THE DIVIDE-AND-CONQUER METHOD FOR THE SOLUTION OF THE SYMMETRIC TRIDIAGONAL EIGENPROBLEM AND TRANSPUTER IMPLEMENTATIONS

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT AT CANTERBURY

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

By

Maria Paula Gonçalves Fachin

June 1994

F151936

To my family.

Abstract

This thesis describes a divide-and-conquer method for solving the symmetric tridiagonal eigenproblem and its parallel implementation on a transputer network.

The method was first developed by Cuppen based on previous ideas of Bunch et al. who showed how to compute the eigensystem of a diagonal matrix which is perturbed by a rank-one matrix.

We develop a new approach to the theory for arriving at the secular equation, the roots of which are the eigenvalues of a given symmetric tridiagonal matrix. We present new proofs for the convergence of the iterative method which finds these roots. We incorporate into the solution process an alternative way of computing any root when it is found to be very close to the right endpoint of the interval, generated from data for the unperturbed matrix, within which it is located. When this situation occurs the computation of the corresponding eigenvector is obstructed. Our reformulation circumvents this difficulty and, used in conjunction with the original means of computing the roots, produces generally more accurate results.

We have implemented the Divide-and-Conquer algorithm on a network of transputers organized in a rectangular grid. This organization works well, while remaining straightforward to program. Because of the way the Divide-and-Conquer method works, our implementation is best suited to a number of processors which is a power of two. The maximum number of transputers for which the program was tested was 16.

The Divide-and-Conquer method was found to be faster than the QL algorithm, even when executed in sequential mode. An explanation for this is suggested by complexity models, which we have developed. The method was also found to have good scalability when the number of processors is increased; that is we still obtain good speed-up values when more processors are used. A further feature is its superlinearity performance: efficiencies greater than 1 have been obtained. To some extent our complexity model explains this; it does not, however, take into account communication issues, which will be the subject of future work.

Acknowledgements

I would like to thank my supervisor, Dr. Barry J. Vowden, for his guidance and for his infinite patience during these years. I am profoundly indebted to him for all his support and for everything I have learned with him.

My warmful thanks to Dr. Timothy R. Hopkins who also helped me in many ways during my stay in Canterbury including many advices in the final written work.

I also wish to thank Rudnei for all his support and encouragement, helping me to keep "going" when it was not very easy to do so. Rudnei was always ready to help everytime I needed.

I must not forget to thank some friends who helped to make life enjoyable.

Thanks are also due to several persons in the Computing Laboratory who helped in technical matters, in special Lynn Maitland and Judith Broom.

I acknowledge the financial support given by the Conselho Nacional de Pesquisa e Desenvolvimento Científico e Tecnológico, Brazil, under grant 204217/89-0 and the Computing Laboratory, University of Kent at Canterbury, U.K. for allowing me to present a paper in a conference.

I am also very thankful to the Instituto de Matemática, Universidade Federal do Rio Grande do Sul, Brazil, for allowing me to undertake this research.

Finally I wish to thank all my family who continued to give me support even in the distance. I am grateful to my parents who helped me in many different forms throughout my stay in Canterbury.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The solution of the eigenvalue problem has been an object of study for many years and is the subject of much recent work. In fact, it had been well studied before 1965 when J.H. Wilkinson gave a thorough account in his book *The algebraic eigenvalue problem* [87]. With the advent of electronic computers and more recently, parallel computers, the kind of research done has changed its focus. Problems that were untreatable before computers existed, became solvable and problems for which the computational power available a few years ago did not allow their solutions in a reasonable amount of time, are now easily solvable. Whole books have been devoted to this subject, the one by Wilkinson as previously mentioned, and later ones by S. J. Hammarling [36], Gourlay and Watson [31] and B. Parlett [60]. The question is, why do we want to find the eigenvalues and eigenvectors of a matrix?

1.1 The diagonal form of a matrix

Usually when a matrix A is defined, it is said that it is a way of organizing some numerical information in rows and columns. This is true, but it is only half the story and does not tell what the purpose of a matrix is, or what a matrix does. We must think of a matrix as an operator, which has some effect over a vector x. That is, a matrix A is really a convenient way of organizing information that says how a vector x is modified by it. It is easy to see what a diagonal matrix (i.e. a matrix which has all its non-null elements in its main diagonal), does to a vector. If, for example, the matrix A is given by

$$A = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix},$$

its action on a vector x, which is represented by Ax, is to double its first component and reverse the second one, so that if x was originally in the first quadrant, it will be in the fourth quadrant after it is operated on by A and will be longer than before. Similar analysis can be carried out in three dimensions, but with larger dimensions, the physical interpretation becomes more difficult.

As was noted in the previous paragraph, it is easy to see what a diagonal matrix does to a vector. In a sense all symmetric matrices, i.e, matrices which have the same elements in "mirror" positions with respect to the diagonal, share this behaviour. For a symmetric matrix, we can always find a transformation that will turn it into a diagonal matrix. This transformation is the equivalent of finding the eigenvalues and eigenvectors of the original matrix A, and it allows us to treat A as though it were a diagonal matrix, making it much easier to work with. For a symmetric matrix of order n, say, it is always possible to find a set of n values λ, and corresponding non-zero vectors x, for which

$$Ax = \lambda x \tag{1.1}$$

and where the vectors x are pairwise orthogonal (and consequently linearly independent). The values λ are called the *eigenvalues* of A, and the corresponding vectors x are called the *eigenvectors*. The eigenvectors can be chosen to be unit vectors and thus all n eigenvectors form an orthonormal basis for the set of all vectors.

As the purpose of a matrix A is to operate on a vector, we observe that the eigenvectors of a matrix are a special kind of vector which remain in the same line after the application of A. If x is an eigenvector of A, when we compute Ax, the resulting vector will be in the same sub-space generated by x. It can change its length or reverse its sense of direction, but it will remain in the same "support" line.

The existence of these n eigenvalues and eigenvectors allows us to write equation (1.1) in a form that includes all of them, and shows the relationships between each λ_i and its corresponding eigenvector. Calling each eigenvector by q_i and forming the matrix Q, whose i^{th} column is q_i, and the diagonal matrix Λ whose diagonal elements are the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$, we can write (1.1) as

$$AQ = Q\Lambda \tag{1.2}$$

As the eigenvectors of A are orthonormal, the matrix of eigenvectors Q is an orthogonal matrix. Consequently, $Q^{-1} = Q^t$ and we can write the above equation (1.2), as

$$Q^tAQ = \Lambda \tag{1.3}$$

or as

$$A = Q\Lambda Q^t \tag{1.4}$$

Equation (1.4) is called the eigen-decomposition of A, and as it stands, it expresses A as the product of orthogonal matrices and a diagonal one. With this formulation, we can see that the action of the matrix A over a vector x will be first to change the basis in which the vector is originally written (usually the canonical basis) to a basis formed from eigenvectors of A and then to scale these coordinates using the eigenvalues.

1.2 Applications of eigenvalues and eigenvectors

Eigenvalues and eigenvectors appear in a wide range of contexts, ranging from engineering to economic applications, to statistics. Sometimes the matrix involved is symmetric, at other times unsymmetric, and some problems lead to a generalized eigenvalue problem $Ax = \lambda Bx$, that can be converted to the standard eigenvalue problem. In some situations all the eigenvalues and eigenvectors are required, in others only the eigenvalues are needed, in yet other cases just a few of the eigenvalues and eigenvectors are sought, and there are times in which just the largest or the smallest eigenvalue is wanted. Though we need to compute eigenvalues in many different kinds of situations, there are problems for which they are a valuable theoretic tool and allow us to analyse the type of solutions that can be obtained, and what the conditions are in which a certain desired solution appears. This happens frequently in the study of the stability of the numerical solution of partial differential equations (see, for example, [36]).

1.2.1 Computing powers of matrices

The fact that a symmetric matrix has n linearly independent eigenvectors means that these eigenvectors form a basis in an n-dimensional space. If the eigenvectors are denoted $q_1, q_2, \ldots, q_n$, we can then write any n-dimensional vector v as a linear combination of these vectors, $v = \alpha_1 q_1 + \alpha_2 q_2 + \cdots + \alpha_n q_n$.

With this formulation and the definition of an eigenvalue/eigenvector pair, the computation of $A^k v$ can be performed more easily and does not involve any product of matrices. For this, observe that, if we have $Aq = \lambda q$, then the following relations are satisfied

$$A^2 q = A(Aq) = A(\lambda q) = \lambda(Aq) = \lambda^2 q$$

and in general

$$A^k q = \lambda^k q.$$

Returning to the computation of $A^k v$, we have

$$A^k v = A^k(\alpha_1 q_1 + \alpha_2 q_2 + \cdots + \alpha_n q_n) = \alpha_1 \lambda_1^k q_1 + \cdots + \alpha_n \lambda_n^k q_n \tag{1.5}$$

In other words, the computation of a product between a power of a matrix and a vector is reduced to the computation of products between scalars and vectors.

This observation is useful when analysing the convergence of iterative methods. Many methods require the repeated application of a matrix to a vector, as described above. The aim of these methods is to make $A^k v$ go to zero as k increases. From equation (1.5) above, we can see that if any of the eigenvalues λ_i has absolute value greater than 1, the whole expression may diverge to infinity and we will not achieve our aim. That is the reason why so many methods require $|\lambda_i| < 1$ for $i = 1, \ldots, n$.

More details to do with the convergence of iterative methods can be found in [68] and also in [36], where the conditions for convergence of the *Gauss-Seidel* and *Jacobi* methods, for solving a linear system, are established. Other basic facts and properties of eigenvalues and eigenvectors can be found in the books already mentioned or other works in linear algebra and matrix computations, such as [74], [30], [73] or [42].

The necessity of having to compute the product between a power of a matrix A and a vector, as described above, arises in a variety of applications. In many cases, we want to compute vectors $x^{(k+1)}$ defined by

$$x^{(k+1)} = Ax^{(k)} + b \tag{1.6}$$

where A is an $n \times n$ matrix and b is a vector of dimension n. Clearly, we can compute $x^{(k+1)}$ iteratively by computing $x^{(1)} = Ax^{(0)} + b$, $x^{(2)} = Ax^{(1)} + b$ and so on, but if we want to know what happens when k is large, or tends to infinity, this is not a very practical approach and will give only an indication of what is happening. Observing that $x^{(k+1)}$ can be written as

$$x^{(k+1)} = A^{k+1}x^{(0)} + A^k b + A^{k-1}b + \cdots + Ab + b,$$

we can see that powers of A, must be computed. If we have an easier and less costly way of computing these powers, the amount of work needed will be greatly reduced. If A is a diagonal matrix with diagonal entries $\lambda_1, \lambda_2, \ldots, \lambda_n$, matrix A^k again will be a diagonal matrix with diagonal entries given by $\lambda_1^k, \lambda_2^k, \ldots, \lambda_n^k$. In this case, the computation of $x^{(k+1)}$ will be greatly simplified and the i^{th} entry of $x^{(k+1)}$ will be given by

$$\lambda_i^{k+1}x_i^{(0)} + (\lambda_i^k + \lambda_i^{k-1} + \cdots + \lambda_i + 1)b_i,$$

where $x_i^{(0)}$ is the i^{th}-entry of $x^{(0)}$ and b_i is the i^{th}-entry of b.

As usually A is not diagonal, we cannot use the approach just outlined directly, but if A can be diagonalized, i.e., if there is a matrix P such that $A = PDP^{-1}$, and D is a diagonal matrix, then a similar approach can be used. For, if $A = PDP^{-1}$, where D is a diagonal matrix whose components are the eigenvalues $\lambda_1, \ldots, \lambda_n$ of A, and P is an orthogonal matrix whose columns are the eigenvectors of A, then $A^k = PD^kP^{-1}$ and

$$P^{-1}x^{(k+1)} = D^{k+1}(P^{-1}x^{(0)}) + D^k(P^{-1}b) + \cdots + D(P^{-1}b) + P^{-1}b.$$

If $b = 0$, system (1.6) is called homogeneous and we have

$$x^{(k+1)} = Ax^{(k)} \tag{1.7}$$

Again, if we find the diagonal decomposition of A by finding its eigenvalues and eigenvectors, we can compute $x^{(k+1)}$ simply as

$$x^{(k+1)} = a_1\lambda_1^{k+1}q_1 + \cdots + a_n\lambda_n^{k+1}q_n$$

where the starting vector $x^{(0)}$ has been written as the linear combination of the eigenvectors $q_1, q_2, \ldots, q_n$, $x^{(0)} = a_1q_1 + \cdots + a_nq_n$ and where we pre-multiplied this expression for $x^{(0)}$ repeatedly by A, using the fact that $Aq_i = \lambda_i q_i$.

1.2.1.1 Practical applications where computing powers of matrices is necessary

Populations The model $x^{(k+1)} = Ax^{(k)}$ is relevant for the study of populations. Here we assume the population is divided into n age-groups. Each individual in a given age-group generates a specified number of new individuals and each individual in a group has a certain probability of being alive in the next period of time. We are, in this case, interested in knowing what the population growth will be. We can also include in the model other factors such as further species which act as predators to the first. Magid [52] gives a simplified example of a population of pre-historic animals, preyed upon by humans. We consider what the population of these animals and the predators (humans) is at a certain starting time such as, for example, the end of the last glacial era, and with the appropriate model and the transformation of the problem to the eigenvalue/eigenvector formulation, we can predict what the size of the population of animals is after, say, 1000 or 2000 years.

When studying a population model, we may be interested only in whether the population will grow, or decrease and then stabilize, or we may really want to compute the value of the vector $x^{(k)}$. In the first case we would need only to know if the absolute values of the λ_i's are greater or smaller than 1, whereas in the second case we would need all the eigenvalues and eigenvectors. Usually, in these kinds of problems, the matrices that arise are not symmetric. For more details see reference [52] cited above, and also Searle [67], where the following problems of population dynamics are investigated:

- model for a species of flour beetle – to predict what the size of the population of beetles will be after some period of time
- model for pulmonary tuberculosis – to determine the probability of an individual diagnosed in one stage of the disease changing to another stage by the next examination
- probability of an individual with a certain genotype having descendants with the same or another genotype

In all these problems, powers of an initial matrix need to be computed, and for this, the eigenvalue/eigenvector decomposition is an effective tool.

Searle also describes a study conducted by two other scientists, in which different hunting policies to cope with the prevalence of rabbits in New Zealand farms was investigated. Matrices representing fertility and survival rates of rabbits were used, and by computing the dominant eigenvalue of a product of these matrices, they could decide which was the best hunting policy to be used in order to eliminate the rabbits.

Smith [70] uses the computation of the eigensystem of a matrix to determine if two species of animals living in a shared habitat will reach an equilibrium and whether this equilibrium is stable or not. In this example, what matters is the presence of positive eigenvalues, which correspond to an unstable equilibrium.

Economic models The problem of computing $x^{(k+1)} = Ax^{(k)}$ also appears in economic models, where the economy is divided into sectors and activities, and in which we consider the costs of each activity and its financial impact within all of the sectors. This application is described, for example, in [52]. Another economic model set forth in [37] tries to forecast the behaviour of a market concerned with the selling of two competing products, and where some assumptions, based on market research, are incorporated about how the consumers behave in relation to the two brands involved.

Decay of isotopes Computations of powers of matrices through the computation of powers of eigenvalues are also applied in Smith [70], where the decay of artificial isotopes is described using repeated products between a diagonal matrix whose elements represent the rates at the various isotopes decay, and a vector with the initial quantities present of each element.

Colourblindness The necessity of computing powers of matrices also appears in the study of colourblindness in men, where some assumptions are made about how defective genes affect men and women and subsequent generations (see [70]). These characteristics are again represented by a matrix, and in order to know the proportion of persons affected by colourblindness in the n^{th} generation, we need to compute $x^{(n)} = M^n x^{(0)}$, where M is the 2×2 matrix which determines how the next generation will be affected, and $x^{(0)}$ represents the number of individuals who have a defective gene at instant t_0. To compute the product $M^n x^{(0)}$, again the eigenvalues (λ_1, λ_2) and eigenvectors (v_1, v_2) of M are computed. The initial vector $x^{(0)}$ is written as a linear combination of the eigenvectors and the vector $x^{(n)}$ is computed as $\alpha_1 \lambda_1^n v_1 + \alpha_2 \lambda_2^n v_2$. In this example, the positive eigenvalue(s) determine the proportion of the population with colourblindness, as negative eigenvalues makes the contribution of the factor, in which they participate, go to zero when n is large.

1.2.2 Computing eigenvalues and eigenvectors in statistics

In statistics, we are often interested in the relationships among the different measurements taken to analyse a certain multivariate phenomena. If we define the matrix X, as a matrix containing these measurements, in such a way that entry x_{ij} represents the measurement of the j^{th} variate for the i^{th} individual, the columns of X represent the variates, and the rows of X represent the different individuals. We can then form the matrix X^tX, in which each element measures the degree of association between two variates. When the matrix X is large, it may be difficult to analyse the different interrelationships between the variates simply by examining matrix X^tX. The analysis is often easier after decomposing this association matrix into its eigenvalues and eigenvectors.

Basilievsky [6] reports on the computation of the eigensystem of an association matrix and gives some examples, in specific contexts, where it is revealing. He describes the analysis of 33 measures of the human body for 256 school children between 8 and 12 years old. The authors of the experiment consider the first 5 of the 33 eigenvalues to be the most significant and interpret the results obtained with the help of the computed eigenvalues and eigenvectors, analysing how the measures are related to each other. For details of how this is done we refer to [6].

Basilievsky also gives other examples, in which the largest or the smallest eigenvalue is required. One such is the case of discriminant analysis where we want to compute a linear function $\alpha_1 X_1 + \cdots + \alpha X_n$ that separates objects into groups in some optimal fashion. The coefficients α_i are the elements of the eigenvector corresponding to the largest eigenvalue of a generalized eigenvalue problem. There are other applications described in the same reference, including orthogonal regression and analysis of time-series.

1.2.3 Systems of linear differential equations

Eigenvalues and eigenvectors also appear naturally when solving a system of linear differential equations

$$\frac{dx}{dt} = Ax(t) \tag{1.8}$$

Each component equation of the system (1.8) has the form

$$x_i' = a_{i1}x_1(t) + \cdots + a_{in}x_n(t)$$

where $x_1(t), \ldots, x_n(t)$ are scalar functions of t and each a_{ij} is a constant scalar. If the matrix A has n linearly independent eigenvectors $v_1, v_2, \ldots, v_n$ we can write each solution $x(t)$ as a linear combination of them

$$x(t) = c_1(t)v_1 + c_2(t)v_2 + \cdots + c_n(t)v_n$$

and then, using (1.8) and the definition of eigenvalue and eigenvector, we find

$$\begin{aligned} \frac{dx}{dt} &= x'(t) = c_1'(t)v_1 + \cdots + c_n'(t)v_n = Ac_1(t)v_1 + \cdots + Ac_n(t)v_n = \\ &= c_1(t)\lambda_1 v_1 + \cdots + c_n(t)\lambda_n v_n \end{aligned}$$

where $\lambda_1, \ldots, \lambda_n$ are the eigenvalues corresponding to the eigenvectors $v_1, \ldots, v_n$ respectively. But then, as $v_1, \ldots, v_n$ are linearly independent, we have

$$c_i'(t) = c_i(t)\lambda_i \tag{1.9}$$

for each i. Each such equation has solutions of the form

$$c_i(t) = C_i e^{\lambda_i t}$$

where C_i is some constant of integration, and then the general solution of 1.8 is given by

$$x(t) = C_1 e^{\lambda_1 t} v_1 + \cdots + C_n e^{\lambda_n t} v_n.$$

Another way to see this is to observe that if $x = e^{\lambda t} v$ is a solution of (1.8), then

$$\frac{dx}{dt} = \lambda e^{\lambda t} v = A e^{\lambda t} v$$

or, as $e^{\lambda t} \neq 0$

$$\lambda v = Av$$

This means that solving linear differential equation systems is intimately connected with finding the eigenvalues and eigenvectors of the associated coefficient matrix.

