beta})$.

Hence our inductive assertion holds for any task in the $Worklist$. Note that our inductive assertion implies that FloatingCount has an n_β^{th} jump-point $\lambda''_{n_\beta} < \beta$ and an $n^{th}_{\alpha+1}$ jump-point $\lambda''_{n_{\alpha+1}} \geq \alpha$.

Let $\lambda' = \min(\max(fl(\frac{\alpha+\beta}{2}), \alpha), \beta)$ be an eigenvalue output by Algorithm SER_BRACKET. For simplicity, we assume that the eigenvalue is of multiplicity 1, i.e., $n_\beta = n_\alpha + 1$. Suppose that $\lambda''_{n_\beta} \in [\alpha, \beta)$. By the working of the algorithm, $|\lambda' - \lambda''_{n_\beta}| \leq \tau_{n_\beta}$, and by the assumption about FloatingCount(x), $|\lambda''_{n_\beta} - \lambda_{n_\beta}| \leq \xi_{n_\beta}$. By the triangle inequality, $|\lambda' - \lambda_{n_\beta}| \leq \tau_{n_\beta} + \xi_{n_\beta}$. If $\lambda''_{n_\beta} \notin [\alpha, \beta)$, then there must be two n^{th}_β jump-points, one smaller than α and the other greater than β. In this case, $|\lambda' - \lambda_{n_\beta}| \leq 2\xi_{n_\beta}$. Hence the computed eigenvalues are correct to within the user specified error tolerance. The proof when $n_\beta > n_\alpha + 1$ is similar. By Assumption 3, *fl(inside*(α, β, T)*)* $\in (\alpha, \beta)$ for all α, β that arise (note that $\alpha \leq \beta$ for all tasks in the *Worklist*). Thus, all subtasks have strictly smaller intervals, and the algorithm must terminate. At any stage of the algorithm, the index sets corresponding to all the tasks in the *Worklist* form a partition of the initial index set, $I^{n_{right}}_{n_{left}}$. Each index i is contained in exactly one index set and hence, each desired eigenvalue is computed exactly once. It is also clear that the computed eigenvalues are in sorted order.

7.1. A Necessary and Sufficient Condition for Correctness. Having found a rather simple fix to the problem of nonmonotonicity in serial bracketing, we now prove a necessary and sufficient condition for the correctness of any serial or parallel implementation of the bracketing algorithm. First, we define a few terms to help us in the ensuing discussion. Consider a *valid* task $T = (\alpha, \beta, n_\alpha, n_\beta, O)$. We say that this task *covers* the index set O. If tasks $T_1, \ldots, T_k$ cover the index sets $O_1, \ldots, O_k$ respectively, then the set of tasks $\{T_1, \ldots, T_k\}$ is said to *cover* the index set $O_1 \cup O_2 \cdots \cup O_k$. We define an *Index Cover* to be a set of tasks which covers the user index set U. A *Disjoint Index Cover* is an index cover such that the index sets covered by any pair of tasks in the index cover are disjoint.

We assume that all the bracketing algorithms discussed below maintain a set of tasks (either explicitly or implicitly). Each task in this set, $(\alpha, \beta, n_\alpha, n_\beta, O)$, is assumed to be such that $\alpha \leq \beta$ and FloatingCount(x_u) $\leq n_\alpha < n_\beta \leq$ FloatingCount(x_l) for some $x_u \geq \alpha, x_l \leq \beta$. When the width of an interval corresponding to some task becomes smaller than $\min_{i=n_\alpha+1}^{n_\beta} \tau_i$, this task is marked as a *final* task, and is not further refined. We will refer to an interval corresponding to a *final* task as a *final* interval. At the end of the algorithm, the midpoints of the *final* intervals are output as the eigenvalues corresponding to the index sets covered by the respective *final* tasks.

