An Efficient Tridiagonal Eigenvalue Solver on CM 5 with Laguerre's Iteration *

Ren-Cang Li and Huan Ren
Department of Mathematics
University of California at Berkeley
Berkeley, California 94720

May 1992

Computer Science Division Technical Report UCB//CSD-94-848, University of California, Berkeley, CA 94720, December, 1994.

Abstract

In this paper, we propose an algorithm for finding eigenvalues of symmetric tridiagonal matrices based on Laguerre's iteration. The algorithm is fullly parallelizable and has been parallelized on CM5 at University of California at Berkeley. We've achieved best possible speedup when matrix dimension is large enough. Besides, we have a well-written serial code which works much more efficient in pathologically close eigenvalue cases than an existing serial code of the same kind due to Li and Zeng.

1 Introduction

Laguerre's iteration turns out to be one of the best iterative methods to find zeros of polynomials. If the polynomial has only real zeros, Laguerre's

*This material is based in part upon work supported by Argonne National Laboratory under grant No. 20552402 and the University of Tennessee through the Advanced Research Projects Agency under contract No. DAAL03-91-C-0047, by the National Science Foundation under grant No. ASC-9005933, and by the National Science Infrastructure grants No. CDA-8722788 and CDA-9401156.

iteration guarantees to produce a sequence of approximations which converge monotonically to the desired zero. Moreover, the convergence is ultimately cubic provided the algebraic multiplicity of the desired zero is estimated correctly. Therefore, convergence to simple zeros is always cubic and at least linearly for multiple zeros.

This report addresses some implementation issues in programming Laguerre's iteration both serially and parallelly to compute eigenvalues of a symmetric tridiagonal matrix. Our contributions are

- Different *Divide-and-Conquer* strategy which we call *Cross-and-Recover* in contrast with Li and Zeng's *Split-and-Merge* [5], which offers better inclusion intervals.
- More efficient zero–finder code in contrast with Li and Zeng's [5]. We make our zero-finder go as fast as possible for pathological cases.
- Efficiently parallel implementation on CM 5. Speedup is almost linear.

We will make our contribution more concrete later.

2 The Basic Structure of the Algorithm

Let T be an $n \times n$ tridiagonal matrix

$$T = \begin{pmatrix} d_1 & e_1 & & & & \\ e_1 & d_2 & e_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & e_{n-2} & d_{n-1} & e_{n-1} \\ & & & & e_{n-1} & d_n \end{pmatrix}, \tag{1}$$

whose eigenvalues,

$$\lambda_1 < \lambda_2 < \cdots < \lambda_n,$$

are to be found. Assume, without loss of generality, all $e_j \neq 0$, i.e., T is unreducible. Let $k = \lfloor n/2 \rfloor$ and write

$$T = \begin{pmatrix} T_0 & e_{k-1} & \\ e_{k-1} & d_k & e_k \\ & e_k & T_1 \end{pmatrix}.$$

If, now, the eigenvalues of $\widehat{T}$, a submatrix of T with its kth row and column crossed out,

$$\widehat{T} = \begin{pmatrix} T_0 & \\ & T_1 \end{pmatrix}$$

are known as

$$\widehat{\lambda}_1 \leq \widehat{\lambda}_2 \leq \cdots \leq \widehat{\lambda}_{n-1},$$

Cauchy's interlacing theorem tells us precisely where each λ_j locates, i.e.,

$$\lambda_1 \leq \widehat{\lambda}_1 \leq \lambda_2 \leq \widehat{\lambda}_2 \leq \cdots \leq \widehat{\lambda}_{n-1} \leq \lambda_n. \tag{2}$$

Also we have

$$\widehat{\lambda}_1 - (|e_{k-1}| + |d_k| + |e_k|) \leq \lambda_1, \ \lambda_n \leq \widehat{\lambda}_{n-1} + (|e_{k-1}| + |d_k| + |e_k|) \tag{3}$$

by Weyl-Lidskii theorem. With (2) and (3), λ_j can be computed by Laguerre's iteration applying to the characteristic polynomial of T.

How do we get $\widehat{\lambda}_j$? The above procedure can be applied recursively until matrices become small enough to be solved easily. Pictorially, our algorithm has the structure as shown in Figure 1, in which we assume the submatrices at the very bottom have the same size.

3 Laguerre's Iteration

Let

$$f(x) = \det(T - xI). \tag{4}$$

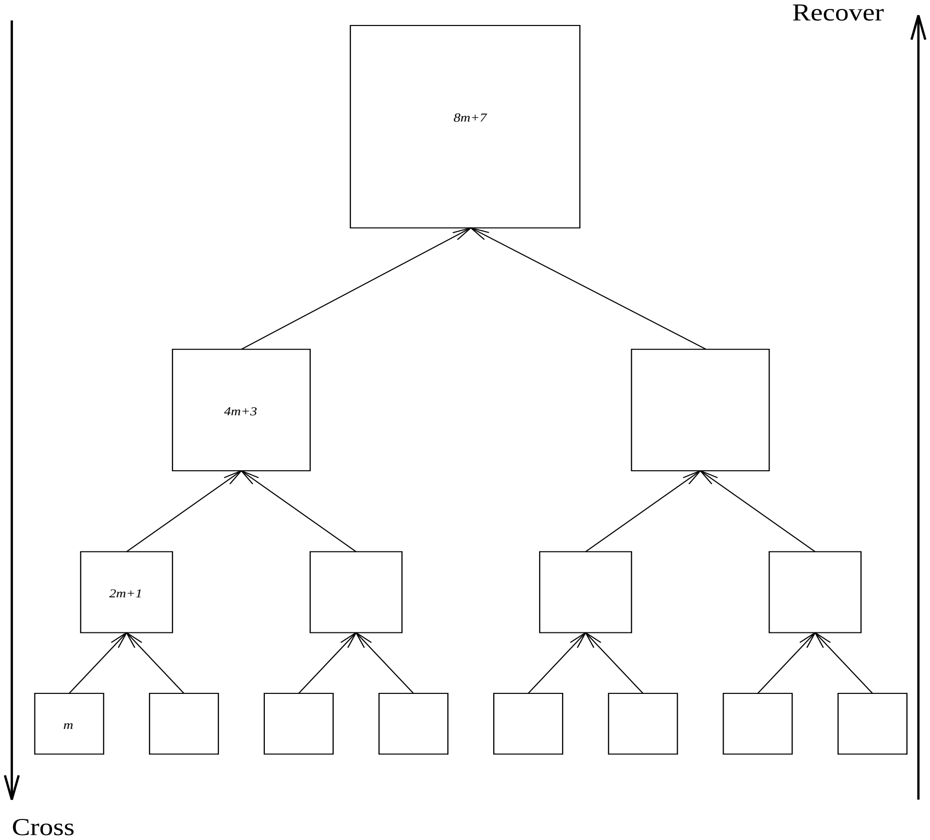

Figure 1: The *Divide-and-Conquer* Process

Laguerre's iteration ($L_\pm \equiv L_{1\pm}$) is defined as

$$L_{r\pm} \stackrel{\text{def}}{=} x + \frac{n}{\left(-\frac{f'(x)}{f(x)}\right) \pm \sqrt{\frac{n-r}{r}\left[(n-1)\left(-\frac{f'(x)}{f(x)}\right)^2 - n\left(\frac{f''(x)}{f(x)}\right)\right]}}, \qquad (5)$$

where r is a parameter to be chosen ideally to accelerate convergence. For our purpose, r is always set to be 1.