Asymptotic behaviour of the solution $x(t)$ We can predict the asymptotic behaviour of the solution $x(t)$ from a knowledge of the eigenvalues of A. As each solution $x(t)$ is a linear combination of functions of the form $C_i e^{\lambda_i t} v_i$, we can see that if λ is a real number, then

- if λ_i is negative, $C_i e^{\lambda_i t} v_i \to 0$ as $t \to \infty$
- if λ_i is positive, the modulus of $C_i e^{\lambda_i t} v_i$ gets large as $t \to \infty$.

The same kind of behaviour applies when λ_i is a complex number, but then we must analyse the sign of the real part of λ_i. So, if all λ_i are negative, the solution $x(t)$ goes to zero as $t \to \infty$, but if some λ_i is positive, $|x(t)| \to \infty$ as $t \to \infty$. In the first case the solution $x(t)$ is said to be *asymptotically stable* and in the second case $x(t)$ is said to be *unstable* (see [82]).

If $\lambda_1 > \lambda_i$ for $i = 2, \ldots, n$, we can divide the expression for $x(t)$ by $e^{\lambda_1 t}$ and observe that when $t \to \infty$, all the terms, with exception of the first one, go to zero. The solution $x(t)$ for large t becomes approximately equal to

$$x(t) = C_1 e^{\lambda_1 t}.$$

In other words, the behaviour of the solution $x(t)$ will be dictated by

$$C_1 e^{\lambda_1 t}$$

when t is large. This property may often be used to analyse the stability or asymptotic behaviour of solutions as $t \to \infty$, when we know that one of the eigenvalues is greater than the others. Illustration of the use of eigenvalues and eigenvectors in solving linear differential equations appears in many of the references which study the solution of the eigenvalue problem, as many phenomena can be modelled by a system of linear differential equations. Some of these references, are, Brebbia and Ferrante [11], Watkins [82], Barnett [4], Ortega and Poole [57], Gourlay and Watson [31], Hammarling [36], Strang [74], Hlawiczka [39], Jennings [42], Stewart [73] and Saad [63].

1.2.4 Situations where linear differential equations arise

1.2.4.1 Chemical reactions

Magid [52] gives as an example, the analysis of chemical reactions. When a reaction involves several substances both as reactants and products, we must assume that the amount of each substance present at a given time, depends on the amounts of the others. In Magid's example there are three substances a, b and c, which are such that a and b can produce c, a and c can produce b and b can make some of c to be converted to a. With assumptions concerning the rates of change of the quantities of a, b and c, the progress of the reaction is governed by

$$x'(t) = \begin{bmatrix} 0 & 0.5 & 0.75 \\ 0.5 & -1 & -0.25 \\ 0.75 & -0.25 & 0 \end{bmatrix} x(t) = Ax(t) \tag{1.10}$$

where $x(t) = [x_1(t), x_2(t), x_3(t)]$ represent the quantities of substances a, b and c at instant t. The initial quantities of each substance, i.e., the quantities of a, b and c at $t = 0$ given respectively by $x_1(0)$, $x_2(0)$ and $x_3(0)$ are written as linear combination of the corresponding components of the eigenvectors and then the general solution is given by $x(t) = \alpha_1 e^{\lambda_1 t} v^{(1)} + \alpha_2 e^{\lambda_2 t} v^{(2)} + \alpha_3 e^{\lambda_3 t} v^{(3)}$. In this example, two of the eigenvalues of A are negative, so that for large t, the solution is governed by the positive eigenvalue and corresponding eigenvector. For large t, the amount of substances a, b and c are then given approximately by $x_1(t) = \alpha_1 v_1^{(1)} e^{\lambda_1 t}, x_2(t) = \alpha_1 v_2^{(1)} e^{\lambda_1 t}, x_3(t) = \alpha_1 v_3^{(1)} e^{\lambda_1 t}$, where $v_1^{(1)}$ is the first component of vector $v^{(1)}$, $v_2^{(1)}$ is the second component and so on.

1.2.4.2 Mechanical systems (1)

This application studies the movements of n objects in space which are interacting with each other. In these cases, we usually consider the accelerations of the objects given by $x''(t)$ and the time evolution is governed by a differential equation system of the form

$$x''(t) = Ax(t) \tag{1.11}$$

System (1.11) can be reduced to system (1.8) by means of the introduction of dummy variables. Setting $y(t) = [x(t); x'(t)]^t$, we have

$$y'(t) = \begin{bmatrix} x'(t) \\ x''(t) \end{bmatrix} = \begin{bmatrix} x'(t) \\ Ax(t) \end{bmatrix}$$

and then (1.11) is transformed into a differential equation system $y'(t) = By(t)$ with double the number of variables and equations where

$$B = \begin{bmatrix} 0 & I \\ A & 0 \end{bmatrix}$$

The eigenvalues and eigenvectors of B can be obtained from those of A. If the eigenvalues of A are given by $\lambda_1, \ldots, \lambda_n$, and the corresponding eigenvectors by $u_1, \ldots, u_n$, the eigenvalues of B will be the numbers $\mu_i, -\mu_i$, where $\mu_i^2 = \lambda_i$. Eigenvectors of B are then the vectors

$$v_i = \begin{bmatrix} u_i \\ \mu_i u_i \end{bmatrix}, \quad w_i = \begin{bmatrix} u_i \\ -\mu_i u_i \end{bmatrix}$$

1.2.4.3 Mechanical systems (2) – vibrating string

The problem of a *vibrating string* (for example, a violin string) [57] reduces in certain simple cases to an instance of the general problem we have just described. That is,

$$y''(x) = \lambda y(x);\ y(0) = 0;\ y(1) = 0.$$

We want to find values of λ for which there are corresponding nonzero solutions which satisfy the conditions above. For this simple problem, we can find explicit solutions which satisfy the boundary conditions, but if we change the problem above to

$$y''(x) = \lambda C(x) y(x);\ y(0) = 0;\ y(1) = 0$$

where C is a positive function, then it maybe not possible anymore to obtain explicit solutions. In this case we can approximate the solutions numerically by discretizing the interval [0, 1] into $n+1$ parts and replacing the second derivative by a difference quotient. The differential equation above is then reduced to solving the n difference equations

$$\frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1}) = \lambda C_i y_i$$

for $i = 1, \ldots, n$. In these difference equations $C_i = C(x_i)$, $x_i = ih$, $h = 1/(n+1)$, $y_0 = y_{n+1} = 0$ and y_i is an approximation to $y(x_i)$. Writing the system in matrix form, we have (see [57])

$$\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \lambda h^2 \begin{bmatrix} C_1 & & & \\ & C_2 & & \\ & & \ddots & \\ & & & C_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{1.12}$$

which is a problem of the form

$$Ay = \lambda By.$$

The problem of solving the differential equation $y''(x) = \lambda C(x)y(x)$ can, in this way, be reduced to solving a generalized eigenvalue problem.

1.2.4.4 Mechanical systems (3) – loading

Another problem in which the model leads to the solution of similar second order differential equation as the ones above, is that of predicting what happens to a column when it is subject to loading. In a simple situation when external forces are not considered, we can find the solutions analytically, as in the string case above, but when there are other forces involved, this is not possible, and the solution must be computed by some other method. One approach is similar to that described above, in which we discretize the problem and find approximate solutions.

1.2.4.5 Mechanical systems (4) – masses and springs

The problem where we have a certain number of masses joined by springs, also leads to the general problem (1.11) (see, for example, [15], [11] or [4]). If we have, for example, two masses m_1 and m_2 joined by one spring and connected to a fixed support by another spring, with coefficients of elasticity k_1 and k_2, respectively, and we want to analyse the way the masses move, we can model the situation by a system of the form $x'' = -Ax$ which can be transformed to a first order linear differential system $x' = Bx$, by defining $x_3 = x_1'$ and $x_4 = x_2'$ so that $x_3' = x_1''$ and $x_4' = x_2''$. In this case x_1 and x_2 are the displacements of masses m_1 and m_2 from their equilibrium position and x_3, x_4 are the velocities of these two masses, respectively. For this problem, considering $m_1 = m_2 = 1$ we have (see [4])

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \\ x_4' \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -k_1 & k_1 & 0 & 0 \\ k_1 & -(k_1+k_2) & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{1.13}$$

Again, there is a basis for the solutions of this equation of the form $x(t) = e^{\lambda t}v$, where λ and v are, respectively, eigenvalue and eigenvector of B.

Another approach to the problem of masses and springs builds in the observation that the solution of a second order linear differential equation of the type $x''(t) = Ax(t)$, is given by $x = v\sin\omega t$. This kind of solution is assumed, because the system composed by masses and springs tends to have an oscillatory nature. In the case of two masses and three springs (one joining the two objects and the other two linking the system to fixed supports), we have the equations (see [11])

$$m_1x_1'' + (k_1 + k_2)x_1 - k_2x_2 = 0$$

$$m_2x_2'' - k_2x_1 + (k_2 + k_3)x_2 = 0$$

where, x_1 and x_2 represent the displacements of the two masses measured from their unperturbed positions. The solutions of this system are given by $x_1 = v_1\sin\omega t$ and

$x_2 = v_2 \sin \omega t$, where v_1 and v_2 are the amplitudes of the vibration and ω is the frequency of the vibration. Instead of transforming the second order system to a first order system as we did before, we can work directly with the second order system and the two solutions above and transform it to an algebraic eigenvalue problem in order to find the v's and ω's. Differentiating the two solutions x_1 and x_2 and substituting the results in the differential equations above, we find

$$[(k_1 + k_2) - m_1\omega^2]v_1 - k_2 v_2 = 0$$

$$-k_2 v_1 + [(k_2 + k_3) - m_2\omega^2]v_2 = 0$$

Dividing the first equation by m_1 and the second by m_2 and rewriting in matrix form, we obtain the eigenproblem (see [11])

$$\begin{bmatrix} \frac{k_1+k_2}{m_1} - \omega^2 & \frac{-k_2}{m_1} \\ \frac{-k_2}{m_2} & \frac{k_2+k_3}{m_2} - \omega^2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = 0 \tag{1.14}$$

It is clearly seen that when $m_1 = m_2$ we obtain a symmetrical eigensystem.

The simple system described above, can be extended in practice to involve hundreds of equations and unknowns. This model can be used to represent a high building, where the masses are the floors and the springs are the supporting columns. In the same way it can represent an oil deposit under the sea, where there are alternating layers of limestone and shale. The shale is like a spring in the sense that it responds to pressure in an elastic way and the limestone, because it is inelastic, can be compared to the masses. The problem consists, then, in trying to learn how the system will move when a pressure like an explosion is applied at a certain location. An extended discussion of this example can be found in [52].

1.2.4.6 Mechanical systems (5) – vibration of structures

As Brebbia and Ferrante [11] describe, there are cases in which we are interested in finding the natural frequencies of some structure. These frequencies are the ones with which, once excited, the system would continue, in theory, to vibrate indefinitely. These frequencies may be determined by assuming that there are no external forces acting on the system. In this case, we arrive at a second order equation of the type $Ku + Mu'' = 0$, whose vector solution has components of the form $A_i \cos \omega t + B_i \sin \omega t$. The accelerations of the system are given by $u'' = -\omega^2 u$, where ω is the frequency of vibration. The differential equation to be solved then becomes $(K - \omega^2 M)u = 0$, which is a generalized eigenvalue problem. Each eigenvalue ω_i represents a natural frequency for the system, and the eigenvectors u_i are the shapes associated with these frequencies. This model represents, for example, a *simply supported prismatic beam* without any external loading. In this case K is the stiffness matrix and M is the mass matrix, both symmetric matrices.

Another interesting application of the model just described, is to the case of a *multi-storey building*. Again, we are interested in the natural frequencies of the building. The structure is assumed to vibrate only in the y-direction as indicated in Figure 1.1. We suppose

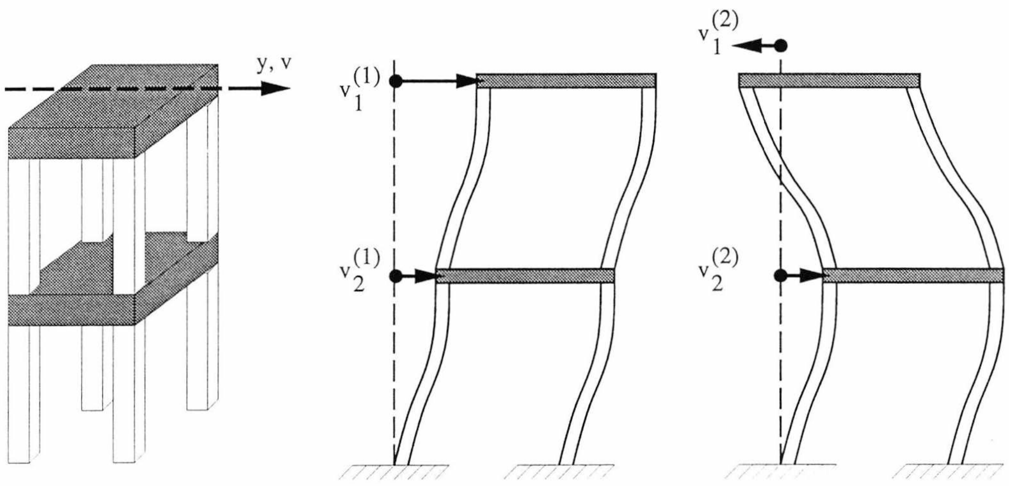

Figure 1.1: Multi-storey building and its possible displacements

that the stiffness of the floors is large in comparison with the stiffness of the columns and the floors just displace horizontally during motion. Following an example given in [11] we arrive again at a system of the type $Ku + Mu'' = 0$, where K, the stiffness matrix, has the form

$$\frac{EI}{l^3}\begin{bmatrix} 12 & -12 \\ -12 & 24 \end{bmatrix}$$

E is the modulus of elasticity, I is a moment of inertia and l is the length of the columns. M has the form

$$\rho V \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

where V is the volume of the slabs and ρ is the density of the material. In this example, the second order system of differential equations is reduced to solving the symmetric eigenproblem

$$\begin{bmatrix} 1-\lambda & -1 \\ -1 & 2-\lambda \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and the eigenvalues λ satisfy $\lambda = \rho V l^3 \omega^2 / 12EI$. In our example, the eigenvalues are approximately $\lambda_1 = 0.39$ and $\lambda = 2.62$, with corresponding eigenvectors

$$\begin{bmatrix} v_1^{(1)} \\ v_2^{(1)} \end{bmatrix} = \begin{bmatrix} 1.62 \\ 1.00 \end{bmatrix} \text{ and } \begin{bmatrix} v_1^{(2)} \\ v_2^{(2)} \end{bmatrix} = \begin{bmatrix} -0.61 \\ 1.00 \end{bmatrix}$$

The shape of the displacements given by each natural frequency is shown in Figure 1.1. Each component of the eigenvectors determines how each slab will be displaced in relation to its initial resting position. As Brebbia and Ferrante show, we can compute the internal energy of the structure for each pair (λ, v). This energy is smaller for the eigenpair corresponding to the smaller eigenvalue, which means that less energy is required to excite the structure in the λ_1 frequency. In general, what is then required is that the frequencies of any external force (e.g. earthquake waves) are very different from the first natural frequencies of the structure.

1.2.4.7 Electric circuits

The differential equations for an electric circuit are of the same kind as those for the spring-mass system and have solutions of the form $i_j = A_j \sin(\omega t)$ where i_j is the current in the j^{th} loop of the circuit (see [15]). Substituting a solution of this form, together with its derivatives, in the system of differential equations, we again reduce to the solution of an eigensystem. If the inductances L and the capacitances C are constant, we obtain a symmetric tridiagonal problem; for details see [15]. The eigenvalues provide the natural frequencies of the circuit and the eigenvectors the strengths of the currents generated within individual loops of the circuit, permitting a knowledge of the circuit's actual behaviour.

1.2.5 The Platzman problem

This problem arose in connection with a model developed by G.W. Platzman [62] to study the response of the oceans to lunar tidal forces. The resonant enhancement of tidal effects was well understood in small regions (like a bay), but to what extent this phenomenon affected the oceans in a global manner was largely unknown due to the difficulties associated with the computation of the normal modes of vibration of large expanses of water.

In his model, Platzman discretizes Laplace's "tidal" equations [62, page 203] using finite-differences. The model requires the solution of a symmetric eigenvalue problem, where the matrix involved is obtained from that discretization. The eigenvalues are the squares of the normal-mode frequencies and the eigenvectors describe comparative resonant tidal enhancements in different geographical locations.

1.2.6 Some other applications in brief

There are some other different areas of study where the necessity of computing eigenvalues, has arisen recently. In this section we will mention some of these cases, but briefly.

Manocha and Demmel, in [53], describe the computation of eigenvalues and eigenvectors for the determination of the points of intersection of two curves. This problem arises in computer graphics and geometric and solid modeling. They reduced the problem of intersection to the computation of the eigensystem of a certain matrix. They mention, that in this way they obtain a method more efficient and robust than the more usual methods of finding these intersection points. The difficulty with existing methods for computing these intersections, is that for curves with degree higher than four, the performance degrades due to numerical instability. The computations of the intersection points involves, in other methods, the computation of the roots of polynomials, and this is well known to be an ill-conditioned problem for polynomials having large degree. Manocha and Demmel's method involves the computation of a matrix determinant, but they construct a numerical matrix based on this determinant and the intersection points can be found from the eigensystem of this matrix. By this means enhanced accuracy is obtained. In some cases, where one of the matrices involved in the problem is close to being singular, the problem of intersection is reduced to a generalized eigenvalue problem. The eigenvalues computed correspond to the pre-images of intersection points. Only the positive real eigenvalues (including zero) are of interest, together with the corresponding eigenvectors, but it is not easy to restrict the computation of the eigenvalues just to the interval of interest. The authors report on computing accurately the intersections of curves of degree 10 which amounts to solving an eigenvalue problem for a matrix of order 100, but they mention that it is possible to carry through the computation for higher degrees as well, since we can compute accurately the eigendecomposition of matrices with orders much larger than 100.

Camp et. al. [14] report on the computation of eigenvectors for the *domain-mapping problem* which arises when numerical meshes (used, for example, for finite-differences, finite-elements or finite-volumes) are to be distributed in a certain way suitable for the use of parallel architectures. This problem consists in finding the best partitioning among the processors, of a certain number of tasks. Ideally, the work executed in each processor should be about the same. Additionally, the interprocessor communication should be minimised. This optimum, or the means of achieving it, is not easily arrived at and several approaches have been tried without much success. Some algorithms give good results, but are very expensive to compute. Others are simple, but in some instances give poor solutions. The authors have developed a method that reduces the problem to that of finding the best partitioning of a graph into sets with equal number of vertices (a vertex represents an individual computation) in such a way that there is a minimum number of edges (representing communication needs) connecting the sets. Additionally, these sets should not be assigned to processors in such a way that communication is required between distant processors. Camp et al. use two or three eigenvectors of a certain matrix, to partition the graph into four or eight pieces at each stage, applying this partition recursively to each of the pieces.

1.2.7 Personal communications to the author

We have also obtained information about other applications for the eigenvalue problem, after a posting made to the numerical analysis community via tha NA-NET digest. A brief summary of the answers we received is given in what follows.

G. Watson [84] reports on the computation of eigenvalues of tridiagonal matrices in *tight-binding models of the solid state.* Some examples mentioned by him are *Bloch electrons in a magnetic field, longitudinally modulated magnetic materials* and *modulated spring models of incommensurate crystals.* Tridiagonal eigensystems also arise in the *field of localisation in disordered systems* where simplified models are considered. The interest in this case can be either in the asymptotic behaviour of the eigenvectors or in the distribution of the eigenvalues, which is the case in *quantum chaos.*

B. Sumner [75] works with the equation

$$\left[\frac{d^2}{dz^2} + k^2(z)\right] e_i(z) = p_i^2 e(z),$$

which generates a tridiagonal system after it is discretized. For this problem, the N largest eigenvalues and corresponding eigenvectors are required.