Consider the two tasks $T_1 = (\alpha_1, \beta_1, n_{\alpha_1}, n_{\beta_1}, O_1)$, and $T_2 = (\alpha_2, \beta_2, n_{\alpha_2}, n_{\beta_2}, O_2)$. Suppose that $\min(\max(fl((\alpha_1+\beta_1)/2), \alpha_1), \beta_1) \leq \min(\max(fl((\alpha_2+\beta_2)/2), \alpha_2), \beta_2)$. We say that the pair T_1 and T_2 is ordered if $\forall i \in O_1, \forall j \in O_2, i \leq j$. A set of tasks is said to be ordered if every pair of tasks in this set is ordered.

THEOREM 7.1. *A bracketing algorithm is correct iff its final tasks form an ordered disjoint index cover.*

Proof. Suppose the *final* tasks form an ordered disjoint index cover. Consider a *final* task $(\alpha, \beta, n_\alpha, n_\beta, O)$ where $O \subseteq I^{n_\beta}_{n_\alpha}$. Then $\forall i \in O$, the eigenvalues output from this task are $\lambda'_i = \min(\max(fl((\alpha+\beta)/2), \alpha), \beta)$. By reasoning similar to that in the proof of Theorem 4.1 $|\lambda_i - \lambda'_i| \leq \tau_i + 2\xi_i$, and every eigenvalue output is computed correctly. Since the *final* tasks form a disjoint index cover, every desired eigenvalue is output exactly once. All *final* tasks are ordered, hence the desired eigenvalues are output in sorted order.

If the *final* tasks do not form an index cover then at least one of the desired eigenvalues will not be output. If any two *final* tasks cover intersecting index sets, then some eigenvalue will be output more than once, and if some pair of *final* tasks is not ordered, then the eigenvalues output will not be in sorted order. Hence, it is necessary for the *final* tasks to form an ordered disjoint index cover. □

It is easy to verify that Algorithm SER_BRACKET satisfies the sufficient conditions of Theorem 7.1. Note that the eigenvalues output will be correct if the *final* tasks form an index cover.

7.2. Proofs of the Parallel Implementations. We now use the above characterization of correct bracketing algorithms to prove the correctness of the parallel algorithms given in section 4.4

In all our correctness proofs in this section, we will show that the tasks ($\alpha^{(i)}$,$\beta^{(i)}$, $n_\alpha^{(i)}$,$n_\beta^{(i)}$,$O^{(i)}$) that are input to SER_BRACKET in the various parallel algorithms satisfy the following :

1. $\alpha^{(i)} \leq \beta^{(i)}$ for $i = 0, 1, \ldots, p-1$.
2. Either $n_\alpha^{(i)} = n_\beta^{(i)}$, or $\exists\, x_l \leq \beta, x_u \geq \alpha$ such that FloatingCount(x_u) $\leq n_\alpha^{(i)} < n_\beta^{(i)} \leq$ FloatingCount(x_l) for $i = 0, 1, \ldots, p-1$.
3. $n_\alpha^{(0)} = 0$, $n_\beta^{(0)} = n$, $\beta^{(i)} = \alpha^{(i+1)}$ and $n_\beta^{(i)} = n_\alpha^{(i+1)}$ for $i = 0, 1, \ldots, p-2$, and $O^{(i)} = I_{n_\alpha^{(i)}}^{n_\beta^{(i)}}$ for $i = 0, 1, \ldots, p-1$.

A set of tasks that satisfy the above conditions form an ordered, disjoint index cover. The correctness of the parallel algorithm would then follow from the correctness of Algorithm SER_BRACKET.

Proof of Theorems 4.3 and 4.4 The initial interval $[\alpha^{(i)}, \beta^{(i)}]$ computed by each processor is such that $\alpha^{(i)} \leq \beta^{(i)}$. In algorithm PAR_ALLEIG1, the monotonicity of FloatingCount(x) implies that FloatingCount($\alpha^{(i)}$) $= n_\alpha^{(i)} \leq n_\beta^{(i)} =$ FloatingCount($\beta^{(i)}$). In algorithm PAR_ALLEIG2, processor 0 does a max scan that guarantees that if $n_\alpha^{(i)} \neq n_\beta^{(i)}$, then FloatingCount($\alpha^{(i)}$) $\leq n_\alpha^{(i)} < n_\beta^{(i)} =$ FloatingCount($\beta^{(i)}$).