It is proved that

$$\boxed{\text{If } x \in (\lambda_j, \lambda_{j+1}), \text{ then } \lambda_j < L_-(x) < x < L_+(x) < \lambda_{j+1}.}$$

To apply Laguerre's iteration, one needs to know $\left(-\frac{f'(x)}{f(x)}\right)$ and $\left(\frac{f''(x)}{f(x)}\right)$. It turns out that they can be computed in a recursive way due to [5] (ref. to **Algorithm** DetEvl). As a by-product, the *neg_count* $\kappa(x)$ at x is computed also, which is the number of eigenvalues of T less than x. In a few cases, however, one may just need the *neg_count* at x for decision making as we will see later. In those cases, calling DetEvl to get *neg_count* is more expensive than necessary. So, instead, a call to **Algorithm** NegCount can be made.

3.1 Costs for Calling DetEvl and NegCount

The following tables present operation counts for one call to DetEvl or to NegCount.

Table 1: Costs

	$+$ or $-$	$\times$	$\div$
DetEvl	$6n-5$	$6(n-1)$	$3n-2$
NegCount	$2n-1$	$n-1$	n

In Table 1, it is assumed that $d_i - x$ and e_{i-1}^2/ξ_{i-1} for each i are computed once, and e_i^2 is computed whenever needed. By looking at DetEvl, one can see that there are rooms for rearranging computations to minimize total arithmetic operations.

Table 2: Costs (Optimized)

	$+$ or $-$	$\times$	$\div$
DetEvl	$6n-5$	$9(n-1)$	n
NegCount	$2n-1$	0	n

```
Algorithm DetEvl:
    input:    T, x;
    output:   −f'(x)/f(x) = η_n,  f''(x)/f(x) = ζ_n  and neg_count;
    begin DetEvl
        ξ_1 = d_1 − x;
        if ξ_1 = 0 then ξ_1 = e_1^2 ε_m^2; /* ε_m is the machine epsilon */
        η_0 = 0, η_1 = 1/ξ_1, ζ_0 = 0, ζ_1 = 0, neg_count = 0;
        if ξ_1 < 0 then neg_count = 1;
        for i = 2 : n
            ξ_i = (d_i − x) − (e_{i−1}^2/ξ_{i−1});
            if ξ_i = 0 then ξ_i = (e_{i−1}^2/ξ_{i−1}) ε_m^2;
            if ξ_i < 0 then neg_count = neg_count +1;
            η_i = [(d_i − x)η_{i−1} + 1 − (e_{i−1}^2/ξ_{i−1})η_{i−2}] / ξ_i;
            ζ_i = [(d_i − x)ζ_{i−1} + 2η_{i−1} − (e_{i−1}^2/ξ_{i−1})ζ_{i−2}] / ξ_i;
        end for
    end DetEvl
```

Figure 2: **Algorithm** DetEvl

```
Algorithm NegCount:
    input:    T, x;
    output:   neg_count;
    begin DetEvl
        ξ_1 = d_1 − x;
        if ξ_1 = 0 then ξ_1 = e_1^2 ε_m^2;
        η_0 = 0, η_1 = 1/ξ_1, ζ_0 = 0, ζ_1 = 0, neg_count = 0;
        if ξ_1 < 0 then neg_count = 1;
        for i = 2 : n
            ξ_i = (d_i − x) − (e_{i−1}^2/ξ_{i−1});
            if ξ_1 = 0 then ξ_i = (e_{i−1}^2/ξ_{i−1}) ε_m^2;
            if ξ_i < 0 then neg_count = neg_count +1;
        end for
    end NegCount
```

Figure 3: **Algorithm** NegCount

In Table 2, it is assumed that e_i^2 is pre-computed and for DetEvl each $1/\xi_i$ is computed for η_i, ζ_i and for $1/\xi_{i+1}$.

3.2 A Very Minor Error in Li and Zeng's Error Analysis

Li and Zeng [5] gave a detailed error analysis for DetEvl. It is correct generically, but it is not in one case. Assume we are working with the following arithmetic model

$$fl(x \circ y) = (x \circ y)(1+\delta),$$

where $\circ$ is one of the operations $+$, $-$, $\times$ and $/$, $fl(\cdot)$ denotes the floating point result of the corresponding operation, and $|\delta| \le \epsilon_m$, the machine epsilon. It was proved by Li and Zeng that

$$fl[f(x)] = fl[\det(T - xI)] = \det((T+E) - xI)(1+\gamma), \qquad (6)$$

where $-(3n-2)\epsilon_m + O(\epsilon_m^2) \le \gamma \le (3n-2)\epsilon_m + O(\epsilon_m^2)$, and

$$E = E_{LZ} = \begin{pmatrix} 0 & e_1\delta_1 & & \\ e_1\delta_1 & 0 & \ddots & \\ & \ddots & \ddots & e_{n-1}\delta_{n-1} \\ & & e_{n-1}\delta_{n-1} & 0 \end{pmatrix}$$

with $-2.5\epsilon_m + O(\epsilon_m^2) \le \delta_j \le 2.5\epsilon_m + O(\epsilon_m^2)$.

We claim *this error estimate is incorrect when* $d_1 - x = 0$. To see why, consider a T with all $d_j = 0$, n odd and $x = 0$. Were their error analysis correct, the $fl[\det(T)]$ produced by DetEvl would be zero. However, DetEvl never yields zero determinants! The error was made because of ignoring the case $d_1 - x = 0$. The correct E for the case $d_1 - x = 0$ is

$$E = \begin{pmatrix} e_1^2\epsilon_m^2 & e_1\delta_1 & & \\ e_1\delta_1 & 0 & \ddots & \\ & \ddots & \ddots & e_{n-1}\delta_{n-1} \\ & & e_{n-1}\delta_{n-1} & 0 \end{pmatrix}$$

4 The Basic Building Block

The kernel of our algorithm is to compute the eigenvalue λ_j of T inside an prescribed interval $[a, b]$ whose endpoints a and b are, initially, the eigenvalues of its two principal submatrices as we saw before and are updated as iteration goes on. Thus λ_j is the only eigenvalue of T inside $[a, b]$. To avoid some unpleasant situations, our code does the following at the beginning.