A. Iserles [41] gives as an example, the computation of the eigenvalues of *Schrödinger operators*, which involves, in most cases, large tridiagonal matrices (a few million variables). This is required in various branches of physics.

H.A. van der Vorst [79] also mentions the solution of the *Schrödinger equation* and gives as another example, the computation of the eigensystem of symmetric matrices that arise from particle iterations or the discretization of the wave equation.

M. Trott [78] works with the modelling of nanoelectronic devices and needs to solve the one dimensional *Schrödinger equation*, which is transformed into a symmetric tridiagonal matrix by finite-difference methods. The interest here is to find the eigenvalues and corresponding eigenvectors which lie in a certain interval.

L. Watson [85] mentions that in structural optimization, several eigenproblems need to be solved as the process of optimization proceeds.

J. Warsa [81] describes the need for eigenvalue computations in the field of neutron transport for reactor analysis. In this case, a "critical" eigenvalue for a given system is sought. Usually this is the smallest eigenvalue of a certain system. This eigenvalue represents the minimum energy necessary to maintain the fission chain reaction.

A. McIntosh [55] reports, in her research, the problem of computing the complete eigensystem of symmetric, positive definite covariance matrices, which can be from order 300 up to 1800.

In the exposition above, we have tried to give some examples to motivate the study and solution of eigenvalue problems. We have therefore not been able to give all the details for each problem, but simply aimed to show where eigenvalue problems originate. For a more complete description of the examples discussed and for some other applications we refer the reader to the bibliography mentioned and to related literature. It must also be stressed that the applications mentioned here, were the ones we had access to, but many others uses for the eigenvalue problem certainly exist.

1.3 Solving the symmetric eigenproblem

As we have seen, the need to compute the eigenvalues and eigenvectors of a matrix arises in many contexts, often of a quite practical kind. Frequently the matrices involved are large and the solution of an eigenproblem can be extremely expensive in computational terms. But in recent years, the advances in computing power, with the advent of parallel computers, has motivated users in many fields of applications to attempt to solve large scale eigenproblems.

Since the 1960s, the algorithm of choice for solving this problem is the QR algorithm, which has been extremely optimised in numerical terms as a sequential algorithm. In spite of this optimisation, the QR algorithm can still take a long time to compute all the eigenvalues and eigenvectors of a large matrix, as its order of complexity is n^3.

In sequential mode, much research has already been done without any major improvement over the methods already developed. It appears that a better approach is to look for algorithms which will be effective when implemented on parallel systems, as large scale eigenproblems demand high computational effort. However, the optimised version of the QR algorithm does not lend itself to efficient parallelisation, mainly due to the intrinsic sequential nature of its operations. Thus, we are confronted with the need for developing parallel algorithms for the symmetric eigenproblem.

Many traditional algorithms can be used for this purpose, including *Lanczos's*, *Davidson's*, *bisection*, the *power method*, *inverse iteration* and *Rayleigh quotient iteration*. Parallel versions of these algorithms can be obtained by the parallelisation of one or more of the linear algebra operations required in the computation (for instance, a matrix-vector product).

Other algorithms, including *Jacobi's*, *divide-and-conquer* techniques and *homotopy* algorithms, have an inherent parallelism that can be exploited. A brief description of some of these methods and some others suitable for parallel computations can be found in the Chapter 2.

1.3.1 Symmetric matrices

In this thesis we are concerned strictly with the symmetric eigenvalue problem. Even though the unsymmetric case is also important, it does not have the nice characteristics that the symmetric case possess. In addition, the method we are interested in studying in more detail is specific to symmetric matrices and indeed to tridiagonal ones.

Symmetric matrices have some advantage over unsymmetric ones as far as the solution of the eigenvalue problem is concerned. For symmetric matrices the problem is well conditioned in relation to the eigenvalues (see [87]), which means that if there is some error in the input matrix (the matrix for which we wish to find the eigenvalues), the absolute error in the computed eigenvalues will not be greater than the error introduced in the initial matrix. In addition, the eigenvalues of an hermitian matrix and in particular those from a real symmetric matrix are always real values. The fact of a matrix being symmetric also implies that some operations can be simplified and when a reduction scheme (like Householder transformations) is applied to a dense/full matrix, this reduction leads to a tridiagonal matrix instead of a Hessenberg one as is the case when the initial matrix is unsymmetric.

1.3.2 Objective of this thesis

Our object of study is a Divide-and-Conquer method proposed first by Cuppen [17] and based on previous ideas of Bunch et al. [13], and its parallel implementation in transputers. It is a method very suitable for computations in parallel as the initial matrix/problem is partitioned into several sub-problems which can be solved independently. We discuss first the theory underlying the *Divide-and-Conquer* method, and later we describe how the implementation was performed and the results we achieved.

As already mentioned, the method works with real symmetric tridiagonal matrices, but if the given matrix is not tridiagonal, we can always reduce the matrix to this form, for example, by using a sequence of *Householder transformations*. This aspect is not studied here, but there are already parallel routines such as PDSYTRD in SCALAPACK [16] which accomplishes the transformation of a full symmetric matrix to a tridiagonal one.

1.4 Notations

The following notations and conventions will be used throughout the thesis. We assume that A is a symmetric matrix and x is a real vector.

- The norm used for vectors is the Euclidean or 2-norm. That is if $x = [x_1, \dots, x_n]$ then $||x|| = \sqrt{x_1^2 + \cdots + x_n^2}$.
- A corresponding subordinate norm will be used for matrices, that is, $||A||_2 = \max\{|\lambda_i|\}$, where the λ_i's are the eigenvalues of A
- e_j represents the j^{th} column of an identity matrix with dimension appropriate to the problem where it is being used. That is $e_j = [0, \dots, 0, 1, 0, \dots, 0]^t$ where the 1 is in the j^{th} position.
- Usually when we refer to a vector it will be a column vector. When we want to use a row vector it will be referred as the transpose of some column vector or we define it as a row vector explicitly. Sometimes it does not make any difference using the row or column form, but the form may be important if we want to define operations between vectors or between vectors and matrices: for example, $z^t z$ has a different meaning if z is a column- or row-vector.
- $A(i_1 : i_2, j_1 : j_2)$ means that we are referring to rows i_1 to i_2 and columns j_1 to j_2 of matrix A. A similar notation holds for vectors. This notation is taken from MATLAB [54] and makes the reference to blocks of a certain matrix particularly easy.
- The notation (a, b, c) represents a tridiagonal matrix with all diagonal elements equal to b and sub and super-diagonals elements equal to a and c respectively.
- The notation $\lfloor x \rfloor$ indicates that the real expression x is to be truncated to the largest integer value less than x, where x has a positive value.

- S_p indicates the speed-up when p processors are used and it is defined by $S_p = T_1/T_p$, where T_1 is the time taken by a certain algorithm in 1 processor and T_p is the time taken by the parallel algorithm (in p processors) we want to compare with the first one.

- Analogously, E_p is a measure for the efficiency of a parallel algorithm and is defined by $E_p = S_p/p$, where p is the number of processors used. For a more complete discussion of these two last measures, see Section 6.2.

Some other notations are defined in the text where they are used, otherwise, the conventions introduced here should be assumed.

1.5 Overview of the thesis

This thesis is organized as follows. In Chapter 2 we review some of the methods and algorithms available for the solution of the symmetric eigenproblem. The Divide-and-Conquer method of Cuppen is described in Chapter 3. In Chapter 4, we discuss the iterative approach used to solve the secular equation that arises in connection with the Divide-and-Conquer method, and also some relevant convergence results. Chapter 5 contains the description of an implementation of the method on a network of transputers. The results obtained with our implementation are given in Chapter 6. Finally, in Chapter 7 we provide some concluding remarks and present some ideas to be pursued in the future.

Chapter 2

Solving the symmetric eigenproblem: an overview of methods

In this chapter we give a brief description of some of the methods suitable for calculating the eigensystem of a symmetric matrix in parallel computing environments. We present results obtained by other authors who have implemented these methods. Our discussion reports on some of the other methods available for solving the eigenvalue problem in the parallel context, apart, that is, from Cuppen's Divide-and-Conquer method [17], which is the subject of this thesis. The range of our presentation does not attempt to be exhaustive nor, in any particular instance, to cover all the aspects of that method, but instead it describes the main results obtained concerning the suitability for parallel implementations, and for which kind of problems any method is most appropriate.

Some surveys of the area of parallel numerical algebra algorithms have already been published, for example that by Heller [38] in 1978 and, more recently, the paper by Gallivan et al. [26] in 1990. These authors discuss characteristics of parallel computers and aspects that must be taken into account when writing an algorithm for parallel computations. In their discussion they include methods, known at the time the surveys were written, for solving linear systems, singular value problems and also the eigenvalue problem. Heller mentions the Jacobi and QR methods for computing all the eigenvalues and eigenvectors of a symmetric matrix and their parallel implementations. While the Jacobi method is inherently parallel, the QR method is reformulated in such a way that linear recurrences arise, and these are to be solved in parallel. We give more details in the next sections. Gallivan et al. also mention the QR method, but only as the most usual way of computing the eigensystem of a matrix in sequential mode, noting, as we already have remarked, that the algorithm used for sequential computations does not offer much scope for parallel implementation. They describe the Divide-and-Conquer method devised by Cuppen [17], and identify it as an alternative for the QR algorithm, appropriate for parallel computers. They also cite a multisectioning algorithm of use when just some eigenpairs are required, and again describe Jacobi algorithms. Besides the two surveys cited above there are two more specific reviews. The first, by Berry and Sameh [7] treats parallel algorithms for the singular value and the symmetric eigenvalue problems, and describes and compares Jacobi methods and multisectioning. The other, by Watkins [83], analyses several methods for solving the eigenvalue problem and establishes relations between them. This latter is primarily concerned with theoretical aspects of the methods and does not address practical implementations. For these reasons we do not consider it further.

More recently, Edelman [23] has published a paper in which he discusses some issues relevant for the implementation of linear algebra algorithms in parallel computers and some applications of linear systems and eigenvalue problems. He discusses the use of Jacobi algorithms and briefly describes a method that is not well known outside the computational chemistry community. This is Davidson's method, which is classified as a hybrid between the methods of Lanczos and inverse iteration, and is used to compute the smallest eigenvalues of very large matrices (with order in the millions).

2.1 Jacobi's method

The Jacobi method is one of the oldest algorithms employed for computing the eigensystem of a symmetric matrix. With the appearance of the QR algorithm, however, its use declined, as the superiority of the QR algorithm became clear, QR proving to be much faster than Jacobi. With the advent of parallel computers, the interest in Jacobi algorithms revived, as QR is essentially a sequential algorithm, while the Jacobi method readily adapts to parallel mode. In recent years, however, as Edelman [23] mentions, the use of Jacobi is again being questioned, because of its considerable requirements of inter-processor communication by comparison with the extent of computation.

The method is based on the use of orthogonal transformations of the form Q^tAQ to annihilate the off-diagonal elements of A. Each time a transformation is applied, A is replaced by Q^tAQ with the net result that, while A is still as full as previously, it is "more diagonal", since the transformation is effected so that off-diagonal elements are reduced in magnitude. Eventually, after a number of such transformations have been applied, the off-diagonal elements are sufficiently close to zero that they can safely be taken as such. Thus, the method systematically reduces the norm of the off-diagonal elements,

$$\mathit{off}(A) = \sqrt{\sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}^2}. \tag{2.1}$$

The orthogonal transformations are products of plane rotation matrices, each of which we will refer to as *Jacobi's rotations* $J(p, q, \theta)$. A Jacobi rotation is obtained from the identity matrix by changing the entries in positions (p, p), (p, q), (q, p) and (q, q) respectively to $\cos\theta$, $-\sin\theta$, $\sin\theta$ and $\cos\theta$. Suppose that at some point in the algorithm (which will be detailed later) a rotation $J(p, q, \theta)$ is to be applied to A, so that the (p, q) and (q, p) entries are transformed to zero. If $B = J^tAJ$, then by the properties of Givens rotations, B is equal to A except in rows and columns p and q. To determine how to choose p and q such that the annihilation is fastest, note that applying an orthogonal transformation preserves the Frobenius's norm. So, we can write

$$a_{pp}^2 + a_{qq}^2 + 2a_{pq}^2 = b_{pp}^2 + b_{qq}^2 + 2b_{pq}^2 = b_{pp}^2 + b_{qq}^2$$

and

$$\begin{aligned} \mathit{off}(B)^2 &= ||B||_F^2 - \sum_{i=1}^{n} b_{ii}^2 \\ &= ||A||_F^2 - \sum_{i=1}^{n} a_{ii}^2 + a_{pp}^2 + a_{qq}^2 - (b_{pp}^2 + b_{qq}^2) \\ &= \mathit{off}(A)^2 - 2a_{pq}^2 \end{aligned}$$

One can minimize *off*(B) if p and q are chosen such that $|a_{pq}| = \max_{i \neq j} |a_{ij}|$. This leads to the *classical Jacobi algorithm* where at each step it is necessary to do a search of the elements of A (a task which requires $O(n^2)$ comparisons). This is not feasible for large n and a possible solution is to determine *a priori* a certain sequence of p's and q's. In this form, we have, for example, the *cyclic Jacobi algorithm* which annihilates successively the elements in positions $(1,2),(1,3),\ldots,(1,n),(2,3),\ldots,(2,n),(3,4),\ldots,(n-1,n)$. Each of these sequences, which covers all such pairs (i,j) is called a *sweep*.

The *Jacobi algorithm* is attractive for its simplicity and intrinsic parallelism, since it is based on the successive application of rotation matrices which modify only a row and a column at a time. By properly organising the order of application of these matrices, it is possible to obtain sets of rows and columns that are independent of each other. In this way, applying the rotation matrices to these independent sets can be done in parallel. For instance, if one partitions the pairs (p,q) into *disjoint* sets which are assigned to different processors, then the Jacobi's rotations can be applied independently by each processor. If A is partitioned by columns across the processors, then each processor

1. computes the rotation parameters (c, s)
2. applies the rotation over the columns stored locally
3. broadcasts (c, s) to the other processors, to update the corresponding rows

The Jacobi algorithm and its variations for non-symmetric matrices have been the subject of much work on parallel implementations, including one for transputers, which are the processors we use in our implementation of the Divide-and-Conquer method. Some of these implementations and descriptions of the parallel version of the Jacobi algorithm are to be found in [64], [12], [25], [77], [86], [58], [59] and [69]. The last three of these, work with variations of the Jacobi algorithm for non-symmetric or complex matrices, and we will not discuss them here.

Sameh [64] was one of the first to propose a parallel version of the Jacobi algorithm. He observes that we can construct $n/2$ independent rotations simultaneously if we have $n/2$ processors available. As $n(n-1)/2$ off-diagonal elements must be annihilated, if each transformation eliminates $n/2$ elements, then we need $n-1$ of these transformations per sweep, if each element is annihilated just once. In this way we obtain maximum efficiency in the parallel computation. He then proposes two different schemes to perform these $n-1$ transformations. He computes $n/2$ angles in parallel and then computes the next iterate matrix $A_{i+1} = J^t A_i J$ in parallel, using routines developed for the ILLIAC IV, for which the algorithm is intended. The computation of rotations continues until the matrix converges to a diagonal matrix following some established criterion. The author does not report on any result from an actual implementation.

Brent and Luk [12] proposed the use of a square array of $n/2 \times n/2$ processors P_{ij} with at least horizontal and vertical connections. Each processor holds a 2×2 submatrix of A, partitioned contiguously by rows and columns (see [12, Fig. 3, p. 75]). Processors that are located in the diagonal of this array generate the rotation parameters to annihilate the off-diagonal elements of their 2×2 matrix and broadcast these along the corresponding rows and columns of the array (i.e. processor P_{ii} sends information to the processors in row i and column i). The off-diagonal processors P_{ij} update their sub-matrix with the values of the rotation parameters c and s received from processors P_{ii} and P_{jj} (see [12]). After a rotation is applied, rows and columns of adjacent processors are exchanged according to a scheme where column 1 is fixed, the second column of each processor is sent to the processor on the left and the first column of each processor, with exception of processor 1, is sent to the processor on the right. The second column of the first processor becomes the first column of the second processor and the first column of the last processor becomes the second column of this same processor in the next step. For example, if processor P_{11} starts with columns 1 and 2 of A, processor P_{12} has columns 3 and 4 and P_{13}, P_{14} have respectively columns $5, 6$ and $7, 8$ in the beginning, in the next stage the columns of A will be distributed as $(1, 4)$, $(2, 6)$, $(3, 8)$ and $(5, 7)$. In order to maintain symmetry, rows are exchanged using the same rule. These exchanges of data are done to obtain a new set of off-diagonal elements to be eliminated by the diagonal processors in the next step. Every sweep of the cyclic Jacobi algorithm takes time $O(n)$ in this architecture. If we assume that $O(\log n)$ sweeps are needed to achieve convergence, the total time required to diagonalize A is of order $O(n \log n)$. The matrix of eigenvectors can be computed whilst matrix A is being diagonalized, by applying the rotations initially to sub-matrices of the identity matrix partitioned in the same way as A. If only the eigenvalues are required, the number of processors can be decreased to $n(n+2)/8$, because we can take advantage of the symmetry of A. On the other hand, if the eigenvectors are also required, this simplification cannot easily be effected. The authors claim that, when comparing their algorithm with the QR algorithm preceded by a tridiagonalization stage, significant speed-ups can be achieved for moderate or large n. Since their architecture was not actually implemented, substantial testing must be made to ascertain the validity of their claim.

Götze *et al.* [25] propose organizing the processors in a triangular array, where each processor stores a 2×2 submatrix of the upper triangular part of A. Each processor on the array is a CORDIC processor, capable of computing rotation angles using sequences of shift and add operations. The authors modify Jacobi's algorithm to use efficiently the CORDIC processor; a single angle defines the Jacobi rotation in each step of the algorithm. Because these rotations are not calculated exactly due to the nature of CORDIC and the chosen algorithm, the quadratic convergence exhibited by Jacobi's algorithm is lost and the number of necessary sweeps increases. In contrast, the complexity of the algorithm is significantly reduced. However, the authors observe that the Jacobi method also has also a linear rate of convergence during the first sweeps and the difference between the number of sweeps becomes significant only if the tolerance for stopping the algorithm decreases. In this case, the authors propose the use of longer data words to ensure that sufficient precision is achieved.

Tervola and Yeung [77] implemented a parallel Jacobi algorithm on transputers organized in a ring. The original matrix A for which the eigensystem is sought is distributed in groups of columns among the processors. If we partition the original matrix theoretically, in square blocks, each processor annihilates the off-diagonal elements of the block diagonal matrices and then sends half of its assigned columns to neighbouring transputers, until all half-groups of columns are paired with each of the others. The scheme they use for this interchanging of columns and rows is similar to that used by Brent and Luk. The process of annihilation is then repeated after each of these shuffles. In their scheme each pair of columns come together just once. When all half-groups have met each other, one full Jacobi cycle is completed and the Frobenius norm of the off-diagonal elements is computed, to ascertain if more iterations are necessary. These authors computed theoretical efficiencies and speed-ups and found that they approximated the observed measures very well. They report that the maximum efficiency they can achieve with their algorithm when compared with serial Jacobi, is $2/3$, which is obtained when the order of the matrix is much larger than the number of processors. They also remark that when the number of processors and the order of the matrix A have the same order of magnitude, the efficiencies that they can obtain are not good.