Also $\beta^{(i)} = \alpha^{(i+1)}$, $n_\beta^{(i)} = n_\alpha^{(i+1)}$ for $i = 0, \ldots, p-2$, $\alpha^{(0)} = gl$, $n_\alpha^{(0)} = 0$, $\beta^{(p-1)} = gu$ and $n_\beta^{(p-1)} = n$.

Proof of Theorem 4.5 We first observe that $\alpha^{(0)} = gl$, $n_\alpha^{(0)} = 0$, and $\beta^{(p-1)} = gu$, $n_\beta^{(p-1)} = n$. For $i = 0, \ldots, p-2$, it can be seen that $\beta^{(i)} = \alpha^{(i+1)}$ and $n_\beta^{(i)} = n_\alpha^{(i+1)}$.

Let $mid_1^{(i)}, mid_2^{(i)}, \ldots, mid_l^{(i)}$ be the sequence of floating point numbers sampled by processor i in the function FIND_INIT_TASK. Clearly, $mid_1 = fl(inside(gl, gu, T))$ is bitwise identical on all the processors since all processors have identical arithmetic. Consider the sequence of numbers sampled by processors $i-1$ and i. Let mid_k be the first floating point number in the two sequences such that $in/p \leq n_{mid_k} < (i+1)n/p$. Clearly $mid_j^{(i-1)} = mid_j^{(i)}$ for $j \leq k$. If there is no such number then $\beta^{(i-1)} = \alpha^{(i)} = \beta^{(i)}$ and $n_\beta^{(i-1)} = n_\alpha^{(i)} = n_\beta^{(i)}$. If mid_k exists and $k = l$, then $\beta^{(i-1)} = mid_l = \alpha^{(i)} \leq \beta^{(i)}$. If $k < l$, then $mid_j^{(i-1)} \leq mid_k$ and $mid_j^{(i)} \geq mid_k$, $\forall j > k$, which implies that $\alpha^{(i)} \leq \beta^{(i)}$. Since FloatingCount($x$) is *forced* to be monotonic in function FIND_INIT_TASK, $n_\beta^{(i-1)} \leq n_{mid_k}$. It is also easy to verify that $n_\beta^{(i)} \geq (i+1)n/p > n_{mid_k}$. Thus $n_\beta^{(i-1)} = n_\alpha^{(i)} < n_\beta^{(i)}$.

Due to statement 4 in figure 4.7, $n_\beta^{(i)} =$ FloatingCount($\beta^{(i)}$), or $n_\beta^{(i)} =$ FloatingCount(x_u) $<$ FloatingCount($\beta^{(i)}$) for some $x_u > \beta^{(i)}$, or $n_\beta^{(i)} =$ FloatingCount(x_l) $>$

FloatingCount($\beta^{(i)}$) for some $x_l > \beta^{(i)}$. Thus, it is true that if $n_\alpha^{(i)} \neq n_\beta^{(i)}$ then $\exists x_l \leq \beta^{(i)}, x_u \geq \alpha^{(i)}$ such that FloatingCount$(x_u) \leq n_\alpha^{(i)} < n_\beta^{(i)} \leq$ FloatingCount(x_l).

Proof of Theorem 4.6 The max_scans in Algorithm PAR_ALLEIG4 ensure that $\alpha^{(i)} \leq \beta^{(i)}$ and $n_\alpha^{(i)} \leq n_\beta^{(i)}$ for $i = 0, 1, \ldots, p-1$. The rest of the proof is identical to the proof of Theorem 4.5.