4.1 Preprocessing

```
max_off := max_{1≤i≤n−2} (|e_i| + |e_{i+1}|);
tol := (2.5 * max_off + |(a+b)/2|) * ε_m;
if b − a ≤ 2 * tol return λ_j := (a+b)/2;
call NEGCOUNT at (a+b)/2;
if κ((a+b)/2) = j then
        b := (a+b)/2; left_half := true;
else /*  κ((a+b)/2) < j */
        a := (a+b)/2; left_half := false;
end if
if left_half = true then
        tol := (2.5 * max_off + |a|) * ε_m;
        x_0 := a + 2 * tol; call DETEVL at x_0;
        if  κ(x_0) = j, then
                return  λ_j := L_−(x_0);
        else
                x_1 := L_+(x_0);
        end if
else /*  left_half = false */
        tol := (2.5 * max_off + |b|) * ε_m;
        x_0 := b − 2 * tol; call DETEVL at x_0;
        if  κ(x_0) < j, then
                return  λ_j := L_+(x_0);
        else
                x_1 := L_−(x_0);
        end if
end if
tol := (2.5 * max_off + |x_1|) * ε_m;
if |x_1 − x_0| ≤ tol, then
        if left_half = true then
                call NEGCOUNT at x_1 + tol;
                if κ(x_1 + tol) = j return λ_j := x_1 + tol;
        else /*  left_half = false */
                call NEGCOUNT at x_1 − tol;
                if κ(x_1 − tol) < j return λ_j := x_1 − tol;
        end if
end if
δ_1 := (b − a) ⁴√ε_m; δ_2 := (b − a) √ε_m; δ_3 := (b − a) ⁴√(ε_m³);
if |x_1 − x_0| > δ_1, go to (*);
```

```
if |x_1 − x_0| ≤ δ_3, then
        if left_half = true then
                call NegCount at x_1 + δ_3;
                if κ(x_1 + δ_3) = j then ,
                        b := x_1 + δ_3; x_2 := x_1 + δ_3/10; go to (*);
                end if
        else /*  left_half = false */
                call NegCount at x_1 − δ_3;
                if κ(x_1 − δ_3) < j then ,
                        a := x_1 − δ_3; x_2 := x_1 − δ_3/10; go to (*);
                end if
        end if
end if
if |x_1 − x_0| ≤ δ_2, then
        if left_half = true then
                call NegCount at x_1 + δ_2;
                if κ(x_1 + δ_2) = j then ,
                        b := x_1 + δ_2; x_2 := x_1 + δ_2/10; go to (*);
                end if
        else /*  left_half = false */
                call NegCount at x_1 − δ_2;
                if κ(x_1 − δ_2) < j then ,
                        a := x_1 − δ_2; x_2 := x_1 − δ_2/10; go to (*);
                end if
        end if
end if
if |x_1 − x_0| ≤ δ_1, then
        if left_half = true then
                call NegCount at x_1 + δ_1;
                if κ(x_1 + δ_1) = j then ,
                        b := x_1 + δ_1; x_2 := x_1 + δ_1/10; go to (*);
                else
                        x_2 := x_1 + δ_1; a := x_2;
                end if
        else /*  left_half = false */
                call NegCount at x_1 − δ_1;
                if κ(x_1 − δ_1) < j then ,
                        a := x_1 − δ_1; x_2 := x_1 − δ_1/10; go to (*);
                else
                        x_2 := x_1 − δ_1; b := x_2;
                end if
        end if
end if
(*)   continue ...  /* Loop Body for Laguerre's Iteration */
```

Although, Laguerre's iteration has very nice convergence properties in applying to unreducible tridiagonal symmetric matrix T, there are two difficult situations which need to be handled carefully in order to make iterations go as fast as it should.

4.2 Two difficult Situations: Detection and Solution

Suppose we are at x on the way to λ . For the sake of convenience, let's assume $x = a$. The following two situations are difficult ones.

Case (i): On the left of x, T has an eigenvalue which is terribly close to x, while x and λ are reasonably apart;

Case (ii): T has a few other eigenvalues on the right of λ and those eigenvalues look almost the same as λ from the view at x; or x is outside the region of cubic convergence of Laguerre's iteration for finding λ.

It is possible that two difficult situations happen at the same time. **Case (i)** is pointed by Kahan [4]. Both [4] and [5] mention **Case (ii)**. A popular treatment to **Case (ii)** is to estimate the "numerical" algebraic multiplicity of λ. However, the multiplicity could not be estimated correctly until x is sufficiently close to x. Proposed here, are effectively numerical detections of the two cases and their possible solutions.

Let $x_i = L_+(x_{i-1})$ $(x_0 = x)$.

The Case (i) can be easily detected by looking at $(x_2 - x_1)/(x_1 - x_0)$. The situation falls into **Case (i)** if the number is bigger than 1. One solution to **Case (i)** is to restart at new $x = (x_2 + b)/2$. It turns out this new x might be too farther to λ than the old x. Fortunately, our preprocessing described above prevents this from happening.

The Case (ii): To detect **Case (ii)**, again one only needs to look at $(x_2 - x_1)/(x_1 - x_0)$. The situation falls into **Case (ii)** if it is less than 1 but not so small to be thought that iterations will go at least superlinearly towards λ for the next few iterations. For an instance, use the following criterion:

$$\frac{1}{5} \leq q \stackrel{\text{def}}{=} \frac{x_2 - x_1}{x_1 - x_0} < 1. \tag{7}$$

As we mention above, one could then use $L_{r\pm}$ for r other than 1 to accelerate iterations. However, the right r is not so easy to get at least at the beginning. The following method works quite well. As a matter of fact, once we detect that (7) holds, we would expect iterations $x_i = L_+(x_{i-1})$ would continue to go for another few iterations at a similar rate, i.e., for the next few iterations, roughly,

$$\frac{x_{i+1} - x_i}{x_i - x_{i-1}} \sim q.$$

Note

$$\lambda = x_1 + (x_2 - x_1) + (x_3 - x_2) + \cdots + (x_{i+1} - x_i) + \cdots.$$

Since Laguerre's iteration goes monotonically upwards for the precent case, each $x_{i+1} - x_i$ in the summation is positive. Suppose the iterations $x_i = L_+(x_{i-1})$ would go linearly at the rate about q, say until x_{k+1}, and then begin superlinear (cubic) convergence. (k could be ∞.) Therefore

$$\frac{x_{i+1} - x_i}{x_2 - x_1} \sim q^{i-1},$$

which yields

$$\begin{aligned} \lambda &\sim x_1 + (x_2 - x_1)(1 + q + \cdots + q^{k-1}) + (\text{negligible} > 0) \\ &= x_1 + (x_2 - x_1)\frac{1 - q^k}{1 - q} + (\text{negligible} > 0), \end{aligned}$$

which means if we know k, we could make $x_1 + (x_2 - x_1)(1 + q + \cdots + q^{k-1})$ be the next approximation which should be much more accurate than x_3. A very simple way to find k is to try $k = 2^\ell$, where $\ell = 1, 2, \cdots, \infty$, sequentially, by calling NegCount. ℓ should be made as big as possible while maintaining $x_1 + (x_2 - x_1)(1 + q + \cdots + q^{k-1})$ still on the left of λ. One restriction on ℓ should be $q^k \geq \epsilon_m$, which gives, roughly,

$$\ell \leq \left\lfloor \frac{3.6 - \ln(-\ln q)}{0.69} \right\rfloor$$

for double precision. According to our experience, ℓ bigger than 6 could be set to ∞.