Weston and Clint [86] have also implemented the Jacobi algorithm for symmetric matrices. The machine they used for such was the DAP, which is an array processor. They say that the Jacobi algorithm is very suitable for implementation on array processors because a great part of the algorithm is taken up by matrix multiplication and this is an operation that can be successfully executed in these machines. They also interchange rows after computing a set of parallel rotations and calculating the transformed matrix A. The scheme they use for these interchanges of columns and rows is different from that used by Brent and Luk, and Tervola and Yeung. If initially the columns and rows of matrix A are organized for Jacobi rotations as $(1,2)$, $(3,4)$, $(5,6)$ and so on, in the next step they will be organized as $(2,3)$, $(4,5)$, $(6,7)$, etc. They remark that rotations and permutations can be combined in one step and thus no data movement is necessary. They compare the results obtained with their Jacobi implementation with another algorithm called *parallel orthogonal transformation algorithm* (POT) (see [86]). They tested their program with banded matrices and concluded that for these cases, the POT algorithm takes less time than Jacobi because it is well adapted for those matrices while the Jacobi cannot take advantage of their structure. However, for some other sparse matrices, the Jacobi algorithm is superior to the POT. They also state that their implementation computes eigenvalues and eigenvectors, with at least ten correct digits (using a precision of 16 decimal places).

2.2 Parallel bisection and Multisection

Algorithms and implementations of a parallel version of bisection and its extension – multisection – are described and/or compared with other algorithms in [50], [40], [26] and [7].

Sequential bisection is usually employed when a few eigenvalues and their corresponding eigenvectors of a symmetric tridiagonal matrix are required. After the eigenvalues are found, the eigenvectors are computed via inverse iteration, where in most cases just one iteration is enough to compute an eigenvector [50]. If there are clusters of eigenvalues a modified Gram-Schimidt orthogonalization process is applied to those eigenvectors corresponding to neighbouring eigenvalues. Bisection uses Sturm sequences (see, for example [88], [36] or [42]) to determine the number of eigenvalues located in a certain interval. It relies on the fact that the number of eigenvalues smaller than λ is the same as the number of disagreements in the signs of successive terms of the Sturm sequence evaluated at λ, so that the number of eigenvalues in the interval $[a, b]$ can be computed from the Sturm sequences calculated at a and b.

Both bisection and multisection have two distinct phases: the first tries to isolate the eigenvalues, so that each sub-interval is known to contain only one eigenvalue, and the second consists in extracting these eigenvalues or computing them to a certain accuracy. Both methods compute very accurate eigenvalues, but need to orthonormalize eigenvectors which correspond to close eigenvalues. The usual linear recurrences which define the Sturm sequences can suffer from over- or under-flow, so that in many cases a nonlinear recurrence is used. The terms are defined by $q_i(\lambda) = p_i(\lambda)/p_{i-1}(\lambda)$, where p_i and p_{i-1} are the usual quantities appearing in the Sturm sequence recurrence. With this modification we find the number of negative terms is the sequence q_i.

The *parallel bisection algorithm* as described by Ipsen and Jessup [40] starts with the computation of an interval which is known to contain all the eigenvalues. This can be found, for example, by considering the union of Gerschgorin disks. The Ipsen and Jessup implementation is intended for a hypercube, a distributed-memory machine, so that some tasks are designated to be executed by all processors. The determination of this initial interval is performed by all processors. Each processor computes a number k of eigenvalues, where $n = kp$ is the order of the matrix and p is the number of processors. After the determination of the initial interval, each processor determines the sub-interval which contains the eigenvalues it was assigned to compute and then computes by bisection the k consecutive eigenvalues for which it is responsible. All the eigenvalues are then exchanged. Each processor then computes k eigenvectors, but this time the eigenvectors do not correspond to consecutive eigenvalues, but processor i performs computations related to eigenvalues $i, i+p, i+2p, \ldots$. These eigenvectors are found by inverse iteration, but the calculations are organized in this way, in order to keep a better load balance if many orthonormalizations are necessary. The final stage then is the orthonormalization of eigenvectors corresponding to close eigenvalues.

The *multisection algorithm* is an extension of bisection. The first stage is also the determination of an interval which contains all the eigenvalues. Again this is calculated by all processors. After this, each processor is allocated a sub-interval of equal length. Each processor uses Sturm sequences to compute how many eigenvalues there are in its sub-interval. All processors exchange the numbers of eigenvalues in their various sub-intervals. The sub-intervals that contain more than one eigenvalue are "put in a queue" and are again successively multisectioned until either each interval contains just one eigenvalue (which means that the eigenvalue has been *isolated*), or clusters of eigenvalues are identified. Clusters are identified by means of some criterion to determine if the interval containing several eigenvalues is small (for example, a small multiple of machine epsilon). The isolated eigenvalues are then computed by bisection each processor being responsible for a certain number of them. As in the bisection algorithm, the eigenvalues are then exchanged, the eigenvectors are computed via inverse iteration, and, if necessary, they are orthonormalized.

Ipsen and Jessup found that their bisection implementation works best (i.e., gives better speed-ups and efficiencies) when compared with QR/QL for matrices having orders that are multiples of the number of processors, because for other orders, the workload is not as well distributed. They report that maximal speed-up is not achieved, due to non-arithmetic overheads or redundant computations. They were able to obtain efficiencies ranging from 77 to 89 percent for matrices with orders greater than 100 (when comparing with the QR/QL algorithm). The results they obtained with multisection were not as good and the efficiencies ranged from 26 to 60 percent for matrices with orders between 64 and 1024. The authors attribute this poor efficiency to increased overheads, reporting that communication and idle time take over 40 percent of the total time at lower orders and 5 percent of total time for a matrix with order 1024. For bisection the corresponding figures for overheads are 20 and 2 percent respectively. The authors compare the *Parallel Bisection, Multisection* and *Cuppen's Divide-and-Conquer* algorithms. They conclude that bisection is the fastest of the three algorithms for computing all the eigenvalues and eigenvectors of a symmetric tridiagonal matrix, but Cuppen's Divide-and-Conquer is the one which gives the more accurate solutions, with smaller residuals and deviations from orthogonality. For matrices with order $n = 512$, for example, the residuals of Cuppen's method are $O(10^{-15})$ while the residuals of the two other methods are $O(10^{-13})$.

The basic stages of the implementations of Lo et al. [50], do not differ much from the description above, but as they use shared-memory machines, tasks are assigned to each processor once they have completed previous ones. Another difference is that they have two different implementations for multisection, one that computes the eigenvalues by bisection (routine TREPS1) and another which employs instead a routine called Zeroin, based on the secant method, to compute the eigenvalues (routine TREPS2). They obtained almost linear speed-ups when executing their program in $1, 2, 4$ and 8 processors. If just some of the eigenvalues and corresponding eigenvectors are required, the speed-ups are related to the number of eigenpairs required. In 8 processors, for a matrix with order 300, the speed-up ranges from 5, when 10 eigenpairs are required, to 7.2 when 200 eigenpairs are sought, achieving a maximum of 7.5 when all eigenvalues and eigenvectors are computed. They also compare their algorithms with EISPACK's TQL1 and TQL2 (implementations of the QL algorithm to find all eigenvalues or all eigenvalues and eigenvectors, respectively) and with SESUPD (an implementation of Cuppen's Divide-and-Conquer). The conclusions are similar to those reported by Ipsen and Jessup. SESUPD gives more accurate solutions, with differences of up to 3 decimal orders when compared with Lo et al.'s multisection. They obtained speed-ups of 32.9 and 131.5 over the QL algorithm, when executing their algorithms in 8 processors for a matrix of order 500. They also report that TREPS2 can be 4.8 times faster than SESUPD for the matrix $(-1, 2, -1)$, but is just 1.6 times faster for a random tridiagonal matrix. However, the orthogonality of TREPS2 is poorer by 3 decimal places when compared with SESUPD. TREPS2 is usually faster than TREPS1, but in some cases the method Zeroin for computing the eigenvalues used in TREPS2 can suffer from over- or under-flow, and in this case the eigenvalues are re-computed using bisection, which then makes TREPS2 slower than TREPS1. Finally they conclude that algorithms TREPS1 and TREPS2 are suited for parallel computations and in their opinion should be the method of choice for computing all the eigenvalues or some of the eigenvalues and corresponding eigenvectors, being also competitive when the whole eigensystem of a symmetric matrix is required.

Gallivan et al. [26] observe that the multisection algorithm proposed by Lo et al. may exhibit problems with accuracy if there are very close eigenvalues. For the Wilkinson matrix W_{127}^{+} (see [87]) the time taken by the multisection algorithm can be twice that taken by Cuppen's Divide-and-Conquer for the same degree of accuracy.

Berry and Sameh [7] compare Jacobi algorithms with multisection and TQL2, in the two last cases coupled with the reduction from a full symmetric matrix to tridiagonal form. They found that the version of multisection which uses matrix-matrix operations (BLAS3) in the reduction to tridiagonal is faster than the other methods and can be 4 times faster on 8 processors than QL combined with the usual reduction to tridiagonal form.

The parallel bisection/multisection algorithm has also been implemented in transputers, as Kalamboukis [43] reports. For this, transputers were organized in a ring with links communicating in both directions. The author was able to obtain almost linear speed-ups and reports on the times obtained for computing all the eigenvalues with $1, 4$ and 9 transputers, mentioning that these times compared favourably with those obtained by Lo et al..

Basermann and Weidner [5] implemented parallel bisection/multisection in a SUPRENUM multiprocessor system with 16 processors and in a CRAY Y-MP8/832 with 8 processors. The difference from other implementations is that after isolating the eigenvalues, they are located by bisection, accelerated by the Pegasus method. This method achieves superlinear convergence only if the starting points are sufficiently near the desired root: thus, to obtain a good approximation, bisection is employed first (for details see [5]). The authors were able to obtain speed-ups of 6.8 to 7.3 when using the Cray and comparing the sequential and parallel implementations of their algorithm. Their sequential version was up to 2.2 times faster than TQL1 when used to find all the eigenvalues and they claim they were able to obtain 2 to 3 additional correct decimal digits beyond those found by TQL1.

2.3 Homotopy methods

Li and Rhee [47] and Li et al. [48] present an homotopy method to compute the eigensystem of a matrix. The method is inherently parallel. The computation of any eigenvalue is independent of the calculation of the others. The idea of the method is to "depart" at time $t = 0$ from an eigenpair of a matrix for which the eigensystem is easily known and "arrive" at time $t = 1$ at an eigenpair of the matrix for which the eigensystem is sought. Supposing that we are working with a real symmetric tridiagonal matrix A, we can define an homotopy equation H by

$$H(x, \lambda, t) = \begin{bmatrix} (1-t)(\lambda x - Dx) + t(\lambda x - Ax) \\ (x^t x - 1)/2 \end{bmatrix} = \begin{bmatrix} \lambda x - A(t)x \\ (x^t x - 1)/2 \end{bmatrix} = 0 \qquad (2.2)$$

where $A(t) = D + t(A - D)$.

It has been shown, that if we choose D to be a diagonal matrix with distinct diagonal elements, then $H^{-1}(0)$ consists of $2n$ disjoint curves, with each one going from an eigenpair of D at $t = 0$ to an eigenpair of A at $t = 1$. The k^{th} eigenpair of A corresponds to the k^{th} eigenpair of D, so that if we want to find an eigenpair of A, we simply need to follow the path that departs from the corresponding eigenpair of D. As A is symmetric, only n of these paths are necessary to compute all the eigenpairs of A. To follow the k^{th} eigenpath to find the k^{th} eigenpair of A, which is $\lambda(1), x(1)$, a predictor-corrector method is used. That is, after choosing a new step $t_i = t_{i-1} + h$, a new prediction $\lambda(t_i)$ for the eigenvalue is computed. This new prediction is used as a shift in the inverse iteration method, to compute a new approximate eigenvector $x(t_i)$, i.e., we solve $[A(t_i) - \lambda(t_i)I]y(t_i) = x(t_{i-1})$ and then set $x(t_i) = y(t_i)/||y(t_i)||$. The value of the new predicted eigenvalue $\lambda(t_i)$ is computed by some interpolation method from the values of $\lambda(t_{i-2}), \lambda'(t_{i-2}), \lambda(t_{i-1})$ and $\lambda'(t_{i-1})$. The two derivatives can be computed by differentiating the equation for $H(x, \lambda, t)$. Once the new predicted eigenvector is computed by inverse iteration, the eigenvalue is "corrected" by computing the Rayleigh quotient $x(t_i)^t A(t_i) x(t_i)$ and making this value the new approximation for $\lambda(t_i)$. The eigenvector is corrected by applying inverse iteration to matrix $A(t_i)$ with shift $\lambda(t_i)$. This process is then repeated until an adequate approximation for the eigenpair $(\lambda(t_i), x(t_i))$ is obtained.

Details of how new approximations for the eigenvalue are found and other aspects concerning the homotopy algorithm can be found in [47] and [48]. The latter discusses issues related to the parallel implementation algorithm and solves the problem, pointed out in the first paper, of computing accurate eigenvectors when there are close eigenvalues. In the parallel implementation, the matrix D is not necessarily taken to be a diagonal matrix, but is a block diagonal matrix where each block is a diagonal block present in the initial matrix A. This matrix D should be chosen as closer as possible to matrix A, but the way of making this optimal choice for D is not completely solved yet. After matrix D has been selected, processor i computes the eigenvalues of matrix A_i, which is the i^{th} block of D. The eigenvalues computed by each processor are communicated to the others, and each processor sorts these eigenvalues to obtain the ordered set of eigenvalues of D. The number of eigenpairs to be determined is partitioned equally among the processors, which then trace the corresponding eigenpaths starting from the eigenpairs of D computed previously. Adjustments can be made if instead, the eigenvalues located in a certain interval are desired. If the number of eigenpairs sought is smaller than the number of processors available or the sub-matrices are too large to fit in the local memories, alternative parallelisations have been devised for following an eigenpath.

Li et al. [48] compare their parallel implementation of the homotopy method in the hypercube with sequential versions of QL and bisection, and compare indirectly with Dongarra and Sorensen's [20] implementation of Divide-and-Conquer and with Lo et al.'s multisection algorithms. For the $(1, 2, 1)$ matrix they were able to obtain speed-ups of 4.1 in 8 processors in relation to the sequential bisection and speed-ups of 16.8, when comparing with sequential QL for a matrix of order 64 and of 34.9 for a matrix with order 128. Again, comparing with sequential bisection and using 16 processors and working with a matrix with order 512 they obtained a speed-up of just 5.9. When tests were done with the Wilkinson matrices W^{+}_{2k+1} [87], and they compared their parallel implementation with the sequential bisection, the performance of their algorithm was better than with the $(1, 2, 1)$ matrix. Bisection takes 37.2 more time than homotopy using 16 processors for a matrix of order 511. When comparing their parallel implementation of homotopy with the sequential one, they obtained up to 99% of efficiency for both test matrices described above. The comparison with parallel implementations of other algorithms is done in terms of the speed-ups achieved for each algorithm in relation to the QL algorithm. They conclude that their algorithm is competitive with both multisection and Divide-and-Conquer, and is an algorithm well suited for parallel computations.

2.4 The QR/QL algorithm

As mentioned before, the QR/QL algorithm has been the preferred choice for many years for computing the full eigensystem of a tridiagonal matrix in a sequential mode. However, as it is inherently sequential, each stage of the computation heavily dependent on previous ones, the parallelisation of the symmetric QR/QL algorithm for finding only the eigenvalues of a matrix does not seem to be an easy task. However, if the eigenvectors are also required, the scope for parallelisation increases, as the computation of the eigenvectors is the part of the algorithm which consumes most of the time and can be computed in parallel. There are some implementations for the non-symmetric case, such as those reported by Ward [80], for vector computers, and by Geist and Davis [28], Stewart [72] and Davis et al. [18], for distributed-memory machines. For the symmetric case we have only encountered the work of Sameh and Kuck [65] (which computes only the eigenvalues), and, more recently, that of Schreiber et al. [66].

Sameh and Kuck transform the nonlinear recurrences that appear in the usual formulation of the QR algorithm to linear recurrence relations, which are then solved in parallel. This includes solving two triangular systems of order n in parallel. They do not show any results of an actual parallel implementation, but only the results they obtained with a simulated implementation on an IBM 360/75, which is a sequential machine. They present theoretical speed-ups, in which they compare their algorithm with a sequential QR implementation for which it was shown that the time required for one iteration is $11n$. In their algorithm they assume that $2n$ processors are used and obtain a complexity of $6 + \log_2 n$, which then gives approximate speed-ups of $11n/4 \log_2 n$ and efficiencies of $11/(8 \log_2 n)$. Easy calculations show that even for a small value of n, as $n = 64$, we obtain an efficiency smaller than 25%. For greater values of n, the efficiencies simply get worse, which leads us to conclude that their algorithm design, does not lead to an efficient scheme for parallelisation. In addition, the algorithm can suffer from under- or overflow, but this can be solved by some scaling. They report that in their simulation they obtained eigenvalues which agreed with the eigenvalues computed by TQL1 (the EISPACK QL routine to compute the eigenvalues alone) in at least 9 decimal places (using double precision with a machine-epsilon of 2.2×10^{-16}).

Schreiber et al. [66] describe a parallel version of the QR algorithm to compute all the eigenvalues and eigenvectors of an hermitian tridiagonal matrix. They have not implemented their algorithm yet, but believe they will obtain good results when this is done. They observe that the task which takes most of the time ($O(n^2)$) is the computation of the elements of matrices $Q^{(p)}$, where $H^{(p)} = Q^{(p)}R^{(p)}$ (a QR factorization) and parallelising this step is their chief concern. Recall that the QR/QL algorithm constructs matrices $R^{(i)}$ and $Q^{(i)}$ such that $H^{(i)} = Q^{(i)}R^{(i)}$, where $R^{(i)}$ is an upper triangular matrix and $Q^{(i)}$ is an unitary matrix, and then compute $H^{(i+1)} = R^{(i)}Q^{(i)}$. After sufficiently many iterations we arrive at the matrix $H^{(k)}$ which is approximately a diagonal matrix, and is unitarily equivalent to the original matrix $H^{(0)}$.

Schreiber et al.'s method supposes we start with an $n \times n$ Hermitian tridiagonal matrix $H = [h_{ij}]$, and set $R^{(0)} = H$ and $Q^{(0)} = I$. At step p we want to annihilate element $h_{p+1,p}$. This annihilation is done by elementary unitary matrices $U_{p,p+1}$ whose elements in positions $(p,p), (p,p+1), (p+1,p)$ and $(p+1,p+1)$ are given by $c^{(p-1)}, s^{(p-1)}, \bar{s}^{(p-1)}, -\bar{c}^{(p-1)}$, respectively, and depend on the values of entries $r_{p+1,p}^{(p-1)}$ and $r_{p,p}^{(p-1)}$ of $R^{(p-1)}$. The other elements of $R^{(p-1)}$ are either 0 (if they are not in the diagonal) or 1 (if they are in the diagonal). Matrices $R^{(p)}$ and $Q^{(p)}$ are then computed by $R^{(p)} = U^*_{p,p+1}R^{(p-1)}$ and $Q^{(p)} = Q^{(p-1)}U_{p,p+1}$. Note that only the elements in rows p and $p+1$ of $R^{(p)}$ and in columns p and $p+1$ of $Q^{(p)}$ differ from those in $R^{(p-1)}$ and $Q^{(p-1)}$, and because of the shape of each matrix $R^{(j)}$, we need to compute at most 5 elements of $R^{(p)}$, for each p. Similarly, not all entries of columns p and $p+1$ of $Q^{(p)}$ need to be computed: if $p < n$, and we are in the first set of iterations of the QR algorithm, $(p+1)$ entries should be calculated in each of these columns. For the computation of $q_{i,p}$ and $q_{i,p+1}$, the values of $c^{(p-1)}$ and $s^{(p-1)}$ are required and for these, the values of $r^{(p-1)}_{p,p}$ and $r^{(p-1)}_{p+1,p}$ are needed. However, the authors remark that for an efficient parallelisation they do not require matrix $R^{(p-1)}$ to be computed in all processors, nor do they need the computed entries to be communicated to all processors. The way they proceed is to use what they call *cyclic reductions*. The initial matrix $R^{(0)} = H$ is broadcasted to all processors and then the required values of $r^{(p-1)}_{p,p}$ and $r^{(p-1)}_{p+1,p}$ can be computed without computing all the other elements. Once $c^{(p-1)}$ and $s^{(p-1)}$ are available, the values of $q^{(p-1)}_{ip}$ and $q^{(p-1)}_{i,p+1}$ can be computed for all the necessary i's. We must note that if we consider all the steps p ($p = 1, 2, \ldots, n-1$), the number of elements of Q that should be computed in each row, decreases as the number of the corresponding row increases. For this reason, the authors suggest using a "team-mapping" strategy to determine which rows each processor should be responsible for. In this way, the first processor should compute the elements in the first row, the second processor the elements in the second row, and so on, until the final processor P is responsible for the P^{th} row. The $(P+1)^{st}$ row is also assigned to processor P, because this is the row which has the greater workload from the remaining ones. The process of assignment continues in the same way, in a "zigzag" manner until all rows of Q are assigned to some processor. The authors claim that in this way the workload is equally distributed and they are thus able to obtain 100% efficiency when the order n of matrix H goes to infinity. This theoretical claim has still to be verified in practical implementations.