7.3. A Family of Correct Parallel Bracketing Implementations.

We now prove the correctness of a family of bracketing algorithms $\mathcal{F}$. Every algorithm in this family starts out with one task which covers the user index set U. All tasks are obtained by refining existing tasks in the task set. A task $T = (\alpha, \beta, n_\alpha, n_\beta, O)$ with $\alpha \leq \beta$ is removed from the task set and is refined to form the k subtasks $(\alpha_1, \alpha_2, n_{\alpha_1}, n_{\alpha_2}, O_1), \ldots, (\alpha_k, \alpha_{k+1}, n_{\alpha_k}, n_{\alpha_{k+1}}, O_k)$, where $\alpha_0 = \alpha$, $\alpha_{k+1} = \beta$, $\alpha_i \leq \alpha_{i+1}$ and $n_{\alpha_i} \leq n_{\alpha_{i+1}}$, for $i = 1, \ldots, k$. Furthermore, $\cup_{i=1}^{k} O_i = O$, and $O_i = I_{n_{\alpha_i}}^{n_{\alpha_{i+1}}} \cap O$, for $i = 1, \ldots, k$. Note that a task may be refined by doing k-way multisection or by a single iteration of a fast root finder, such as Newton's or Laguerre's iteration. These subtasks are now added to the task set(some of them being marked *final* and not further refined). The tasks in the task set may be processed by one or more processors. By the manner in which the tasks are refined, it is easy to see that at each step of the algorithm, the tasks in the task set form an ordered disjoint index cover. In particular, the *final* tasks form an ordered disjoint index cover. Hence, using Theorem 7.1 we get :

THEOREM 7.2. *Every bracketing algorithm from the family $\mathcal{F}$ is correct if Assumption 1 holds.*

Algorithms SER_BRACKET and PAR_ALLEIG1 are easily seen to belong to the above family. Algorithm SER_BRACKET may be modified simply to yield a parallel algorithm, where all the tasks are enqueued and dequeued from a global $Worklist$ that is distributed across all the processors. Such an algorithm has been implemented on the CM-5 [15] — the work is initially partitioned among the processors (as in algorithms PAR_ALLEIG1 and PAR_ALLEIG3), and load imbalance is reduced by enqueuing and dequeing tasks from other processors. This algorithm has been observed to give good performance even when the initial partitioning of work is not good. Such an algorithm also belongs to the family $\mathcal{F}$, and hence is correct.

8. Practical Implementation Issues.

In section 5.3, we introduced the FlCnt_IEEE (Algorithm 4) which has no explicit tests and branches in the inner loop. However, there are some practical issues we need to consider. In this section, we will briefly discuss three topics: SignBit, division by zero and over/underflow.

8.1. SignBit.

An acceptable way to compute `SignBit`(d) in Fortran would be

$$\texttt{SignBit}(d) = 0.5 - \mathrm{SIGN}(0.5, d)$$

except that we need not a REAL number but an INTEGER to add to Count, so extra time would have to be spent upon REAL-to-INTEGER conversion. In the language C, the expression "$(d < 0.0)$" could be used in place of "`SignBit`(d)". However, both of these expedients produce `SignBit`$(-0.0) = 0$, and that can cause function FloatingCount(x) to malfunction on a few aberrant computer designs.

However, if we do some preprocessing work, like $a_i = a_i + 0$, before entering the function FloatingCount(x), then all the d_i's will never become -0 no matter

Table 8.1
Width of real variable

# of Bytes	Fortran declarations	C declarations
4	REAL, SINGLE PRECISION, REAL*4	float
8	DOUBLE PRECISION, REAL*8	double
10	EXTENDED, TEMPREAL, REAL*10	long double

whether by exact cancellation or underflow, on any commercially significant computer regardless of whether it conforms to IEEE standard 754 or 854 for Floating-Point Arithmetic. But on machines that almost conform to such a standard, departing from it only in that they may force underflowed subtractions to -0.0, function Count requires that `SignBit`$(-0.0) = 1$ in order to account properly for the sign of $\pm\infty$ produced subsequently by division by -0.0. On such computers `SignBit` must be implemented in one of the following ways, which are optional for other computers.

IEEE Standards 754 & 854 recommend that conforming computers provide a function CopySign which we may use safely in place of Fortran's SIGN function to implement

$$\texttt{SignBit}(d) = 0.5 - \mathrm{CopySign}(0.5, d).$$

Through an unfortunate accident, the arguments of CopySign have been reversed on Apple computers, which otherwise conform conscientiously to IEEE 754; they require `SignBit`$(d) = 0.5 - \mathrm{CopySign}(d, 0.5)$.