Roughly speaking, 2^ℓ calls to DETEVL are saved by $\ell + 1$ calls to NEG-COUNT.

5 Numerical Tests for Serial Implementation

We implement our algorithm based on what we have discussed both serially and parallelly. In this section we are going to present some numerical results on how efficient our serial implementation is in comparison with Li and Zeng's and `DSTEBZ`, the bisection routine in LAPACK for computing eigenvalues only. For reference, we also list times taken by `DSTEIN`, the inverse iteration routine in LAPACK for computing eigenvectors after calling `DSTEBZ`.

Our code was written in C. Li and Zeng's code, and routines from LAPACK as well, were written in Fortran. Our first example is glued Wilkinson matrices which have many pathologically close eigenvalues. To be more specific, the tridiagonal matrices are of form

$$T = \begin{pmatrix} W_{21}^+ & \beta & & & & \\ \beta & W_{21}^+ & \beta & & & \\ & \beta & \ddots & \ddots & & \\ & & \ddots & \ddots & \beta & \\ & & & \beta & W_{21}^+ & \beta \\ & & & & \beta & W_{21}^+ \end{pmatrix},$$

where W_{21}^+ is the well-known Wilkinson matrix whose diagonal entries are $10, 9, \cdots, 1, 0, 1, \cdots, 9, 10$ and off-diagonal entries all 1, β a parameter. Bigger testing matrices can be obtained by gluing more W_{21}^+.

Table 3: Timing for Glued Wilkinson Matrices with $\beta = 10^{-5}$

n	21	105	210	315	420	525	630	735	840
Ours	0.04	0.62	2.48	4.84	9.07	12.6	19.6	25.0	33.1
Li & Zeng	0.03	1.25	4.23	8.66	13.7	22.0	27.4	39.8	43.9
`DSTEBZ`	0.25	1.09	3.25	6.45	10.6	15.7	22.2	28.6	36.5
`DSTEIN`	0.09	2.60	13.3	36.1	76.5	138	228	344	496

Table 4: Timing for Glued Wilkinson Matrices with $\beta = 10^{-10}$

n	21	105	210	315	420	525	630	735	840
Ours	0.04	0.45	1.58	2.09	5.02	6.87	9.94	12.2	15.8
Li & Zeng	0.03	1.16	3.22	7.39	9.13	18.1	19.7	32.4	26.3
`DSTEBZ`	0.25	0.73	1.81	3.19	4.90	6.87	9.01	11.6	14.4
`DSTEIN`	0.09	2.62	13.3	36.4	76.3	138	226	343	495

Table 3 and Table 4 indicates that our code handles pathologically close eigenvalue cases much better than Li and Zeng's. For $\beta = 10^{-5}$, our code runs a little bit faster than `DSTEBZ`. However, when $\beta = 10^{-10}$, `DSTEBZ` got more potential as n goes larger and larger because eigenvalues are extremely clustered together.

For random testing matrices, our code takes comparable times as Li and Zeng's code does as the following table shows. On the other hand, `DSTEBZ` is very slow.

Table 5: Timing for Random Matrices

n	21	105	210	315	420	525	630	735	840
Ours	0.04	0.63	1.93	3.81	6.24	9.20	13.0	16.6	21.3
Li & Zeng	0.04	0.47	1.81	3.98	5.81	9.69	14.4	13.9	22.7
`DSTEBZ`	0.19	2.44	9.21	20.1	35.3	54.6	77.9	105.6	138
`DSTEIN`	0.10	2.05	8.21	18.5	33.08	51.9	75.5	104	140

To make Tables 3 to 5 more visible, we plot Tables 3 and 4 into Figure 4. Figure 5 plots Table 5.

6 Parallel Implementation on CM5

We code our parallel version of the algorithm in Split-C. For the sake of convenience of our presentation, the notation $T(i:j)$ denotes the principle

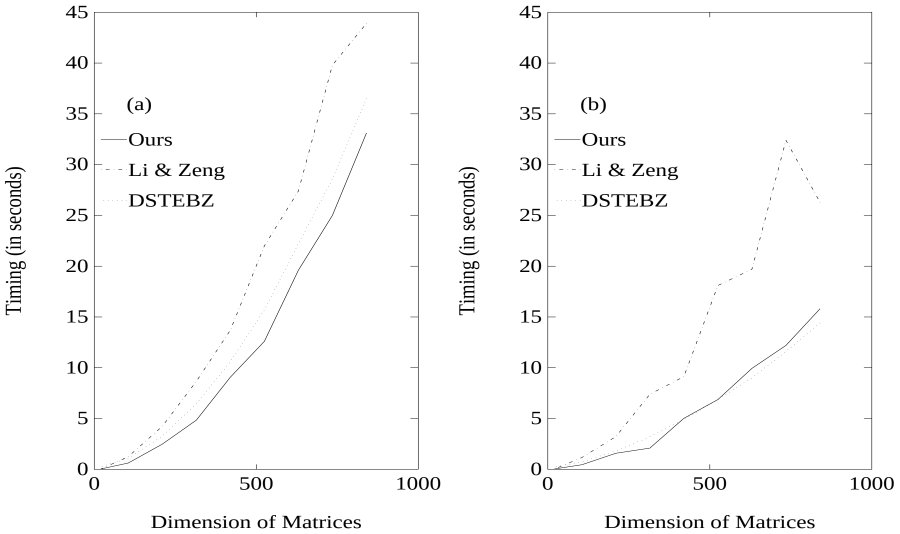

Figure 4: Glued Wilkinson Matrices: (a) $\beta = 10^{-5}$, (b) $\beta = 10^{-10}$