2.5 Summary

We have presented some methods for computing the eigenvalues and eigenvectors of a symmetric matrix which are suitable for parallel implementations. Some other methods for computing the eigensystem of a symmetric matrix can be found in [51], [76] and [61].

The Jacobi algorithm which was much employed for finding the eigensystem of a matrix before the QR algorithm assumed its superiority, gained a renewal of interest with the appearance of parallel computers, because it is highly suitable in that environment, while to date no efficient parallel implementation appears to exist for the QR algorithm. However, parallel Jacobi methods need substantial amounts of communication, which reduces their performance and can make them compare unfavourably with other parallel methods. It is difficult to compare execution times between different algorithms run on different machines, but the study done by Berry and Sameh [7] shows that Jacobi implementations are not, in general, the best choice for computing the eigensystem of a symmetric matrix and can be much slower (even taking up to 17 times more time) than multisection when the eigenvalues of these matrices are uniformly distributed. However, it can compete with multisection when the eigenvalues of the matrix in question are poorly separated (or in clusters), being up to 3.5 times faster than multisection for small matrices (order up to 30) and taking approximately the same time for matrices with orders between 40 and 80. The Jacobi implementations also seem to deliver eigenvectors with good mutual orthogonality, by comparison with multisection.

Berry and Sameh's experiments included the transformation of the full symmetric matrix to tridiagonal form using BLAS2 routines (vector-matrix operations) or BLAS3 routines (matrix-matrix operations). In their tests, the gains of multisection over TQL2 with TRED2 (the EISPACK routine to transform a full matrix to tridiagonal form) were smaller than those obtained by Lo et al.. While the latter were able to obtain speed-ups of up to 131.5, when both algorithms operate on tridiagonal matrices, the speed-ups obtained by Berry and Sameh were only around 4, but the explanation for this discrepancy may be in part due to the routine to tridiagonalise a matrix not being efficiently parallelised. Also in Berry and Sameh's tests, multisection is 2 times faster than QR/QL, when using only one processor, which can make the former a good option even in sequential mode. However, for computing the eigenvalues alone, Lo et al.'s work found smaller gains compared to when all the eigenvalues and corresponding eigenvectors are determined. In the former case, the fastest version of multisection is only 5 times faster than TQL1, in 8 processors.

The multisection algorithm can also be faster (almost 5 times) than Divide-and-Conquer when tested on the matrix $(-1, 2, -1)$, but this gain is not as high when the test matrix is a random one (1.5 times faster). In addition, the accuracy obtained by Divide-and-Conquer is usually higher than multisection, both in the residuals and in the measure for orthogonality. Ipsen and Jessup arrive at the same kind of conclusions: multisection is faster, but Divide-and-Conquer is more accurate. In their experiments, which were done in a distributed-memory machine, in contrast to the other authors we have mentioned, who used shared-memory computers, they also compared these two algorithms with parallel bisection and concluded that the latter is the fastest of the three, with accuracies comparable to those of multisection. In relation to TQL2 they compared efficiencies obtained with their distributed-memory implementations with the efficiencies obtained by other researchers in shared-memory applications. They concluded that bisection and multisection can be efficiently parallelised in a machine with local-memory, because both algorithms do not need much communication. On the other hand, Cuppen's Divide-and-Conquer seems to behave better on shared-memory architectures.

The homotopy algorithm can be compared, indirectly, with bisection/multisection and Dongarra and Sorensen's [20] implementation of Cuppen's Divide-and-Conquer. Li et al. observe that the speed-up obtained by their parallel homotopy algorithm in relation to QL, is greater (34.9 in 8 processors for $n = 128$) than those obtained by Dongarra and Sorensen (9.4 for $n = 100$) and is comparable to speed-ups obtained by Lo et al. with their bisection/multisection algorithm.

In conclusion, there are some recent parallel algorithms which seem to have efficient implementations. These algorithms are better options than the QR/QL algorithm when employing a parallel machine, as far as execution times are concerned; the latter does not yet have any efficient parallel version. The gains obtained over QR/QL can be very large, but it must be observed that in general, the results obtained by QR/QL are more accurate than those obtained by other methods. Comparing bisection, multisection, Jacobi and Divide-and-Conquer, bisection seems to be the fastest, but Divide-and-Conquer delivers more accurate results. In addition, there are cases in which Jacobi is competitive with multisection. Homotopy also seems to be a good choice for computing the whole eigensystem of a matrix. It is not possible to give a definitive answer in respect to which method is the best. For this, more experiments should be done in which all algorithms are executed on the same machine. In addition, we must take into account exactly what is the goal when selecting this "best" algorithm: whether our objective is to reduce the time taken to compute what we want while retaining acceptable accuracy, or whether we want the best possible accuracy, even though execution times may not be so good.

2.6 Eigenvalue routines in Numerical Libraries

One of the first attempts to provide high-quality numerical software for the symmetric eigenvalue problem was made by Wilkinson and Reinsch in [88]. The routines in this book are in ALGOL60. This was followed by the EISPACK library containing FORTRAN alternatives to those routines given in [88]. This project was extended to solve a wider class of eigenproblems [27].

2.6.1 LAPACK and SCALAPACK

The LAPACK library [2] offers a number of routines for the symmetric tridiagonal eigenproblem. These routines employ basically the QL, QR (and variations) and the *bisection* algorithms. For non tridiagonal matrices, a number of routines are available to reduce to tridiagonal form, both in the real and complex cases.

This project was an attempt to provide the functionality of LINPACK and EISPACK in a way that it is portable for a large range of modern computers. The routines on LAPACK not only provide more efficient implementations of the EISPACK routines, but also improve on the accuracy of some of the standard algorithms.

The `xSTEQR` routine uses a combination of the QL and QR algorithms which makes it more robust than the `IMTQL1` and `IMTQL2` routines previously provided with EISPACK. A square-root free version of the above algorithms is provided by the `xSTERF` routine, but it does not compute eigenvectors. For positive definite matrices a combination of Cholesky factorization and the bidiagonal QR algorithm is available on `xPTEQR`; its accuracy is in some cases better than that provided by `xSTEQR` or `xSTERF`.

The `xSTEBZ` routine uses the bisection algorithm to compute some or all of the eigenvalues. The selection of the desired eigenvalues may be given in terms of their ordering (e.g., from the i-th to the j-th eigenvalue) or in terms of a containing closed real interval. The corresponding eigenvectors can then be computed with the `xSTEIN` routine which uses inverse iteration (this routine can also be used in conjunction with `xSTERF`).

We also mention the SCALAPACK project [16], which aims to produce a version of LAPACK with algorithms suitable for scalable (distributed-memory) multiprocessor computers, while the latter was aimed at shared-memory machines. The current version (1.0BETA) offers parallel reduction routines (to upper Hessenberg, bidiagonal and tridiagonal forms) which can be used as building blocks for a parallel eigensolver (not yet available on SCALAPACK).

2.6.2 NAG

The NAG library offers a number of subroutines for eigenproblem solution, of which we mention `F02AMF` which computes all eigenvalues and eigenvectors of a real symmetric tridiagonal matrix, and `F02BEF` which computes selected eigenvalues and corresponding eigenvectors of such a matrix.

Chapter 3

The Divide-and-Conquer method

3.1 Introduction

From our discussion in Chapters 1 and 2, we need to look for new algorithms which work well on parallel systems. When we began our work, the Divide-and-Conquer method for finding the eigensystem of a tridiagonal symmetric matrix was still in its early stages of development. There was some published work (as Bunch et al. [13], Cuppen [17], Dongarra and Sorensen [20] and Ipsen and Jessup [40]). However, the theoretical development was still not complete and there was no implementation for transputers. Since beginning this investigation in 1990, several substantial papers have been published on various aspects of divide-and-conquer methods. We report on these papers in Chapter 7. Here we describe the basic theory underlying the Divide-and-Conquer method for determining matrix eigenvalues and eigenvectors. We include what we believe to be a new development of that part of the theory leading up to the so called *secular equation*, whose roots are the eigenvalues. We also show how the method behaves when the initial partition has more than two pieces, as it is this form which is needed for a parallel implementation. We defer to the next chapter our consideration of matters related to finding the roots of the secular equation.

Divide-and-conquer methods in general consist of an initial partition of the original problem into sub-problems (the "divide" part) and, after some appropriate computations are performed on the individual sub-problems, a phase in which these partial results are joined together in a suitable way to produce the solution to the initial problem (the "conquer" part). These sub-problems are of the same type as the original problem. Normally, divide-and-conquer methods involve divisions into two parts, but these divisions can be repeated in a recursive way, until we arrive at small scale problems that are easy to solve. We describe the application of this idea to our particular case of finding the eigensystem of a symmetric tridiagonal matrix, considering at first, a division into two parts, to make the presentation of the method clearer.

3.2 The Divide-and-Conquer method

Consider a tridiagonal matrix T,

$$T = \begin{bmatrix} \tau_1 & \sigma_1 & & & \\ \sigma_1 & \tau_2 & \sigma_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \sigma_{n-2} & \tau_{n-1} & \sigma_{n-1} \\ & & & \sigma_{n-1} & \tau_n \end{bmatrix} \tag{3.1}$$

We can write this as

$$T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + \alpha b b^t \tag{3.2}$$

where

$$b = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{array}{l} \\ \\ \\ n_1^{th} entry \\ (n_1+1)^{st} entry \\ \\ \\ \\ \end{array} \tag{3.3}$$

and

$$\alpha = T(n_1, n_1 + 1) = \sigma_{n_1}$$

is the $(n_1, n_1+1)^{th}$ entry of T. Here n_1 could be any natural number in the range $1, 2, \dots, n$, but in implementations we normally chose $n_1 = \lfloor n/2 \rfloor$.

We then have

$$T_1 = \begin{bmatrix} \tau_1 & \sigma_1 & & & \\ \sigma_1 & \tau_2 & \sigma_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \sigma_{n_1-2} & \tau_{n_1-1} & \sigma_{n_1-1} \\ & & & \sigma_{n_1-1} & \tau_{n_1} - \sigma_{n_1} \end{bmatrix}$$

$$T_2 = \begin{bmatrix} \tau_{n_1+1} - \sigma_{n_1} & \sigma_{n_1+1} & & & \\ \sigma_{n_1+1} & \tau_{n_1+2} & \sigma_{n_1+2} & & \\ & \ddots & \ddots & \ddots & \\ & & \sigma_{n-2} & \tau_{n-1} & \sigma_{n-1} \\ & & & \sigma_{n-1} & \tau_n \end{bmatrix}$$

What we try to do is find the eigensystems of the smaller matrices T_1 and T_2, and use these as intermediaries to determine the eigensystem of T. For the moment suppose we know the eigensystems of T_1 and T_2, so that we may assume the diagonal forms

$$T_1 = Q_1 D_1 Q_1^t$$

and

$$T_2 = Q_2 D_2 Q_2^t$$

where D_1 and D_2 are diagonal matrices and Q_1 and Q_2 are orthogonal as usual. We then have

$$T = \begin{bmatrix} Q_1 D_1 Q_1^t & \\ & Q_2 D_2 Q_2^t \end{bmatrix} + \alpha b b^t \tag{3.4}$$

We can write this as

$$T = \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} \left(\begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix} + \rho z z^t \right) \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix}^t \tag{3.5}$$

where z is the column vector which satisfies

$$\sqrt{\rho} \begin{bmatrix} Q_1 & \\ & Q_2 \end{bmatrix} z = \sqrt{\alpha} b$$

(assuming $\alpha > 0$) and ρ is some positive scalar; that is

$$\sqrt{\rho} \begin{bmatrix} Q_1 z(1:n_1) \\ Q_2 z(n_1+1:n) \end{bmatrix} = \sqrt{\alpha} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

This is equivalent to

$$\begin{aligned} \sqrt{\rho} Q_1 z(1:n_1) &= \sqrt{\alpha} e_{n_1} \\ \sqrt{\rho} Q_2 z(n_1+1:n) &= \sqrt{\alpha} e_1 \end{aligned}$$

It is convenient to scale z and determine ρ, by setting

$$z = \frac{1}{\sqrt{2}} \begin{bmatrix} (\mathit{last\ row\ of}\ Q_1)^t \\ (\mathit{first\ row\ of}\ Q_2)^t \end{bmatrix} \tag{3.6}$$

and $\rho = 2\alpha$, so that z is now a unit vector.

Calling $D = \begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix}$ we have $T = Q(D + \rho zz^t)Q^t$ and now, to determine eigenvalues and eigenvectors for T, it suffices to find the eigensystem of the "central" matrix

$$D + \rho zz^t\,. \tag{3.7}$$

We look at a way of doing this in the next section.

We note that Dongarra and Sorensen [20] suggest we use a scaling factor in the tearing process (3.2), in order to avoid cancellation. In this way they multiply the α that appears in (3.2) by a quantity θ chosen according to entries $T(n_1, n_1)$ and $T(n_1 + 1, n_1 + 1)$ and modify b correspondingly.

3.3 The Eigensystem of the Matrix $D + \rho zz^t$

In this section we will see that the eigenvalues of the matrix $D + \rho zz^t$, when $\rho \neq 0$, are the zeros of a certain rational function and that there is a formula for expressing each eigenvector in terms of its associated eigenvalue. For the moment we assume that $d_1 < d_2 < \cdots < d_n$ and that $z_j \neq 0$ for all j. In Section 3.4, which addresses the issue of deflation, we show how we may always reduce to this situation. Recall that $\|z\| = 1$. It is also convenient to suppose $\rho > 0$. If the latter is not the case we replace ρ by $-\rho$ and D by $-D$.

The usual way to arrive at the rational function, cited above, is first to find the formula for the eigenvectors and deduce from it the secular equation. We think that this approach "inverts" the development of the theory (finding the eigenvectors and then the eigenvalues); the other more natural way is to isolate the eigenvalues and then calculate the eigenvectors.

Our proof, which we believe to be new, is not, perhaps, as direct as the usual one, requiring some preliminary preparation. It uses the following theorem which is a particular case of a more general theorem stating that the $det(I_m + AB) = det(I_n + BA)$, where det indicates the determinant of the expression within the brackets, A is a matrix of order

$m \times n$, B is $n \times m$ and I_m, I_n are identity matrices of order m and n, respectively.

Theorem 3.1 *$det(I + ab^t) = 1 + b^t a$, where a and b are column vectors.*

Proof. We may suppose that $\|a\| = 1$ by replacing a with $a/\|a\|$ and b with $\|a\|b$. For any orthogonal $n \times n$ matrix P, because $(det\, P)(det\, P^t) = 1$, we have

$$\begin{aligned} det(I + ab^t) &= (det\, P)det(I + ab^t)(det\, P^t) \\ &= det(P(I + ab^t)P^t) \\ &= det(I + Pab^tP^t) \\ &= det(I + (Pa)(Pb)^t) \end{aligned}$$

Now, we may choose P to be the Householder matrix $I - (2uu^t)/\|u\|$, where $u = a - \|a\|e_1 = a - e_1$. Then $Pa = (1, 0, \ldots, 0)^t = e_1$ (see [30], [74] or [73]).

If $Pb = c$, then

$$(Pa)(Pb)^t = e_1c^t = \begin{bmatrix} c_1 & c_2 & \cdots & c_n \\ 0 & & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & & \cdots & 0 \end{bmatrix}$$

Now

$$\begin{aligned} det(I + e_1c^t) &= \begin{vmatrix} 1 + c_1 & c_2 & & \cdots & c_n \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & & \cdots & 0 & 1 \end{vmatrix} \\ &= 1 + c_1 = 1 + c^te_1 \\ &= 1 + (Pb)^t(Pa) \\ &= 1 + b^tP^tPa \\ &= 1 + b^ta \end{aligned}$$

So that

$$det(I + ab^t) = det(I + e_1c^t) = 1 + b^ta.$$

We apply the above result to the problem of determining the eigenvalues of $\tilde{D} = D + \rho zz^t$. The eigenvalues of $\tilde{D}$ are the roots of the polynomial equation $det(\tilde{D} - \lambda I) = 0$. When λ is distinct from d_i, $1 \leq i \leq n$, then

$$\begin{aligned} \tilde{D} - \lambda I &= D + \rho zz^t - \lambda I = (D - \lambda I) + \rho zz^t \\ &= (D - \lambda I)(I + \rho(D - \lambda I)^{-1}zz^t) \end{aligned}$$

Thus

$$det(\tilde{D} - \lambda I) = det(D - \lambda I)det(I + \rho(D - \lambda I)^{-1}zz^t)$$

But $det(I + ab^t) = 1 + b^ta$ (see Theorem 3.1), hence setting $a = (D - \lambda I)^{-1}z$ and $b = \rho z$, we have

$$det(\tilde{D} - \lambda I) = det(D - \lambda I)(1 + \rho z^t(D - \lambda I)^{-1}z). \tag{3.8}$$

We look carefully at the second factor above

$$1 + \rho z^t(D - \lambda I)^{-1}z. \tag{3.9}$$

In terms of the components of z and D we can write (3.9) as

$$1 + \rho[\, z_1, \;\ldots,\; z_i, \;\ldots,\; z_n \,] \begin{bmatrix} \frac{1}{d_1-\lambda} & & & & \\ & \ddots & & & \\ & & \frac{1}{d_i-\lambda} & & \\ & & & \ddots & \\ & & & & \frac{1}{d_n-\lambda} \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_i \\ \vdots \\ z_n \end{bmatrix}$$

$$= 1 + \rho \begin{bmatrix} \frac{z_1}{d_1-\lambda} & \cdots & \frac{z_i}{d_i-\lambda} & \cdots & \frac{z_n}{d_n-\lambda} \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_i \\ \vdots \\ z_n \end{bmatrix}$$

or

$$1+\rho\sum_{j=1}^{n}\frac{z_j^2}{d_j-\lambda} \tag{3.10}$$

We observe below that (3.10) has exactly n zeros, so that any zero of (3.10) is necessarily one of the n eigenvalues of $D+\rho zz^t$.

By considering the graph of the function $f(\lambda)=1+\rho\sum_{j=1}^{n}\frac{z_j^2}{d_j-\lambda}$ we can ascertain the number and distribution of its zeros. To do this, observe that $f'(\lambda)=\rho\sum_{j=1}^{n}\frac{z_j^2}{(d_j-\lambda)^2}$ is always positive (as $\rho>0$), and this means that f is always increasing. Furthermore, we know that f has asymptotes at the points $\lambda=d_j$, for $j=1,\ldots,n$ and is continuous otherwise. This together with the previous observation means that f has a zero in each interval (d_j,d_{j+1}) for $j=1,\ldots,n-1$. Now we need to know only what happens in the intervals $(-\infty,d_1)$ and (d_n,∞). There can be no zero in the interval $(-\infty,d_1)$ because the formula for f shows that $f>0$ there. In the interval $(d_n,+\infty)$ we observe that as $\lambda\to\infty$, $f\to 1$ and when $\lambda\to d_n$ with $\lambda>d_n$, $f\to-\infty$. So there must be a zero of f greater than d_n.