Hardly any other computer's compilers' libraries offer CopySign.

All computers can compute `SignBit`(d) quickly by shifting the sign bit of d *logically* into the rightmost bit position of an integer register leaving zeros in all the other bits. Equally good is a *twos-complement arithmetic* right shift that fills the register with copies of the sign bit, thereby producing $-$`SignBit`(d). Can either of these shifts be expressed in a higher-level language in such a way as will achieve the desired effect on every computer? Two obstacles get in the way.

The first obstacle is a disparity of widths. The INTEGER variables Count and n are likely to be 2 or 4 bytes wide. (The algorithm for FloatingCount(x) can cope with matrices T of as big a dimension n as memory capacity allows.) The REAL variable d may be 4 bytes wide but is most likely 8. INTEGERs 8 bytes wide are not in common use, so d is probably wider than the widest INTEGERs supported by the compiler. Therefore the leading(leftmost) 2 or 4 bytes of d must be first located and then extracted as an integer before the shift. Most computers, with separate registers for INTEGERs and REALs, must first store d in memory and then reload it into an integer register.

Unfortunately, different computers order the bytes of d differently. Motorola 68040s give d and the byte with its sign the same address. Intel 486s must add ((width of d) $-$ 1) to the address of d to find the byte with d's sign; the width of d in bytes can be found in table 8.1.

MIPS microprocessors can match either of the first two above. (Who gets to choose?) DEC VAXs do something a little bit different again. These diverse byte orderings constitute a second obstacle impeding efficient and portable programming of the `SignBit` function.

Both obstacles can be overcome to a degree by Conditional Compilation in C using #define and #ifdef commands in its preprocessor to find out whether the computer to which the program is being compiled belongs to a previously recognized family for which an efficient sequence of instructions has been prepared. This expedient fails to cope with new computers whose C compilers proclaim conformity with all applicable standards but whose arithmetic properties and memory-register mappings were unknown at the time the program for FloatingCount(x) was promulgated. A better solution to this problem is to include appropriately defined CopySign functions in language standards; CopySign should reveal the sign of -0.0 on a computer whose arithmetic respects it, and hide that sign on a computer whose arithmetic ignores it, and return both REAL and INTEGER values according to the type of its first argument.

8.2. Division by Zero.

IEEE 754 & 854 require by default (unless the programmer explicitly requests otherwise) that "nonzero/zero" quotients be computed as appropriately signed infinities. Of course, "finite/infinite" quotients must produce appropriately signed zeros. Function IEEE FloatingCount(x) works perfectly under these conventions; that is why its program contains no test to avert division by zero. A test like that is necessary on computers that cannot tolerate division by zero, but wastes time because division by zero is unlikely to occur by accident as often as once in a million passes around the inner loop, and is certain to be noticed by the computer if it does occur.

A little known alternative has long existed for users of proprietary Fortran compilers on IBM/370s and DEC VAXs; their programs may request that "nonzero/zero" quotients deliver the computer's biggest floating-point magnitude with the numerator's sign. This works almost as well as ∞ would in the program for FloatingCount(x).

Unfortunately, most computers that do conform to IEEE 754/854 treat division by zero no better than nonconforming computers. The trouble is linguistic; language designers and compiler writers have yet to agree upon standard ways for programmers to request IEEE standards' default infinities or IBM's or DEC's biggest magnitude. Instead, division by zero is left undefined or defined as an error; either way, computation stops.

8.3. Overflow.

Computers that abort computation when overflow occurs present the same problems as those that stop on division by zero. Once again, ways exist to tell any commercially significant computer to replace every overflow by either ∞ or the biggest finite floating- point number with an appropriate sign, but no higher level programming language provides a single way that works for every computer.