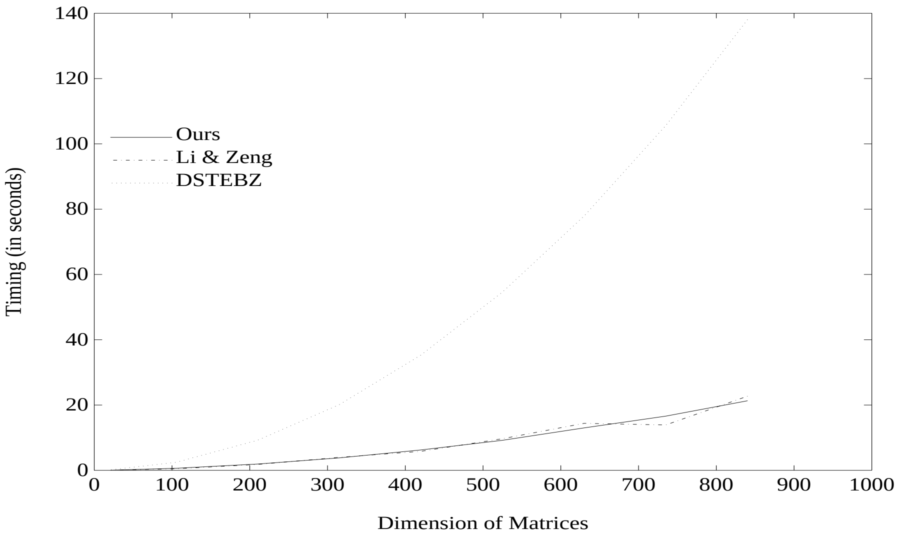

Figure 5: Random Matrices

submatrix consisting the intersection of rows i to j and columns i to j, i.e.,

$$T(i:j) \stackrel{\mathrm{def}}{=} \begin{pmatrix} d_i & e_i & & & & \\ e_i & d_{i+1} & e_{i+1} & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & e_{j-2} & d_{j-1} & e_{j-1} \\ & & & & e_{j-1} & d_j \end{pmatrix}.$$

6.1 Outline of our Implementation

Let `PROCS` be the number of processors. It is a power of 2 (`PROCS` $= 2^5 = 32$ for `rodin` and `PROCS` $= 2^6 = 64$ for `rodin3`).

Let's begin with an example: $n = 50$, `PROCS` $= 8$. First of all, T is decoupled into

$$\begin{array}{cccc} T(1:6), & T(8:13), & T(15:20), & T(22:26), \\ T(28:32), & T(34:38), & T(40:44), & T(46:50). \end{array}$$

Each of those small decoupled problems then can be solved serially by the 8 processors each of which takes care of one of them in order. We call this *Step 0*. *Step 1* is for $T(1:13)$, which will activate communications between `Processor 0` and `Processor 1`, and for $T(15:26)$, which will activate communications between `Processor 2` and `Processor 3`, and so on. After *Step 1*, *Step 2* is reached. All eigenvalues of T are computed after the completion of *Step 3*. Figure 6 gives a bird-view of how our algorithm goes on the example, where $i:j$ indicates a submatrix $T(i:j)$ on which one or more processors are working.

Assume `PROCS` $= 2^p$. Generally, our implementation flow is as described in Figure 7.

6.2 Implementation Details

6.2.1 Subtle Issues to Be Handled with Care

To implement our algorithm on CM5 efficiently turns out not to be an easy task. There are two major issues, among others, we have to deal with in

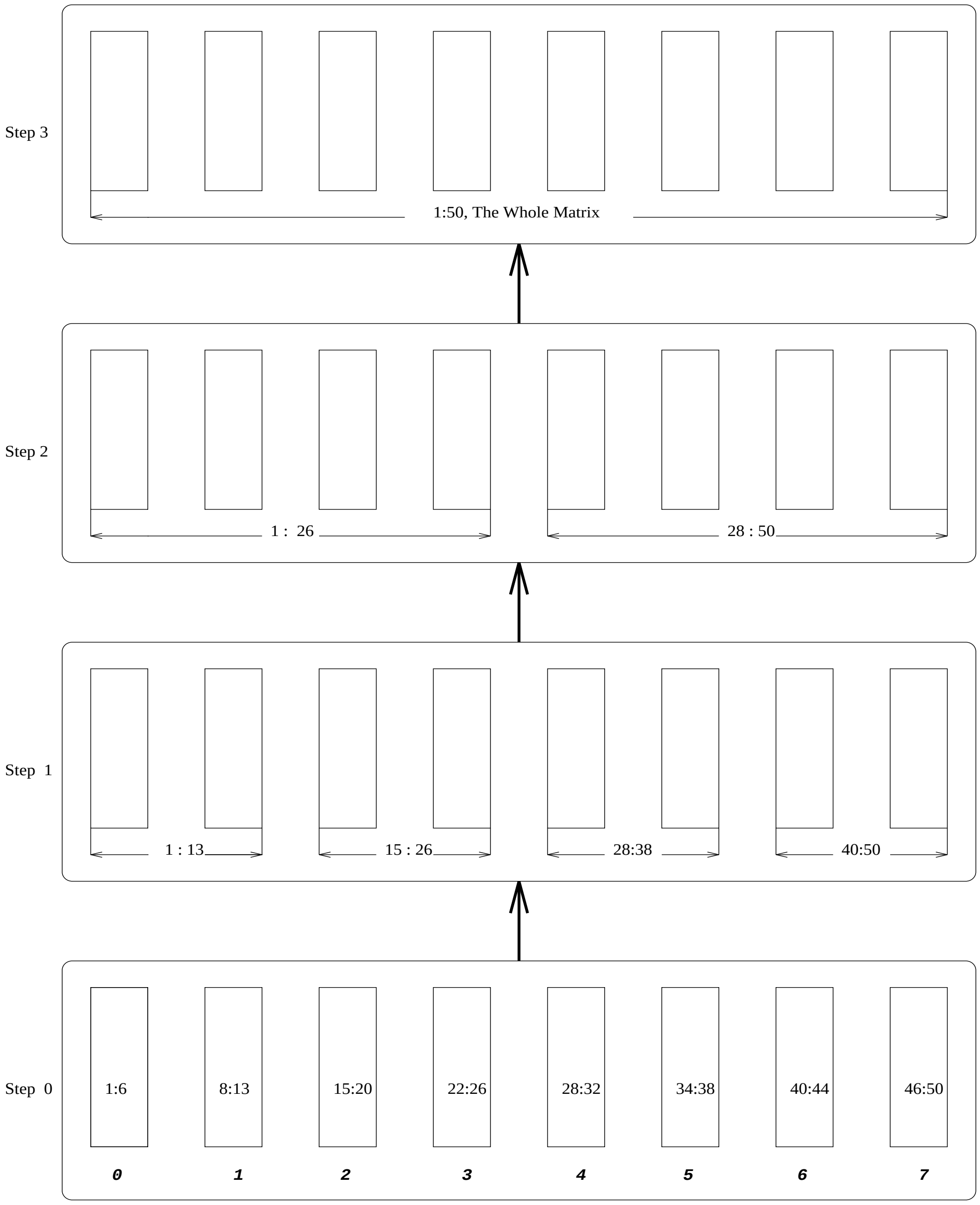

Figure 6: Algorithm Flow: An Example

```
q = ⌊(n-(PROCS-1))/PROCS⌋; r = [n - (PROCS - 1)] - q * PROCS;
Step 0:
        Processor i solves the eigenvalues of T(i_start : i_end) serially,
        where
                if  r = 0 then
                        i_start := i(q + 1) + 1, i_end := i(q + 1) + q;
                else /* r ≥ 1 */
                        for 0 ≤ i ≤ r - 1,
                                i_start := i(q + 2) + 1 and i_end := i(q + 2) + q + 1;
                        end for
                        for r ≤ i ≤ PROCS - 1,
                                i_start := r + i(q + 1) + 1, and i_end := r + i(q + 1) + q;
                        end for
                end if
end of Step 0
..................
Step j:
        for  i = 0 : (2^(p-j) - 1),
                Processors i2^j to (i + 1)2^j collaborate together to solve
                the eigenvalues of T((i2^j)_start : ((i + 1)2^j)_end);
        end for
end of Step j
..................
Step p:
        All processors collaborate together to solve the eigenvalues of T;
end of Step p
```

Figure 7: **Algorithm Flow**

order to achieve the best performance on CM 5 as we should.