Each of the n zeros $\lambda_1,\lambda_2,\ldots,\lambda_n$ of the function $1+\rho\sum_{j=1}^{n}\frac{z_j^2}{d_j-\lambda}$ makes $det(\tilde{D}-\lambda I)=0$ because of (3.8), and so $\lambda_1,\lambda_2,\ldots,\lambda_n$ are the n eigenvalues of the matrix $\tilde{D}$. We can deduce more about their location, in fact $\lambda_i<d_i+\rho$ for each i. This follows because $f(d_i+)<0$ and $f(d_i+\rho)>0$ and so the root λ_i is in the interval $(d_i,d_i+\rho)$, i.e., $\lambda_i<d_i+\rho$. To see that $f(d_i+\rho)>0$, observe first that we need consider only the case $d_i+\rho<d_{i+1}$, because if $d_{i+1}\le d_i+\rho$, then the assertion is trivially true, as $\lambda_i<d_{i+1}$. Now

$$f(d_i+\rho)=1+\rho\sum_{j=1}^{n}\frac{z_j^2}{d_j-d_i-\rho}=\sum_{j=1}^{n}\frac{z_j^2(d_j-d_i)}{d_j-d_i-\rho}>0$$

because $\sum_{j=1}^{n}z_j^2=1$, and d_j-d_i and $d_j-d_i-\rho$ have the same sign. Hence the complete graph of f assumes the shape shown in figure 3.1.

As stated above, by just observing the graph of f we know that each interval (d_i,d_{i+1}), $i=1,2,\ldots,n-1$ contains a root of f, and as seen above we can relate each eigenvalue λ_i to the corresponding element of D, in the following way

$$\lambda_i=d_i+\rho\mu_i \tag{3.11}$$

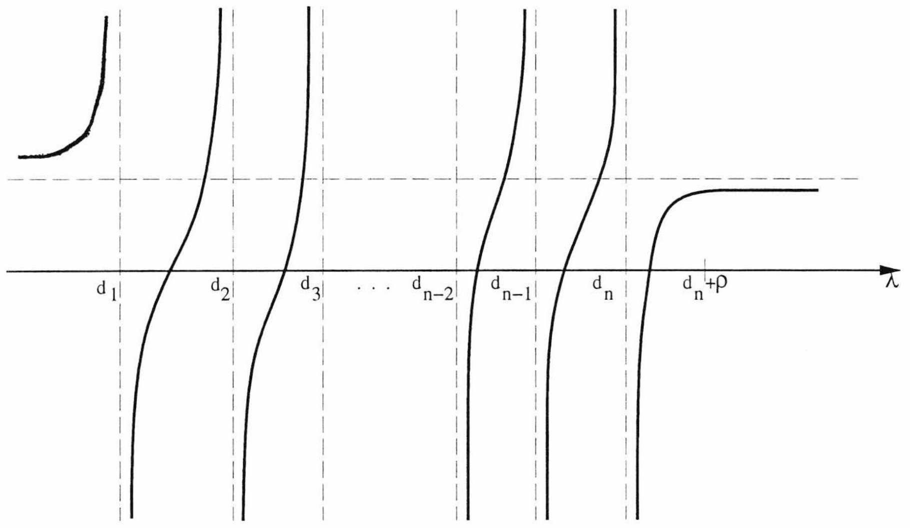

Figure 3.1: The graph of $f(\lambda) = 1 + \rho\sum_{j=1}^{n} z_j^2/(d_j - \lambda)$.

where $0 < \mu_i < 1$, because $d_i < \lambda_i < d_i + \rho$.

This result can also be seen as a consequence of a version of Weyl monotonicity theorem in which we consider the matrix $\tilde{D} = D + \rho z z^t$ (see [60], [87]). As the eigenvalues of $\tilde{D}$ are $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$; the eigenvalues of D are its own diagonal elements $d_1 \leq d_2 \leq \cdots \leq d_n$ and the eigenvalues of $\rho z z^t$ are $0, \ldots, 0, \rho$ where $\rho > 0$, Weyl's theorem gives

$$d_i + 0 \leq \lambda_i \leq d_i + \rho$$

that is $d_i \leq \lambda_i \leq d_i + \rho$, which implies that $\lambda_i = d_i + \rho\mu_i$ with $0 \leq \mu_i \leq 1$. As we already know that the d_i are strictly separated and $d_i < \lambda < d_{i+1}$, we can conclude that $0 < \mu_i \leq 1$.

Furthermore, $\sum_{i=1}^{n} \mu_i = 1$. To see why this is true, we recall that if $A = B + C$ then $trace(A) = trace(B) + trace(C)$ and applying this to the matrix $\tilde{D} = D + \rho z z^t$ and using $tr(\rho z z^t) = tr(\rho z^t z) = \rho$ we get:

$$\begin{aligned}(d_1 + \rho\mu_1) + \cdots + (d_n + \rho\mu_n) &= d_1 + \cdots + d_n + \rho \\ \rho(\mu_1 + \cdots + \mu_n) &= \rho \\ \mu_1 + \cdots + \mu_n &= 1\end{aligned}$$

Thus no $\mu_i = 1$, as this would imply all the $\mu_j = 0$ for $j \neq i$ and hence $0 < \mu_i < 1$.

As we shall see later, the above relations, true in exact arithmetic, cannot be relied upon in practice. When we depend on a computer with floating-point arithmetic, these results are no longer valid and the machine-representation for μ_i can be 0 and consequently, $\lambda_i = d_i$ or μ_i can be 1 and $d_i = d_{i+1}$, which is not possible in theory. If these conditions occur, then the method for the computation of the eigenvectors (see 3.14) breaks down. In the next chapter we show how we can overcome this difficulty.

Computing the eigenvectors Having found how the eigenvalues are characterized we find now a formula for computing the eigenvectors. Suppose (q, λ) is an eigenpair of $D + \rho z z^t$. Then we have

$$(D + \rho z z^t)q = \lambda q$$

Rearranging the terms we get

$$(D - \lambda I)q = -\rho z z^t q$$

Observing that $z^t q$ is a scalar, we can rewrite the above expression as

$$(D - \lambda I)q = -\rho(z^t q)z \tag{3.12}$$

Because each eigenvalue of $\tilde{D} = D + \rho z z^t$ coincides with no d_i, we can write

$$q = -\rho(z^t q)(D - \lambda I)^{-1} z \tag{3.13}$$

and to obtain a normalized eigenvector, we divide the vector by its norm. As $-\rho z^t q$ is simply a multiplicative scalar, a formula for the eigenvector is

$$q = \frac{(D - \lambda I)^{-1} z}{\| (D - \lambda I)^{-1} z \|} \tag{3.14}$$

In passing, note that having found the expression above for the eigenvectors of (3.7) we now have the more usual way of arriving at the equation for the eigenvalues. Pre-multiplying both sides of (3.13) by z^t we get

$$z^t q = -\rho(z^t q) z^t (D - \lambda I)^{-1} z$$

and as a careful analysis of the entries in q and z shows, $z^tq \neq 0$ (if $z^tq = 0$, (3.13) would give $q \equiv 0$) and we can divide the above equation by z^tq to obtain

$$1 = -\rho z^t(D - \lambda I)^{-1}z$$

or

$$1 + \rho z^t(D - \lambda I)^{-1}z = 0.$$

3.4 Deflation

There are cases in which we can considerably lower the order of the problem we have to solve. These include the cases where one or more of the z_j are zero or nearly equal to zero and where some of the d_j coincide. In these cases we say that we *deflate* the problem or we perform *deflation*.

But with the Divide-and-Conquer method, deflation is not simply a question of reducing the size of the problem: deflation is a necessary preliminary for the method to work. If we do not do the tests to check if deflation is needed and subsequently eliminate the elements of D for which $d_k = d_{k+1}$, or elements of z for which $z_k = 0$, we cannot compute the initial approximation for the eigenvalue. We do not have, in this case, a non-trivial interval (d_k, d_{k+1}) within which there is a root, because this interval reduces to a point (see next chapter for the method to find these roots).

In addition, it must be noted that deflation is not just something that happens by chance. As Parlett [60, page 260] observes and Cuppen [17] proves, there are some symmetric tridiagonal matrices for which deflation will certainly occur because few elements of the corresponding matrix of eigenvectors located in the last or first row will be non-negligible. This seems to happen in general with matrices whose entries are random numbers in the interval $(0, 1)$.

Recall that in the Divide-and-Conquer method we partition the original matrix T into two tridiagonal matrices T_1 and T_2. The vector z is formed from the last row of the matrix of eigenvectors (Q_1) of T_1 and from the first row of the matrix of eigenvectors (Q_2) of T_2. For deflation to happen, we look for small entries in the vector z. As both T_1 and T_2 are tridiagonal matrices, the phenomenon mentioned in the previous paragraph happens with both matrices of eigenvectors Q_1 and Q_2. This means that several elements in the last row of Q_1 and in the first row of Q_2 may be found to be very small and thus, the vector z itself, will have several small entries, which in turn means that a good amount of deflation will take place.

Cuppen also observes that matrices with constant diagonal and off-diagonal elements do not satisfy the criteria which assures the occurrence of small elements in the first and last rows of a tridiagonal matrix and so, are not likely to give rise to much deflation.

We have run some tests, using MATLAB [54], where we constructed a symmetric tridiagonal matrix with random entries. We observed from these experiments that there are frequently some very small entries in the first and last rows of the matrix of eigenvectors. The number of small entries in these rows seems to get proportionally larger as the order of the matrix increases. This phenomenon also seems to occur in other rows close to the extremes of the matrix. To be more specific, when we did tests with matrices of order $n = 40$ and $n = 50$ we obtained 4 elements smaller than 10^{-12} in absolute value in the first row. For the same matrices, 6 elements were smaller than 10^{-12} in the last row of the matrices of eigenvectors. For $n = 70$ there were 8 small elements in the first row and 9 small elements in the last row. For $n = 100$ the number of small entries increased considerably and there were 55 elements smaller than 10^{-12} in the first row and 34 small elements in the last row.

3.4.1 Cases where deflation should be performed

We will see now how we can always reduce the given problem to a smaller one where all the z_j are non-zero and all the d_j are distinct (at least to some desired accuracy).

Case 1 : $z_j = 0$. In this case we have

$$[(D + \rho z z^t)q]_j = d_j q_j$$

i.e., the j-th component of $\tilde{D}q$ is $d_j q_j$. So, if we choose q to be e_j, we notice that e_j is an eigenvector, with d_j as its associated eigenvalue.

Case 2 : $d_j = d_{j+1}$. We observe here that if we apply a Givens rotation in the $(j, j+1)$ plane (see [30]) to the matrix D, then D is not modified by it. This comes from the fact that we are applying a combination of sines and cosines to two elements that have the same value, so that we have

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} d & \\ & d \end{bmatrix} \begin{bmatrix} c & -s \\ s & c \end{bmatrix} = \begin{bmatrix} (c^2+s^2)d & -scd+scd \\ -scd+scd & (c^2+s^2)d \end{bmatrix} = \begin{bmatrix} d & \\ & d \end{bmatrix}$$

We choose a rotation J such that the $(j+1)^{st}$ component of z will be transformed to zero, that is we choose c and s so that $cz_{j+1} - sz_j = 0$ and hence we reduce to Case 1. In other words, whenever we have two equal d_j we can make one of the corresponding elements of z zero without altering D itself. We may then take the $(j+1)^{st}$ eigenvector to be e_{j+1} and the associated eigenvalue to be d_{j+1}. It must be observed, however, that the solution found for this modified matrix requires further computation before it is the solution to the original problem (see Section 3.4.2).

After recognizing that one of these cases occurs, and computing the necessary rotations to zero out the elements of z involved (if necessary), the zero elements of z are translated to the bottom of the vector and, simultaneously, corresponding elements of D are similarly displaced. These zeros can be the ones that were originally present (Case 1) or can be the ones created (Case 2). We then continue by working with the upper part of $D + \rho z z^t$, in which all the z_j's are non-zero and the d_j's are all different.

There is a third case, which can occur, and which reduces the problem to a very trivial one. This is the situation when $|z_j| = 1$ for some j. As we have normalized z so that

$||z|| = 1$, in this case all the other z_k are zero and we are reduced to Case 1. If this situation occurs, the problem may be immediately solved, since the eigenvalues will be the d_k themselves with eigenvectors e_k except for the j^{th} eigenvalue λ_j which equals $d_j + \rho$ (with corresponding eigenvector e_j).

3.4.2 Performing deflation changes the original problem

Recall that our aim, after the first partition of the original problem, is to find the diagonal decomposition of

$$D + \rho z z^t. \tag{3.15}$$

However, as we have just seen, there are cases in which *deflation* is needed, and the necessary operations to perform it modify the matrix $D + \rho z z^t$. Prior to deflation, $D + \rho z z^t$ must be modified by a permutation so as to order the elements of D, but as this process is analogous to the permutations considered here, we will omit it from our explanation.

Recalling that the operations referred to in this subsection are, in order, the rotations mentioned in Case 2 and the subsequent permutations to ensure that the non-zero z_i's appear at the top of the vector z and distinct d_i's in the top left corner of the matrix D, effectively what we compute is the eigensystem of

$$\Pi^t J^t (D + \rho z z^t) J \Pi \tag{3.16}$$

Here J represents a product of rotation matrices and Π a product of permutations. The eigen-decomposition of (3.16) is given, say, by

$$\Pi^t J^t (D + \rho z z^t) J \Pi = \tilde{X} \Lambda \tilde{X}^t \tag{3.17}$$

and as we want the decomposition for $D + \rho z z^t$, what we need is

$$(D + \rho z z^t) = J \Pi \tilde{X} \Lambda \tilde{X}^t \Pi^t J^t \tag{3.18}$$

This means, that after computing the eigenvectors of (3.16), we need to pre-multiply the eigenvectors matrix $\tilde{X}$ by $J\Pi$ to obtain the eigenvectors of $D + \rho z z^t$. In practice these products are not calculated in the usual way as matrices products, but instead we exploit the special structure of the component matrices involved (see Chapter 5).

3.4.3 When shall we deflate?

According to our previous analysis, it is essential to deflate when two elements of D are equal or when an element of z is zero. However, when carrying out calculations, we work with finite-precision arithmetic, and we must allow for deflation also when these conditions nearly occur, i.e., when elements of D are almost equal or when elements of z are nearly zero. That is, we should allow that deflation will be necessary whenever $|d_{i+1} - d_i| < tol$ or $|z_i| < tol$, where tol is a tolerance or threshold value to be established. One problem is how this tolerance is to be determined and we consider whether it is possible to deflate in more cases than an expected choice for the tolerance (i.e., a small multiple of the machine-epsilon) would allow. In any case, it is to be anticipated that this tolerance should be related to the norm of the matrix we are working on.

As we have seen in Section 3.4.1, if $z_j = 0$, then d_j is an eigenvalue and e_j is a corresponding eigenvector. (A fact that we have already used in the situation where $d_i = d_{i+1}$.) Following the discussion given by Dongarra and Sorensen [20] we shall determine in general, when the pair (d_j, e_j) is a good approximation to the eigenpair we are looking for. For this, observe that

$$||(D + \rho z z^t)e_j - d_j e_j|| = |\rho z_j| \, ||z|| = |\rho z_j|$$

(recall that $||z|| = 1$). In this way we can use (d_j, e_j) as an eigenpair for the matrix $D + \rho z z^t$ when the following condition is satisfied

$$|\rho z_j| \leq tol$$

where $tol > 0$ is the tolerance for the error we are willing to accept in using the residual as a measure of how good the approximate eigenpair is. Another way of looking at this issue is following Parlett's Theorem (4-5-1) [60, page 69]. It asserts that for any scalar σ and any nonzero vector x, there is an eigenvalue λ of A that satisfies

$$|\lambda - \sigma| \leq ||Ax - x\sigma||/||x||$$

In our case, we want to know how good the pair (d_i, e_i) is as an approximation for the eigenpair, so that we take $\sigma = d_i$, x as e_i and $A = D + \rho zz^t$. Thus, $Ax - x\sigma$ is $(D + \rho zz^t)e_i - e_i d_i = d_i e_i + \rho z z_i - d_i e_i = \rho z_i z$ whence by Parlett's Theorem there is an eigenvalue λ of $D + \rho zz^t$ such that

$$|\lambda - d_i| \leq ||Ae_i - d_i e_i|| = \rho|z_i|.$$

Hence if we make $\rho|z_i|$ small enough, or less than a certain tolerance, then d_i will approximate an eigenvalue of $D + \rho zz^t$ within this same tolerance.

In our early work we simply used $macheps$, the machine epsilon, for tol, but from experience, we noticed the need to use a multiplier indicating the order of magnitude of the elements on which we are working (usually the norm of the matrix itself). In practice, the computation of the norm of a matrix may be very time consuming and we employ an easier, approximate, bound. Dongarra and Sorensen suggest using

$$tol = macheps \times \eta(max(|d_1|, |d_n|) + |\rho|)$$

where η is a constant of order unity. Note that the quantity $max(|d_1|, |d_n|) + |\rho|$ is in reality an approximation for the norm of $D + \rho zz^t$, as this is a symmetric matrix and its norm is given by $max(|\lambda_1|, |\lambda_n|)$. Using the Gerschgorin Theorem (see [36]) or more simply, just observing the locations of the eigenvalues of $D + \rho zz^t$, we know that each eigenvalue λ satisfies

$$|\lambda| \leq max_i(|d_i| + |\rho|), \quad \text{for } 1 \leq i \leq n$$

As the d_i's are ordered, $max_i(|d_i| + |\rho|) = max(|d_1|, |d_n|) + |\rho|$ and hence

$$norm(D + \rho zz^t) = max(|\lambda_1|, |\lambda_n|) \leq max(|d_1|, |d_n|) + |\rho|$$

However, the cited criterion does not always define the optimal test for deflation. We considered whether we could do deflation more frequently, by relaxing the criterion, but this approach did not seem to work in all cases. In Chapter 6, where we report the results obtained using our implementation, we show some examples of the different situations that can occur when we change that tolerance which determines if an element of z can be considered to be zero. This issue deserves further investigation. Theorem (11-7-1) in Parlett ([60]) gives a hint that we might use the approximation $\lambda \approx d_i$ for eigenvalues even when $\rho|z_i|$ exceeds tol, but we must take care in cases where the eigenvalues are very close together. The relevant theorem (11-7-1) asserts that

> if y is a unit vector, $\theta = \rho(y)$ is its Rayleigh quotient and we consider α as the eigenvalue of A which is closest to θ and z the corresponding eigenvector, then $|\theta - \alpha| \leq ||r(y)||^2/\gamma$, where $r(y)$ is the residual $Ay - \theta y$ and γ is the minimum distance between θ and the other eigenvalues of A with exception of α.

3.5 Dividing the original matrix in more parts

We have explained how the Divide-and-Conquer method works when the original matrix is partitioned into two parts. Even though partitioning into more than two parts does not change the method, a clearer understanding of the full divide-and-conquer process can be gained if we show what happens when we work with four sub-matrices instead of just two. Note that the total number of parts into which we divide the original matrix, corresponds directly, in our implementation, to the number of processors in use. In this way, the greater the number of processors used, the greater the number of necessary reassembling steps. The meaning of the expression *steps* will become clearer from the following description.