Inattention to troublesome details by designers and implementors of programming languages creates headaches for programmers of would-be portable (reusable) programs. The details in question here are the CopySign function and humane exception-handling. To get around the lack of adequate language standards, programmers must avoid those details by inserting extra tests and branches into their programs. The annoyance at having to complicate so simple a program is compounded by the performance penalty incurred by data-dependent branches taken rarely, especially on massively parallel and vectorized computers.

9. Conclusions.

We have proved necessary and sufficient conditions for a bisection algorithm to be *correct*. We have also seen examples of natural serial and parallel implementations that are incorrect, the errors arising from a nonmonotonic FloatingCount(x) and/or roundoff. Thus every bisection implementation must be carefully analyzed and proven to satisfy the sufficient conditions for correctness.

The problem of finding an algorithm that is simultaneously parallel, load balanced, devoid of communication, insensitive to heterogeneity, and correct, remains open.

Acknowledgements. The authors acknowledge the many contributions of W. Kahan, especially to section 8.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide, Release 1.0*, SIAM, Philadelphia, 1992.

[2] ANSI/IEEE, New York, *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[3] ANSI/IEEE, New York, *IEEE Standard for Radix Independent Floating Point Arithmetic*, Std 854-1987 edition, 1987.

[4] M. Assadullah, J. Demmel, S. Figueroa, A. Greenbaum, and A. McKenney, *On finding eigenvalues and singular values by bisection*, Unpublished Report.

[5] J. Barlow and J. Demmel, *Computing accurate eigensystems of scaled diagonally dominant matrices*, SIAM J. Numer. Anal., 27 (1990), pp. 762–791.

[6] H. Bernstein and M. Goldstein, *Parallel implementation of bisection for the calculation of eigenvalues of a tridiagonal symmetric matrices*, Technical Report, Courant Institute, New York, NY, 1985.

[7] G. Blelloch, *Prefix sums and their applications*, School of Computer Science Technical Report CMU-CS-90-190, Carnegie Mellon University, November 1990.

[8] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK: A portable linear algebra library for distributed memory computers - Design issues and performance*, Computer Science Dept. Technical Report CS-95-283, University of Tennessee, Knoxville, March 1995, (LAPACK Working Note #95).

[9] J. Demmel, *Underflow and the reliability of numerical software*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 887–919.

[10] J. Demmel and W. Gragg, *On computing accurate singular values and eigenvalues of acyclic matrices*, Linear Algebra Appl., 185 (1993), pp. 203–218.

[11] J. Demmel, M. Heath, and H. van der Vorst, *Parallel numerical linear algebra*, In A. Iserles, editor, Acta Numerica, volume 2, Cambridge University Press, 1993.

[12] J. Demmel and W. Kahan, *Accurate singular values of bidiagonal matrices*, SIAM J. Sci. Statist. Comput., 11 (1990), pp. 873–912.

[13] J. Demmel and X. Li, *Faster numerical algorithms via exception handling*, IEEE Trans. Comput., 43(8):983–992, 1994. LAPACK Working Note 59.

[14] J. Demmel and H. Ren, *The instability and nonmonotonicity of the parallel prefix algorithm*, in preparation, 1994.

[15] I. S. Dhillon and J. W. Demmel, *A parallel algorithm for the symmetric tridiagonal eigenproblem and its implementation on the CM-5*, Unpublished Report.

[16] Y. Huo and R. Schreiber, *Efficient, massively parallel eigenvalue computations*, preprint, 1993.

[17] W. Kahan, *When to neglect offdiagonal elements of symmetric tridiagonal matrices*, Computer Science Dept. Technical Report CS42, Stanford University, Stanford, CA, July 1966.

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

[19] W. Kahan, *Analysis and refutation of the International Standard ISO/IEC for Language Compatible Arithmetic*, SIGNUM Newsletter and SIGPLAN Notices, 1991.

[20] S.-S. Lo, B. Phillipe, and A. Sameh, *A multiprocessor algorithm for the symmetric eigenproblem*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 155–165.

[21] R. Mathias, *The instability of parallel prefix matrix multiplication*, SIAM J. Sci. Statist. Comput., 16 (1995), pp. 956–973.

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

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

[24] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford, 1965.