- Minimizing Communications
  The first few things come to our mind is to minimize the number of messages to be communicated between processors, to maintain communication locally if possible, and to use fast communication tool (`bulk-get`) always; We do let every node processor do some redundant operations at the same time as *tradeoff* to reduce the number of messages, which turns out to be a good thing to do and affects negligibly on final speedups.

- Load-Balancing
  Load-balancing is another important issue which has heavy influence on final speedups if it is handled poorly. We will be more concrete about on how we distribute work among processors so that each processor would have roughly the same amount of work to finish with.

6.2.2 Memory Requirements

Since we are solving a symmetric tridiagonal matrix, the memory we need to store the matrix is $2n - 1$. To reduce communication, we let every node processor has a copy of the matrix. A global spreader array `glambda` is created across processors for the purpose of communication between processors and has size n in each node. Besides all this, a work space of size n in each node is needed. For work distribution, an array of length at most n of *structures*, each of which takes 2 double floating numbers and one integer, is also needed. Totally, about $6.5n$ size of double precision floating number space is used in each node processor.

6.2.3 Communication Details

Communications come in after the completion of each step and before entering the next step (ref. to Figures 6 and 7). We illustrate our points by the same example in Figure 6.

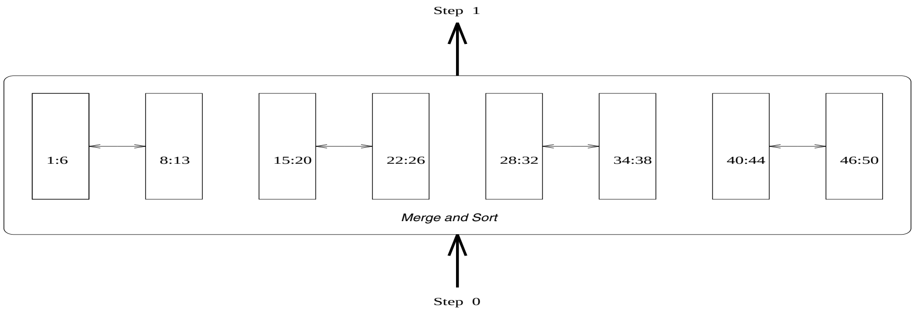

Figure 8: Communications: After *Step 0* and before *Step 1*

After *Step 0* and before *Step 1*: There is only one level of communications to deal with. As this stage, `Processor 0` owns (usually in ascending order) the eigenvalues of $T(1:6)$, and `Processor 1` owns those of $T(8:13)$. In order for the eigenvalues of $T(1:13)$ to be computed in *Step 1* by the two processors together, both processors have to know the eigenvalues of $T(1:6)$ and of $T(8:13)$, which serve as separators for the eigenvalues of $T(1:13)$. So what we do is "`Processor 0` get the eigenvalues of $T(8:13)$ from `Processor 1`, while *at the same time* `Processor 1` gets those of $T(1:6)$, and then let both processors put all eigenvalues together and do sorting by *Merge Sort*. (Hence, one of the two is doing redundant work!)" After sorting, `Processor 0` will decide its responsibility to find how many first few (roughly half, generally), eigenvalues of $T(1:13)$, while `Processor 1` will do the rest. Work distribution strategy will be discussed later. Similarly, `Processor 2` to `Processor 7` do their own part. Figure 8 displays what we just said, in which a sign $\leftrightarrow$ indicates communications between the corresponding two processors.

After *Step 1* and before *Step 2*: Two levels of communications are involved. As this stage, `Processor 0` owns the first part of the spectrum of $T(1:13)$, while `Processor 1` owns the rest; `Processor 2` owns the first part of the spectrum of $T(15:26)$, while `Processor 3` own the rest. In order for the eigenvalues of $T(1:26)$ to be computed in *Step 2* by the four processors

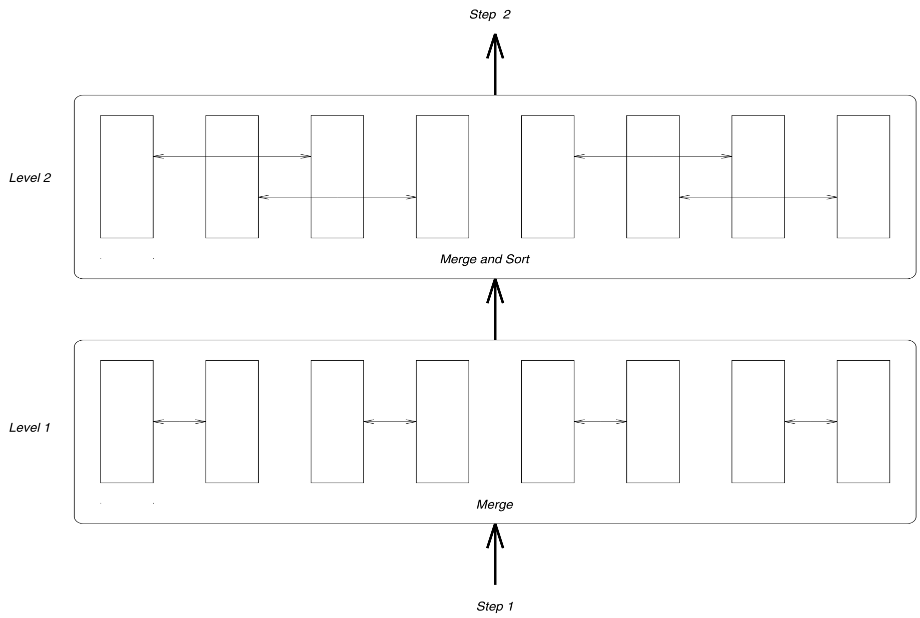

Figure 9: Communications: After *Step 1* and before *Step 2*

together, they have to know the eigenvalues of $T(1:13)$ and of $T(15:26)$, which, after sorted, serve as separators for the eigenvalues of $T(1:26)$. To this end, we do communications as shown by Figure 9. After *Level 1*, `Processor 0` and `Processor 1` both own the entire eigenvalues in proper order of $T(1:13)$, while `Processor 2` and `Processor 3` both own the entire eigenvalues in proper order of $T(15:26)$; after *Level 2*, each of the four processors has all eigenvalues (sorted) of $T(1:13)$ and of $T(15:26)$.