Consider the tridiagonal matrix T as in (3.1). We partition it initially into two parts as in (3.2) and then, each of the two matrices T_1 and T_2 is divided again in two parts. For clarity we will consider in this discussion that the order, n, of the matrix T is divisible by 4, but any dimension can be treated; the manner in which we then chose the dimensions of the sub-matrices will be explained in Chapter 5. We thus obtain

$$T = \begin{bmatrix} \begin{bmatrix} T_{11} & \\ & T_{12} \end{bmatrix} + \alpha_1 b_1 b_1^t & \\ & \begin{bmatrix} T_{21} & \\ & T_{22} \end{bmatrix} + \alpha_2 b_2 b_2^t \end{bmatrix} + \alpha b b^t \tag{3.19}$$

where

$$T_{11} = \begin{bmatrix} \tau_1 & \sigma_1 & & & \\ \sigma_1 & \tau_2 & \sigma_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \sigma_{(n/4)-2} & \tau_{(n/4)-1} & \sigma_{(n/4)-1} \\ & & & \sigma_{(n/4)-1} & \tau_{n/4} - \sigma_{n/4} \end{bmatrix}$$

$$T_{12} = \begin{bmatrix} \tau_{(n/4)+1} - \sigma_{n/4} & \sigma_{(n/4)+1} & & & \\ \sigma_{(n/4)+1} & \tau_{(n/4)+2} & \sigma_{(n/4)+2} & & \\ & \ddots & \ddots & \ddots & \\ & & \sigma_{(n/2)-2} & \tau_{(n/2)-1} & \sigma_{(n/2)-1} \\ & & & \sigma_{(n/2)-1} & \tau_{n/2} - \sigma_{n/2} \end{bmatrix}$$

with T_{21} and T_{22} defined in a similar way. The element $T((n/2)+1,(n/2)+1)$ which was assigned to T_{21} is changed to $\tau_{(n/2)+1} - \sigma_{n/2}$, the element $T(3n/4, 3n/4)$ (which is also in T_{21}) changes to $\tau_{3n/4} - \sigma_{3n/4}$. T_{22} has only one element modified:
$T((3n/4)+1,(3n/4)+1)$ changes to $\tau_{(3n/4)+1} - \sigma_{3n/4}$. The scalars α_1 and α_2 are the elements $T(n/4,(n/4)+1) = \sigma_{n/4}$ and $T(3n/4,(3n/4)+1) = \sigma_{3n/4}$, respectively; b_1 and b_2 are vectors of appropriate dimension (in this case $n/2$) which have all the components zero except those in positions $n/4$ and $(n/4)+1$ which are equal to 1.

We then find the eigensystems of each matrix T_{ij} using *QR/QL*, obtaining decompositions $T_{ij} = Q_{ij}D_{ij}Q_{ij}^t$ giving

$$T = \begin{bmatrix} \begin{bmatrix} Q_{11}D_{11}Q_{11}^t & \\ & Q_{12}D_{12}Q_{12}^t \end{bmatrix} + \alpha_1 b_1 b_1^t & \\ & \begin{bmatrix} Q_{21}D_{21}Q_{21}^t & \\ & Q_{22}D_{22}Q_{22}^t \end{bmatrix} + \alpha_2 b_2 b_2^t \end{bmatrix} + \alpha b b^t \tag{3.20}$$

as before, the expression above is transformed to

$$T = \begin{bmatrix} \begin{bmatrix} Q_{11} & \\ & Q_{12} \end{bmatrix} (D\rho z z_1) \begin{bmatrix} Q_{11} & \\ & Q_{12} \end{bmatrix}^t & \\ & \begin{bmatrix} Q_{21} & \\ & Q_{22} \end{bmatrix} (D\rho z z_2) \begin{bmatrix} Q_{21} & \\ & Q_{22} \end{bmatrix}^t \end{bmatrix} + \alpha b b^t \tag{3.21}$$

where

$$D\rho z z_1 = \begin{bmatrix} D_{11} & \\ & D_{12} \end{bmatrix} + \rho_1 z_1 z_1^t, \tag{3.22}$$

$$D\rho z z_2 = \begin{bmatrix} D_{21} & \\ & D_{22} \end{bmatrix} + \rho_2 z_2 z_2^t. \tag{3.23}$$

The first $n/4$ components of z_1 are the entries of the last row of Q_{11} and the next $n/4$ components are the entries of the first row of Q_{12}. z_2 is formed in a similar way: its first $n/4$ components are the entries of the last row of Q_{21} and the next $n/4$ components come from the first row of Q_{22}. $\rho_1 = 2\alpha_1$ and $\rho_2 = 2\alpha_2$.

In terms of the decomposition of each $D\rho z z_i$, we obtain

$$T = \begin{bmatrix} \begin{bmatrix} Q_{11} & \\ & Q_{12} \end{bmatrix} P_1\Lambda_1 P_1^t \begin{bmatrix} Q_{11} & \\ & Q_{12} \end{bmatrix}^t & \\ & \begin{bmatrix} Q_{21} & \\ & Q_{22} \end{bmatrix} P_2\Lambda_2 P_2^t \begin{bmatrix} Q_{21} & \\ & Q_{22} \end{bmatrix}^t \end{bmatrix} + \alpha b b^t \tag{3.24}$$

or

$$T = \begin{bmatrix} Q_1P_1 & \\ & Q_2P_2 \end{bmatrix} \left(\begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix} + \rho z z^t \right) \begin{bmatrix} Q_1P_1 & \\ & Q_2P_2 \end{bmatrix}^t \tag{3.25}$$

where

$$Q_1 = \begin{bmatrix} Q_{11} & \\ & Q_{12} \end{bmatrix} \tag{3.26}$$

$$Q_2 = \begin{bmatrix} Q_{21} & \\ & Q_{22} \end{bmatrix} \tag{3.27}$$

Again, z and ρ are found as before

$$\begin{aligned} z &= [\textit{last row of } Q_1P_1; \textit{first row of } Q_2P_2]^t && (3.28) \\ &= [\textit{last row of } Q_{12}P_1; \textit{first row of } Q_{21}P_2]^t && (3.29) \end{aligned}$$

and $\rho = 2\alpha$.

To complete the method, we need once more to find the eigensystem of the central matrix

$$\begin{bmatrix} \Lambda_1 & \\ & \Lambda_2 \end{bmatrix} + \rho z z^t = X \Lambda X^t \tag{3.30}$$

in order to obtain the eigenvalue/eigenvector decomposition of T, namely

$$T = \begin{bmatrix} Q_1P_1 & \\ & Q_2P_2 \end{bmatrix} X \Lambda X^t \begin{bmatrix} Q_1P_1 & \\ & Q_2P_2 \end{bmatrix}^t \tag{3.31}$$

The eigenvectors of T are then the columns of the product of matrices given by

$$\begin{bmatrix} Q_1P_1 & \\ & Q_2P_2 \end{bmatrix} X \tag{3.32}$$

and the eigenvalues of T are clearly the diagonal entries of Λ.

Note that we do not carry out the partition first into two parts and then each part into two more, but partition directly into four pieces (or 8,16,..., as the case may be). We have described it as a two-stage process in order to make clear that for the "middle" matrices (T_{12}

and T_{21} in our case), two elements will be modified and not just one as might appear from the division into just two parts. Another point to remark and which is important for the parallel implementation, is the way the computations can be organized. In our example we can see that the *QR/QL* algorithm is applied to four sub-matrices independently. The results obtained from each independent computation are combined together two by two and used to form matrices which have the form $D + \rho zz^t$ ($D\rho zz_1$ and $D\rho zz_2$ in our example). The eigensystems of these matrices are found by solving the secular equation (originating from 3.10) and then computing the eigenvectors by formula (3.14). To find the eigenvectors of matrices T_1 and T_2 (which have dimension $n/2$ each), we need to pre-multiply each matrix of eigenvectors just computed (P_1 or P_2) by the matrix formed from the eigenvectors computed by QR/QL (matrices Q_1 or Q_2, respectively). Note that as Q_1 and Q_2 are block-diagonal matrices, we can take advantage of this structure to compute the product mentioned. We then again

1. form a "$D + \rho zz^t$" matrix (matrix (3.30) at this stage)
2. compute its eigensystem ($X\Lambda X^t$) in the way previously mentioned (secular equation and the eigenvector formula)
3. compute the product of the matrix of eigenvectors calculated at this stage with the matrix formed by blocks of eigenvectors matrices that come from the previous step (which corresponds to the product represented in (3.32), where the latter matrix is the left-hand matrix shown in (3.32)), and finally obtain the eigensystem of the initial matrix T.

To sum up, we can see that in the case of the partition into four sub-problems, we need to form two $D\rho zz$ matrices of order $n/2$ in a first step (matrices $D\rho zz_1$ and $D\rho zz_2$), compute their eigenvectors by formula (3.14) and update them to obtain the eigenvectors of tridiagonal matrices of order $n/2$ (matrices T_1 and T_2). This updating is done by computing products of matrices Q_1P_1 and Q_2P_2, respectively. In a second step, we repeat these tasks again, but we work with just one "big" matrix of order n.

As can be observed, in each step, the order of each sub-matrix doubles (or the two previous orders are added) and the number of distinct sub-problems is decreased by half. To make it clearer, we can observe that in our example, we had initially four matrices of order $n/4$ each. These matrices had their eigensystems computed and the information obtained from them was gathered in pairs, to form two matrices of order $n/2$ each. In a final stage these two matrices have again their eigensystems computed and data from them is joined so that we obtain one whole matrix (matrix (3.30)) of order n and need again to compute its eigensystem to be able to obtain the eigensystem of the original matrix T.

We can see that sequences of operations/computations are repeated from time to time. This is what we shall call a *step*. In other words, with the exception of the first computation of eigensystems by the QR/QL algorithm, which occurs just once, the other tasks are repeatedly applied to compute matrices which are twice the size of the preceding ones. The main tasks of a typical *step* are then

- Form the matrix $D + \rho zz^t$ where the elements of D come from the eigenvalues of two adjacent matrices computed in the previous step, and z is derived from the eigenvectors of these same matrices,
- Compute the eigensystem of the matrix $D + \rho zz^t$ using the secular equation method and the formula for the eigenvectors,
- Compute the product of two matrices of eigenvectors, the first formed from two matrices of eigenvectors computed in the previous step and the second from the eigenvectors computed in the above item

These three tasks become clearer if the reader follows the previous diagrams (3.19) to (3.32). It can then be seen that the number of steps needed is $\log_2 p$, where p is the number of initial partitions used. So that in our example we took two steps to complete the algorithm.

Chapter 4

The Solution of the Secular Equation

4.1 The solution of $f(\lambda) = 1 + \rho \sum_{j=1}^{n} \frac{z_j^2}{d_j - \lambda} = 0$

As we saw before, to find the eigenvalues of $D + \rho z z^t$ we need to find the zeros of the function $f(\lambda) = 1 + \rho \sum_{j=1}^{n} \frac{z_j^2}{d_j - \lambda}$. Knowing the intervals in which the zeros are located and how they are related to the entries of D, enables us to provide a good method for accurately determining these zeros. First we suppose that all the z_j are non-zero and that the d_j are all distinct.

We modify slightly the function f, in order to obtain more accuracy when computing the eigenvectors and indeed to be able to compute the eigenvectors at all in certain difficult cases (see [13] and Sections 4.4 and 4.5).

Observing that whenever $\lambda = d_i + \rho\mu_i$ is a zero of f, we have that μ_i is a zero of

$$f_i(\mu) = 1 + \sum_{j=1}^{n} \frac{z_j^2}{\delta_j - \mu} \tag{4.1}$$

where $\delta_j = (d_j - d_i)/\rho$, we can work with the functions (4.1) and compute in each case the zero which is in the interval $(0, \delta_{i+1})$. From now on, for simplicity of notation we will put $\delta = \delta_{i+1}$.

So, what we need now is an accurate method to find the zeros of (4.1). Bunch, Nielsen and Sorensen [13] devised a method based on simple rational approximants which has a quadratic rate of convergence and in practice usually converges in very few iterations (four or five).

First we split the sum that appears in function f_i, into two parts. The first part is the sum of terms up to index i and the other one the sum of the remaining terms. This will be done for $1 \leq i < n$ and the functions have the following form

$$\psi(t) \equiv \sum_{j=1}^{i} \frac{z_j^2}{(\delta_j - t)} \tag{4.2}$$

$$\varphi(t) \equiv \sum_{j=i+1}^{n} \frac{z_j^2}{(\delta_j - t)} \tag{4.3}$$

This separation lies at the heart of the Bunch *et al.* method for finding a zero of the function f_i. Instead of looking for the zero of the whole function, we try to locate the point of intersection of the graphs of the functions $-\psi$ and $1 + \varphi$ in the interval considered. In other words, we try to solve

$$-\psi(t) = 1 + \varphi(t) \tag{4.4}$$

The idea of the method is to use two curves with simpler expressions which approximate the curves ψ and φ as in the graph shown in Figure 4.1. These curves (we will call them $\bar{\psi}$ and $\bar{\varphi}$) are tangential to ψ and φ at an initial approximation t_0 to the root μ_i. The next iteration point, t_1, is the intersection of the curves $-\bar{\psi}$ and $1 + \bar{\varphi}$ and the procedure is applied iteratively using t_1 as the initial point for the next stage. We can see that the next approximation t_1 is bigger than t_0 and is getting closer to the root μ_i which we are looking for.

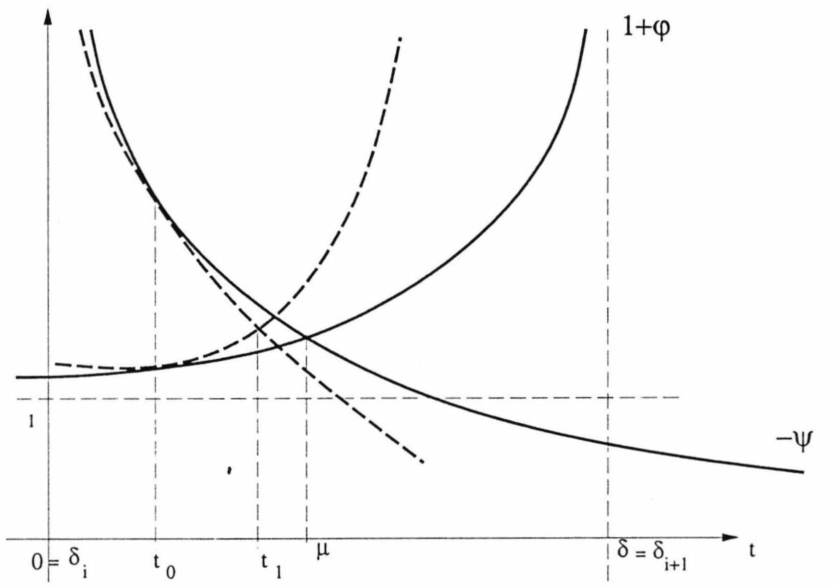

Figure 4.1: The curves $1+\varphi$ and $-\psi$ and their approximations $1+\bar{\varphi}$ and $-\bar{\psi}$.

We can define these simpler rational functions as below

$$\begin{aligned}\bar{\psi}(t) &= \frac{p}{q-t}\\ \bar{\varphi}(t) &= r+\frac{s}{\delta-t}\end{aligned}$$

Then

$$\begin{aligned}\bar{\psi}'(t) &= \frac{p}{(q-t)^2}\\ \bar{\varphi}'(t) &= \frac{s}{(\delta-t)^2}\end{aligned}$$

We choose $-\bar{\psi}$ to have the same ordinate and gradient as $-\psi$ at $t=t_0$ so that

$$\frac{p}{q-t_0}=\psi(t_0)$$

and

$$\frac{p}{(q-t_0)^2}=\psi'(t_0)$$

Similarly for $\bar{\varphi}$ and φ,

$$r + \frac{s}{\delta - t_0} = \varphi(t_0)$$

and

$$\frac{s}{(\delta - t_0)^2} = \varphi'(t_0)$$

Calling $\psi(t_0) = \psi_0$, $\psi'(t_0) = \psi_0'$ and so on we find after some easy calculations that the quantities p, q, r and s are given by

$$\begin{aligned} p &= \psi_0^2/\psi_0' && (4.5)\\ q &= t_0 + \psi_0/\psi_0' && (4.6)\\ r &= \varphi_0 - (\delta - t_0)\varphi_0' && (4.7)\\ s &= (\delta - t_0)^2\varphi_0' && (4.8) \end{aligned}$$

The new approximation t_1 is then the intersection of the curves $-\bar{\psi}$ and $1 + \bar{\varphi}$. This means that t_1 is a solution of

$$-\frac{p}{q - t} = 1 + r + \frac{s}{\delta - t} \qquad (4.9)$$

It is easily seen that this equation can be transformed into a quadratic equation. After some manipulations of (4.9) and setting $u = t - t_0$ we arrive at

$$u^2 - au + b = 0 \qquad (4.10)$$

where

$$\begin{aligned} a &= \Delta + \frac{\psi_0}{\psi_0'} + \frac{s + p}{1 + r}\\ b &= \Delta\frac{\psi_0}{\psi_0'} + \frac{s\frac{\psi_0}{\psi_0'} + p\Delta}{1 + r} \end{aligned}$$

and we have written $\Delta = \delta - t_0$. Using the values for p, q, r and s given previously, we can simplify these expressions for a and b and obtain

$$a = \frac{1}{c}(\Delta(1+\varphi_0) + \frac{\psi_0^2}{\psi_0'}) + \frac{\psi_0}{\psi_0'} \tag{4.11}$$

$$b = \frac{1}{c}(\Delta w \frac{\psi_0}{\psi_0'}) \tag{4.12}$$

where $c = 1 + r = 1 + \varphi_0 - \Delta\varphi_0'$ and $w = 1 + \varphi_0 + \psi_0$.

Finally, we need to be careful about which solution we select. The root we are looking for must be positive and smaller than δ. Looking at the graph of the two curves for which we are trying to find the intersection point (see equation (4.9)) and observing that the quantity $c = 1 + r$ is positive, we see that we want the smaller of the two roots.

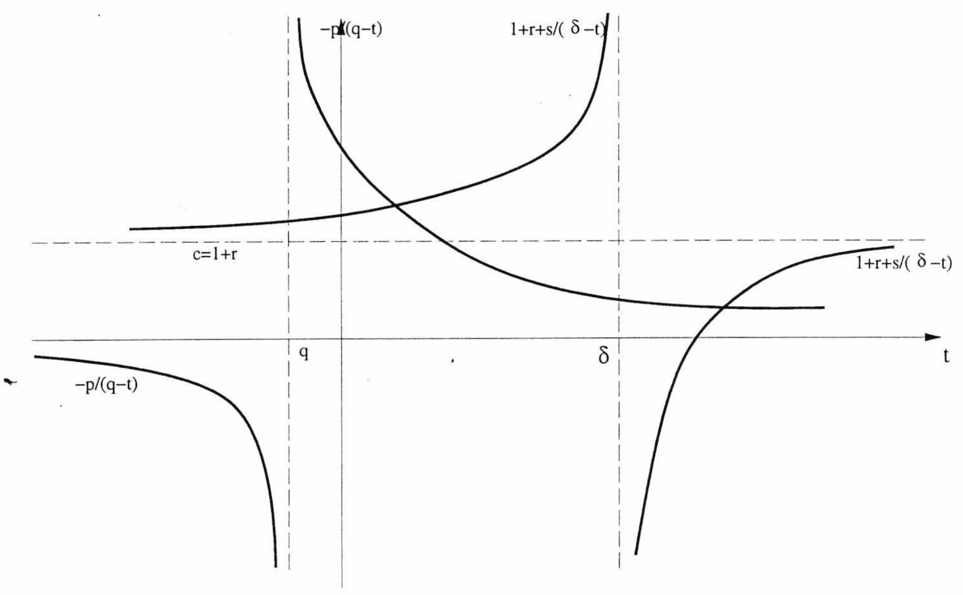

Figure 4.2: Possible intersections of the curves $1 + r + s/(\delta - t)$ and $-p/(q - t)$.

We can see that the term $c = 1 + r > 0$ by writing out the expressions that compose it and grouping appropriate terms together.

$$\begin{aligned} c = 1 + r = 1 + \varphi_0(t) - \Delta\varphi_0'(t) &= 1 + \sum_{j=i+1}^{n} \frac{z_j^2}{\delta_j - t_0} - (\delta_{i+1} - t_0) \sum_{j=i+1}^{n} \frac{z_j^2}{(\delta_j - t_0)^2} \\ &= 1 + \sum_{j=i+1}^{n} \frac{z_j^2}{(\delta_j - t_0)^2}(\delta_j - \delta_{i+1}) \end{aligned}$$

As $\delta_j \geq \delta_{i+1}$ for $j \geq i+1$, we have $\delta_j - \delta_{i+1} \geq 0$, and as all other terms are positive, this proves that $1 + r > 0$.