After *Step 2* and before *Step 3*: Three levels of communications are involved. As this stage, `Processor 0` to `Processor 3` together own the eigenvalues of $T(1:26)$, and each of the four processors owns a consecutive part. Also `Processor 4` to `Processor 7` together own the eigenvalues of $T(28:50)$, and again each of them owns a consecutive part. In order for the eigenvalues of T to be computed in *Step 3* by all processors together, they have to know the eigenvalues of $T(1:26)$ and of $T(28:50)$, which, after sorted, serve as separators for the eigenvalues of T. To this end, we do communications as

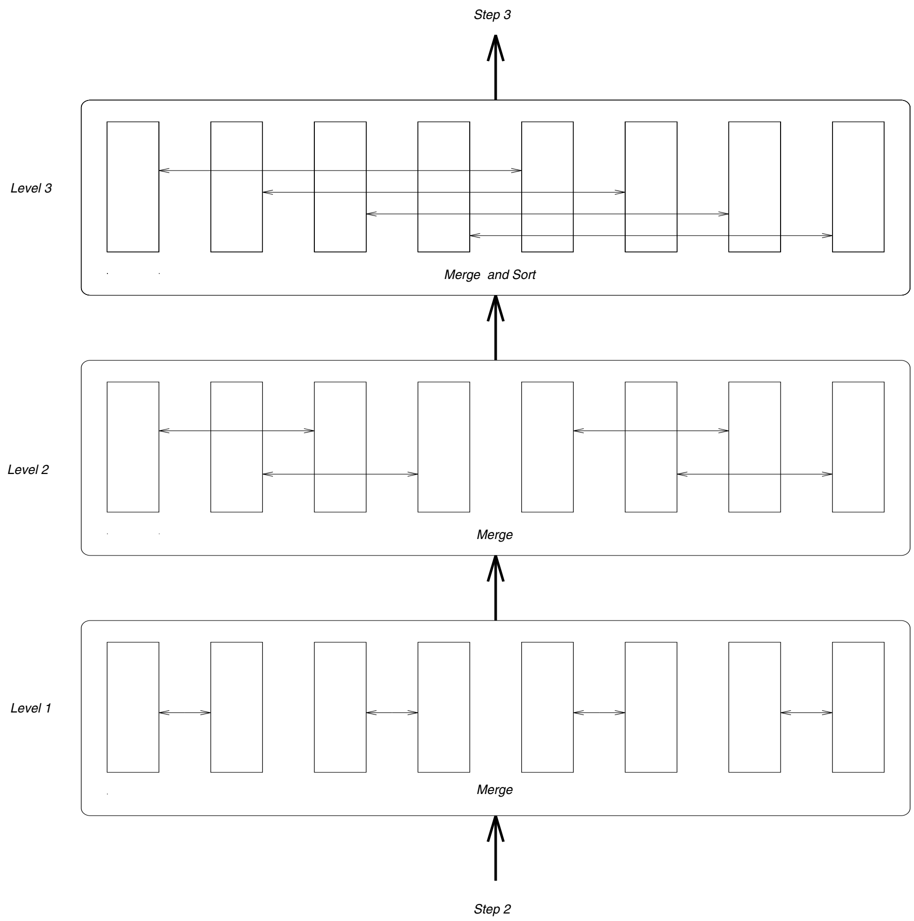

Figure 10: Communications: After *Step 2* and before *Step 3*

shown by Figure 10. After *Level 2*, each of `Processor 0` to `Processor 3` owns the entire (ordered) eigenvalues of $T(1:26)$, while each of `Processor 4` and `Processor 7` owns the entire (ordered) eigenvalues of $T(28:50)$; after *Level 3*, each of the eight processors has all eigenvalues of $T(1:26)$ and of $T(28:50)$.

Generally, there are j levels of communications after *Step* $j-1$ and before *Step* j. At each level, all messages can be passed simultaneously as all processors are connected by fat tree wires. Each message, hopefully, is of equal length. Thus there would be no processors finish message passing earlier than any others and become idle.

6.2.4 Task Distribution

Right before each step and after the last level of the very last communication before entering the next step, each processor owns two sets of eigenvalues each of which is in proper order, and they together form separators for the eigenvalues of the current (sub)matrix to be found by relevant processors in the coming step. Before getting into the step, there are two things to be done:

1. Sort the two sets of eigenvalues into a set of numbers in proper order; We use *Merge-Sort* to fulfill this task since eigenvalues in each set are already in proper order. Each relevant processors do the same sorting actually. Those redundant sortings do not affects overall speedups as they are cost marginally.

2. Tell cheaply which part of the spectrum of the current (sub)matrix each relevant processors are going to compute while maintaining load-balancing.

To deal with the second one, we again let every relevant processor does the same thing in trading for less communications. After each processor finishes sorting, we let them scan the sorted eigenvalues which are, as a matter of fact, separators for the eigenvalues to be found, and do the following

If two consecutive separators are close enough, an new eigenvalue is found instantly. Put the eigenvalue into the corresponding position in `glambda`; otherwise an entry of an array of *structure* is created, where the *structure* has form

```
New {
      double left;
      double right;
      int index;
    }
```

Here `left` is the left end point of the interval contains the eigenvalue we want to compute and `right` is the right end point, and `index` indicates which eigenvalue to be computed.

At the end, every relevant processor knows how many structures have been created, which, in turn, gives the number of eigenvalues needed to be computed by Laguerre iterations. At this point, it's easy to distribute the work evenly among relevant processors without introducing further communications. Doing so makes every processor compute roughly the same number of eigenvalues, and hopefully keeps load-balancing.

7 Numerical Tests on CM 5

Our parallel implementation on CM 5 is a great success in sense that the best speedup is achieved on all kinds of matrices we have tested. Table 6 lists times consumed on matrices whose eigenvalues are well-separated, where p denotes the actual number of processors participating computations. To be more specific, the diagonal entries of the matrices are $1, 2, \cdots, n$ and off-diagonal entries 1.

Table 6: Timing for Matrices with Well-Separated Eigenvalues

n	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$
8192	1776.87	890.67	446.52	224.61	112.79	59.02	31.54
4096	349.80	176.05	88.78	44.84	22.69	12.55	7.25
2048	90.68	45.52	23.07	11.76	6.12	3.81	2.39
1024	24.25	12.29	6.29	3.30	1.82	1.43	1.32
512	6.80	3.38	1.76	0.97	0.56	0.49	0.44
256	1.97	1.00	0.53	0.32	0.21	0.19	0.14

Table 7 displays times taken on tridiagonal matrices whose entries are uniformly distributed on $[-1, 1]$ subject to symmetry.

Table 7: Timing for Random Matrices

n	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$
8192	1863.37	964.26	470.83	233.53	120.28	61.86	32.31
4096	380.65	191.59	96.85	48.08	24.66	12.89	7.03
2048	103.92	52.58	26.81	13.10	7.08	3.85	2.55
1024	29.04	14.66	7.45	3.71	2.11	1.41	1.07
512	9.06	4.75	2.41	1.18	0.70	0.51	0.40
256	2.62	1.32	0.71	0.37	0.30	0.27	0.24

To see how much speedup we have gotten, we plot two figures showing speedups hiding in Table 6 and Table 7. Glued Wilkinson matrices are always an interesting test matrices. Table 8 and Figure 13 display how efficient our implementation is on this kind of matrices.

Table 8: Timing for Glued Wilkinson Matrices

n	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$
8400	2555.48	1387.49	729.14	367.27	191.73	101.58	55.55
4200	649.13	357.17	188.07	95.52	49.96	26.46	14.41
2100	177.03	97.26	51.84	25.55	14.36	7.93	4.35
1050	47.69	26.70	13.98	6.72	4.03	2.30	1.63

References

[1] Cuppen, J. J. M., A divide and conquer method for the symmetric tridiagonal eigenproblem, *Numer. Math.*, **36**(1981), 177–195.

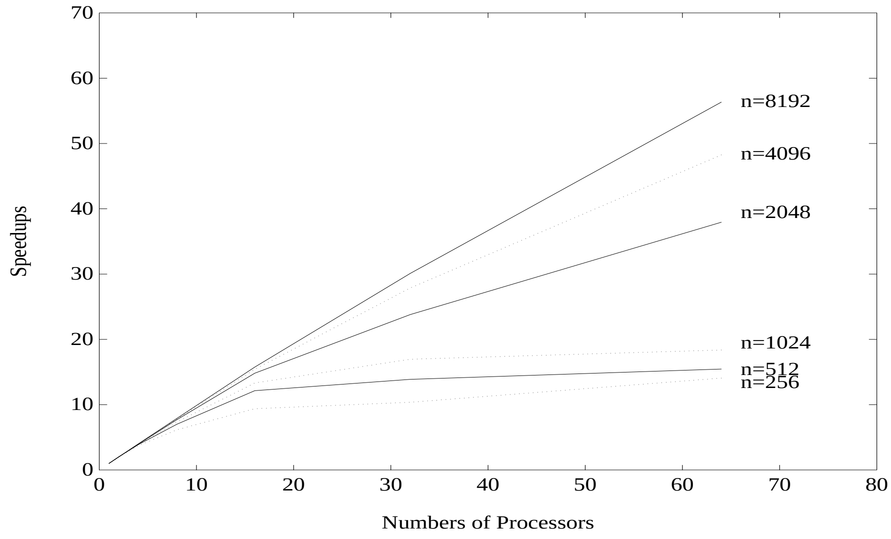

Figure 11: Speedups for Matrices with Well-Separated Eigenvalues

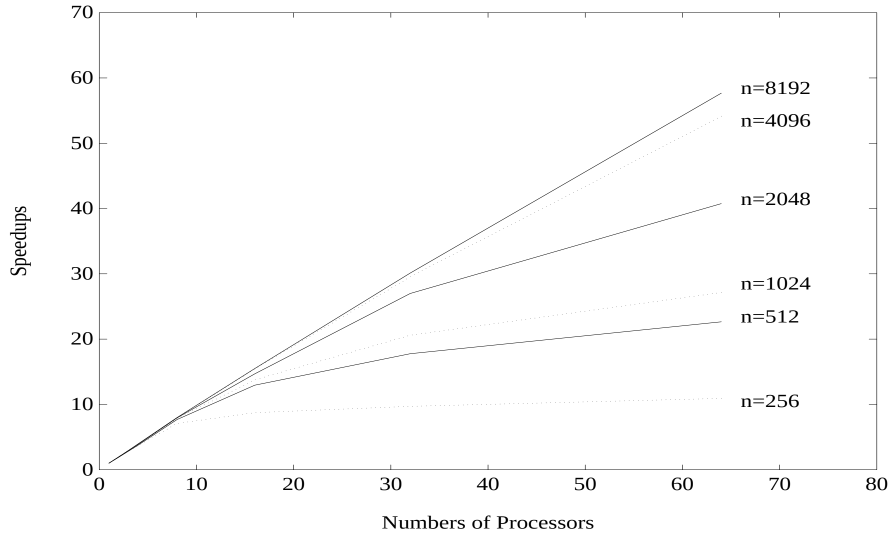

Figure 12: Speedups for Random Matrices

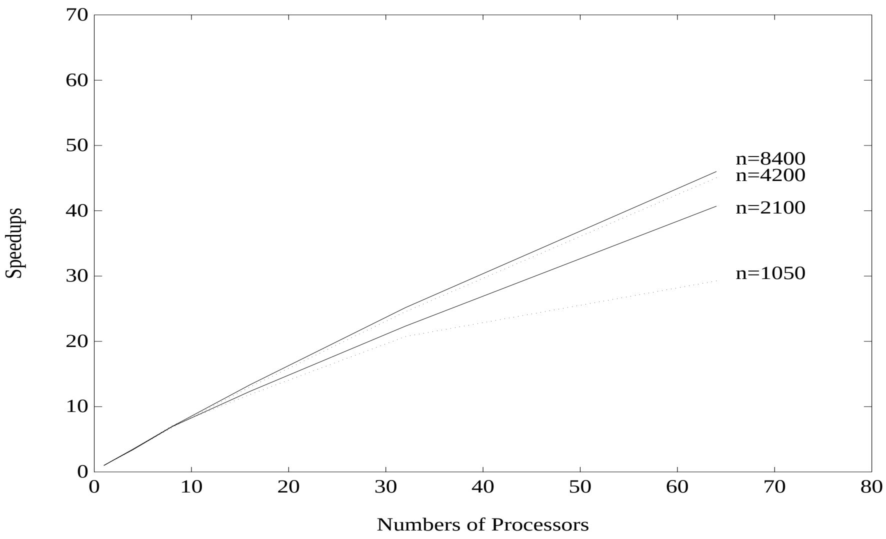

Figure 13: Speedups for Glued Wilkinson Matrices

[2] J. Demmel, M. T. Heath, and H. A. van der Vorst, Parallel numerical linear algebra, *Acta Numerica*, 1993.

[3] G. H. Golub, and Ch. F. van Loan, *Matrix Computations*, The Jhons Hopkins University Press, 2nd edition, 1989.

[4] W. Kahan, Notes on Laguerre's iteration, 1992

[5] T. Y. Li and Zhonggang Zeng, Laguerre's iteration in solving the symmetric tridiagonal eigenproblem—a revisit, preprint, 1992.

[6] B. Parlett, Laguerre's method applied to the matrix eigenvalue problem, *Math. Comp.*, **18**(1964), 464–485.