In addition, because it is important not to lose accuracy when computing this root, we must adopt a suitable formula for solving the quadratic equation. Here, we only try to avoid cancellation when adding or subtracting a in the formula for the root. We could take care of other possible problems such as overflow in the computation of $a^2 - 4b$, but it did not seem necessary.

In our case, since a is positive (see proof in next paragraph), we use

$$u = \frac{2b}{a + \sqrt{a^2 - 4b}}$$

or, as we want the value for $t = t_1$ and $u = t - t_0$

$$t_1 = t_0 + \frac{2b}{a + \sqrt{a^2 - 4b}} \tag{4.13}$$

To prove that a is positive, we must first observe that the quantity $1 + \varphi(t_0) + \psi(t_0)$ is negative. For this, recall that at the root μ, $1 + \varphi(\mu) + \psi(\mu) = 0$, so that we can write

$$1 + \varphi(t_0) + \psi(t_0) = 1 + \varphi(t_0) + \psi(t_0) - (1 + \varphi(\mu) + \psi(\mu)).$$

But this is the same as

$$\begin{aligned} 1 + \varphi(t_0) + \psi(t_0) &= \sum_{j=1}^{n} \frac{z_j^2}{\delta_j - t_0} - \frac{z_j^2}{\delta_j - \mu} \\ &= \sum_{j=1}^{n} \frac{z_j^2}{(\delta_j - t_0)(\delta_j - \mu)}(t_0 - \mu) \end{aligned}$$

as $t_0 - \mu < 0$ and $(\delta_j - t_0)(\delta_j - \mu) > 0$ (see Fig. 4.1), we have proved $1 + \varphi_0 + \psi_0 < 0$. Now, a is given by 4.11, which can be re-written as

$$a = \frac{\Delta(1 + \varphi_0)}{c} + \frac{\psi_0}{\psi_0'}\frac{(\psi_0 + c)}{c}.$$

As $\Delta(1 + \varphi_0)/c > 0$ (because each of its terms are positive) and ψ_0' and c are also both positive, if we show that $\psi_0(\psi_0 + c) > 0$, then we will have proved that $a > 0$. As $\psi_0 < 0$, we need only to show that $\psi_0 + c < 0$. Now,

$$\psi_0 + c = \psi_0 + 1 + \varphi_0 - \Delta\varphi_0'. \tag{4.14}$$

Since Δ and φ'_0 are both greater than zero and as shown before, $1+\psi_0+\varphi_0 < 0$, then ψ_0+c is negative as required and consequently, a is positive.

Subsequent iterations (or approximations to the root μ_i) are found in exactly the same way as t_1 was computed from t_0. We now compute p, q, r and s with t_1 in the place of t_0 and the next iteration t_2 will be computed with these new values, using formula (4.13), with t_2 in place of t_1 and t_1 in place of t_0 wherever these variables appear.

The iterations continue until some criteria for stopping them is satisfied. As the method was shown to converge quadratically to the root, we can use a simple criteria which determines if two sucessive iterations are computationally the same. In other words, if t_k is an approximation for the root and t_{k+1} is the next computed approximation which uses the previous iterate, we can then check if $|t_{k+1} - t_k| \leq \epsilon$, where ϵ is some tolerance to be determined, as for example the machine-epsilon. Or, using a relative measure, we can stop the iterations if $|t_{k+1} - t_k| \leq \epsilon |t_k|$. This last inequality, however, while accounting for relative errors, which is a common practice in iterative algorithms, can be a very stringent criteria if the quantities t_k and t_{k+1} are very small. If, for example, they are of order 10^{-20} and the machine-epsilon is $O(10^{-16})$, we are requiring that the difference between two successive iterations is of order 10^{-36}. In some cases this relative difference is not achieved, and the iterations can run forever if another criteria is not used as well. In our case, this other criteria was simply to limit the number of iterations. If this number is, say, greater than or equal to 30, we consider the last iteration as being the root sought and stop the search for that root. Bunch et al. suggest we use a combination of the relative criteria given above and another measure to assess if the quantity $|1 + \psi(t_k) + \varphi(t_k)|$ is relatively small. We did not use that because it would imply performing some extra calculations, but we intend to include it in future implementations to observe how it compares with the criterion which limits the number of iterations.

As we based our method on the graph shown in Figure 4.1, we need to verify that the rational functions chosen satisfy the conditions we assumed of them. In other words, we need to show that

$$\frac{-p}{q-t} < -\psi(t) \tag{4.15}$$

and

$$r + \frac{s}{\delta - t} > \varphi(t) \tag{4.16}$$

for $0 < t < \delta$, $t \neq t_0$. This is undertaken in the following section.

4.2 Why the method for finding the roots of the rational equation works – some proofs of convergence

Here, we prove some results that will show that the method described in the previous section converges to the solution of equation (4.4) and that this convergence is quadratic. We begin by showing that the curves chosen satisfy conditions (4.15) and (4.16). Initially the proofs we had written were adapted from Bunch *et al* [13]. However, even for those we have introduced a number of simplifications, and in the case of Result 2 we supplied a new argument, where their original proof is in error. The proofs provided now for Result 1 and 2 are completely different from those presented by Bunch et al. They are a great deal simpler than the ones we had before. Though the proof of Result 3 uses a slightly more complex argument, the volume of calculations is greatly reduced, giving a clearer idea of what is being done. Nonetheless we have included our original versions of Bunch's proofs and show them at the end of this chapter. These proofs were first set out in [24] as well as the proof for Result 3 that appears here.

Result 1 *Suppose* $0 < t_0 < \mu_i$ *is given and* p, q, r *and* s *are defined as before. Then*

$$\frac{-p}{q-t} < -\psi(t) \tag{4.17}$$

and

$$r + \frac{s}{\delta - t} > \varphi(t) \quad \textit{for} \quad 0 < t < \delta = \delta_{i+1} = (d_{i+1} - d_i)/\rho,\ t \neq t_0. \tag{4.18}$$

We first prove (4.17), but this is the same as proving $p/(q-t) > \psi$ or, graphically, that the curve $\bar{\psi} = p/(q-t)$ is above the curve ψ.

For this, let us first make some observations about the function ψ (see expression 4.2):

1. ψ is a negative increasing function: increasing because

$$\psi'(t) = \sum_{j=1}^{i} \frac{z_j^2}{(\delta_j - t)^2} \geq 0,$$

and $\psi(t) \leq 0$, because $\delta_j - t < 0$ for $j \leq i$.

2. ψ is concave:

$$\psi''(t) = 2\sum_{j=1}^{i} \frac{z_j^2}{(\delta_j - t)^3} \leq 0,$$

consequently, the tangent line is above ψ.

3. (a) We have, by the definition of $\bar{\psi}$:

$$\bar{\psi}(t_0) = \psi(t_0) \text{ and } \bar{\psi}'(t_0) = \psi'(t_0),$$

in other words, the functions ψ and $\bar{\psi}$ touch at t_0.

(b) If we compare $\frac{1}{\psi(t)}$ and $\frac{1}{\bar{\psi}(t)}$ we can see that these two functions also touch at t_0 because

$$\frac{1}{\psi(t_0)} = \frac{1}{\bar{\psi}(t_0)}$$
$$\left(\frac{1}{\psi(t)}\right)'_{t=t_0} = -\frac{\psi'(t)}{\psi(t)^2}\bigg|_{t=t_0} = -\frac{\psi'(t_0)}{\psi(t_0)^2} = -\frac{\bar{\psi}'(t_0)}{\bar{\psi}(t_0)^2} = \left(\frac{1}{\bar{\psi}(t)}\right)'_{t=t_0}$$

(c)

$$\frac{1}{\bar{\psi}(t)} = \frac{q-t}{p} = \frac{-t}{p} + \frac{q}{p}$$

(d) From (b) and (c) above, $1/\bar{\psi}$ and $1/\psi$ have the same tangent at $t = t_0$, but as $1/\bar{\psi}$ is a line, its tangent is itself. This means that the tangent line of $1/\psi$ at $t = t_0$ is given by

$$y(t) = \frac{-t+q}{p}.$$

4. We want to prove that $\bar{\psi}(t) > \psi(t)$, but this is equivalent to the following statements:

$$\begin{aligned}
\bar{\psi}(t) > \psi(t) \quad &\Leftrightarrow \quad \frac{1}{\psi(t)} > \frac{1}{\bar{\psi}(t)} \text{ (because } \psi, \bar{\psi} < 0) \\
&\Leftrightarrow \quad \frac{1}{\psi(t)} > \frac{-t+q}{p} \\
&\Leftrightarrow \quad \text{the tangent line to } \frac{1}{\psi} \text{ which is } \frac{-t+q}{p} \text{ is below the curve } \frac{1}{\psi} \\
&\Leftrightarrow \quad \text{the curve } \frac{1}{\psi} \text{ is convex, i.e., } (\frac{1}{\psi})'' \geq 0
\end{aligned}$$

(see Fig. 4.3)

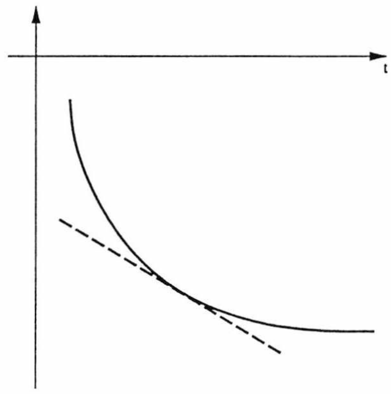

Figure 4.3: A convex curve and its tangent line

As all the statements in 4 are equivalent, in order to prove (4.17), it is enough to show that the curve $1/\psi$ is convex, which is the same as showing $(\frac{1}{\psi})'' \geq 0$. The proof of this is done in the following stages:

1. First observe that the expression for the second derivative of $1/\psi$ is given by

$$\left(\frac{1}{\psi}\right)'' = \left[-\psi'\psi^{-2}\right]' = -\psi''\psi^{-2} - \psi'(-2)\psi^{-3}\psi' = \frac{1}{\psi^3}\left[2(\psi')^2 - \psi''\psi\right] \quad (4.19)$$

2. As $\psi < 0$, we must show that $2(\psi')^2 - \psi''\psi \leq 0$ or $\psi''\psi - 2(\psi')^2 \geq 0$.

   Since the function ψ, is a sum of functions, if we prove that the condition above is true for a generic term within this function, then by the next remark and induction we will have the general result.

3. We show that if u_1,v_1 and w_1 satisfy $u_1 < 0$ and $u_1 w_1 \geq 2v_1^2$, and u_2, v_2,w_2 satisfy similar conditions, then the condition is also true for the sum of corresponding terms. In other words, $(u_1 + u_2)(w_1 + w_2) \geq 2(v_1 + v_2)^2$.

   Consider the expression $u_1 t^2 + 4v_1 t + 2w_1$; it can be written as

$$u_1 \left(t + \frac{2v_1}{u_1}\right)^2 + 2w_1 - \frac{4v_1^2}{u_1} = u_1 \left(t + \frac{2v_1}{u_1}\right)^2 + \frac{2}{u_1}(u_1 w_1 - 2v_1^2).$$

   As we are assuming $u_1 w_1 - 2v_1^2 \geq 0$ and $u_1 < 0$, the whole expression above is negative, i.e.,

$$u_1 t^2 + 4v_1 t + 2w_1 \leq 0 \text{ for all } t.$$

   In the same way, we have

$$u_2 t^2 + 4v_2 t + 2w_2 \leq 0 \text{ for all } t.$$

   But then, it is easy to see that

$$(u_1 + u_2)t^2 + 4(v_1 + v_2)t + 2(w_1 + w_2) \leq 0 \text{ for all } t. \tag{4.20}$$

   Exactly as before, (4.20) can be re-written as

$$(u_1 + u_2)\left(t + \frac{2(v_1 + v_2)}{u_1 + u_2}\right)^2 + \frac{2}{u_1 + u_2}\left[(u_1 + u_2)(w_1 + w_2) - 2(v_1 + v_2)^2\right]. \tag{4.21}$$

   Now, making $t = -2(v_1 + v_2)/(u_1 + u_2)$, (4.21) is simplified to

$$\frac{2}{u_1 + u_2}\left[(u_1 + u_2)(w_1 + w_2) - 2(v_1 + v_2)^2\right].$$

   As (4.20) is true for all t and $u_1 + u_2$ is negative, the above expression must also be negative, and then,

$$(u_1 + u_2)(w_1 + w_2) \geq 2(v_1 + v_2)^2.$$

4. In our case, we consider $u = f_j$, $v = f_j'$, and $w = f_j''$, where $f_j(t) = z_j^2/(\delta_j - t)$. We must show that the induction hypotheses are satisfied when we use these functions,

but this is trivially true, because

$$f_j' = \frac{z_j^2}{(\delta_j - t)^2}$$

$$f_j'' = \frac{2z_j^2}{(\delta_j - t)^3}$$

and so,

$$uw = f_j(t) f_j''(t) = \frac{2z_j^2}{(\delta_j - t)^3} \frac{z_j^2}{(\delta_j - t)} = \frac{2z_j^4}{(\delta_j - t)^4} = 2(f_j'(t))^2 = 2v^2$$

and f_j is negative.

As the result $uw \geq 2v^2$ was proven to be true for one generic term f_j of function ψ, the induction hypothesis shows that the result is also valid when we consider the sum of two of these terms. Now, as the result is valid for the sum of two terms and also for one third term, it will hold true for the sum of three terms and so on. In general, if we call $g = \sum_{j=1}^{N-1} z_j^2/(\delta_j - t) = \sum_{j=1}^{N-1} f_j$ and $\psi = g + f_N$, then by induction we have $gg'' \geq 2(g')^2$, and as we already know that $f_N f_N'' \geq 2(f_N')^2$, we obtain $\psi\psi'' \geq 2(\psi')^2$.

5. The above argument has proved that $\psi''\psi \geq 2(\psi')^2$, that is, $(1/\psi)'' \geq 0$ which in turn proves that $-\bar{\psi}(t) < -\psi(t)$.

To prove the second part of Result 1, although the same method will work, we use a completely different, somewhat simpler, approach. We are able to show directly that $r + s/(\delta - t) > \varphi(t)$, using the expressions given for r, s and $\varphi(t)$ and manipulating them algebraically.

We want $r + s/(\delta - t) - \varphi(t) > 0$. This is the same as having

$$\varphi(t_0)(\delta - t) - \varphi'(t_0)(\delta - t_0)(\delta - t) + \varphi'(t_0)(\delta - t_0)^2 - \varphi(t)(\delta - t) > 0$$

(see (4.7) and (4.8)). Expanding each of the terms above, using the definition of φ (in (4.3)) and the corresponding expression for the first derivative of φ at t_0, the left-hand side becomes

$$M \sum_{j=i+1}^{n} \frac{z_j^2}{(\delta_j - t_0)^2(\delta_j - t)} \tag{4.22}$$

where

$$M = (\delta - t)(\delta_j - t)(\delta_j - t_0) + (\delta - t_0)(\delta_j - t)(t - t_0) - (\delta - t)(\delta_j - t_0)^2$$

grouping the first and third products we get

$$M = (\delta_j - t_0)(\delta - t)(t_0 - t) + (\delta - t_0)(\delta_j - t)(t - t_0)$$

which after further simplification gives

$$M = (t - t_0)(\delta_j - \delta)(t - t_0) \tag{4.23}$$

and as $\delta_j > \delta$, for $j \geq i + 1$ which is the case here, we have $M > 0$. Because $t < \delta_j$ for $j \geq i + 1$ this result makes the whole expression (4.22) positive, which is what we required.

The following result guarantees the convergence of the sequence $\{t_k\}_{k=1,2,\ldots}$ where t_{k+1} is obtained from t_k in the same way as t_1 was obtained from t_0 in the previous discussion once the start of the iteration $t_0 \in (0, \mu_i)$ is given. We show later how to obtain a good initial approximation t_0.

Result 2 *The sequence t_k defined as above converges to the solution μ_i. In other words $t_k \longrightarrow \mu_i$ as $k \longrightarrow +\infty$.*

For this, recall that the sequence t_k is increasing, but each $t_k < \mu_i$. Suppose $t_k \longrightarrow t_*$. If p_k, q_k, r_k and s_k denote the values of p, q, r and s when the curves $p/(q-t)$ and $r + s/(\delta - t)$ are fitted at t_k, then

$$\frac{p_k}{q_k - t_{k+1}} + r_k + \frac{s_k}{\delta - t_{k+1}} + 1 = 0 \tag{4.24}$$

where

$$p_k = \frac{\psi_k^2}{\psi_k'}; \quad q_k = t_k + \frac{\psi_k}{\psi_k'}; \quad r_k = \varphi_k - (\delta - t_k)\varphi_k'; \quad s_k = (\delta - t_k)^2\varphi_k'$$

If we let $k \to \infty$, then $p_k \to p_*$, $q_k \to q_*$, etc., where $p_* = \psi(t_*)^2/\psi'(t_*)$ and analogously for the other expressions.

Letting $k \to \infty$ in (4.24), we find

$$\frac{p_*}{q_* - t_*} + r_* + \frac{s_*}{\delta - t_*} + 1 = 0$$

But then,

$$\frac{\frac{\psi(t_*)^2}{\psi'(t_*)}}{t_* + \frac{\psi(t_*)}{\psi'(t_*)} - t_*} + \varphi(t_*) - (\delta - t_*)\varphi'(t_*) + \frac{(\delta - t_*)^2\varphi'(t_*)}{\delta - t_*} + 1 = 0$$

which after some simplification is the same as

$$\psi(t_*) + \varphi(t_*) + 1 = 0$$

But then t_* is a root of the secular equation, and thus $t_* = \mu_i$.

Result 3 *Consider the sequence of iterates $\{t_k\}$ as before ((4.9) and (4.13)). Then the sequence $\{t_k\}$ converges quadratically to μ. In other words, there is a constant K such that $|t_{k+1} - \mu| \leq K|t_k - \mu|^2$, for all k.*

To prove this, consider the function in two variables

$$F(u,v) = \frac{p}{q - v} + r + \frac{s}{\delta - v} + 1 \tag{4.25}$$

where p, q, r and s are defined as before (see expressions (4.5), (4.6), (4.7), (4.8)), but considered as functions of u; u and v are in the interval $(0, \delta)$.

First, notice the following characteristics of F

- $F(\mu, \mu) = 0$. This comes from the definitions of p, q, r and s at μ, because $p/(q - \mu) = \psi(\mu)$, $r + s/(\delta - \mu) = \varphi(\mu)$ and μ is the root of

$$\psi(\mu) + \varphi(\mu) + 1 = 0$$

- $F_1(\mu,\mu) = 0$ (where the subscript denotes the first partial derivative with respect to u). The verification of this statement although tedious, is simply an exercise of computing the derivatives of the functions in question.

The expression for the first partial derivative of F with respect to u is given by

$$F_1 = \frac{\partial F}{\partial u} = \frac{\frac{\partial p}{\partial u}(q-v) - p\frac{\partial(q-v)}{\partial u}}{(q-v)^2} + \frac{\partial r}{\partial u} + \frac{\frac{\partial s}{\partial u}}{\delta - v}$$

The partial derivatives involved in the previous expression are given by

$$\begin{aligned}
\frac{\partial p}{\partial u} &= \frac{\psi(u)}{(\psi'(u))^2}[2(\psi'(u))^2 - \psi(u)\psi''(u)] \\
\frac{\partial(q-v)}{\partial u} &= 1 + \frac{(\psi'(u))^2 - \psi(u)\psi''(u)}{(\psi'(u))^2} \\
\frac{\partial r}{\partial u} &= 2\varphi'(u) - (\delta - u)\varphi''(u) \\
\frac{\partial s}{\partial u} &= (\delta - u)[(\delta - u)\varphi''(u) - 2\varphi'(u)]
\end{aligned}$$

Substituting these expressions where they appear in F_1, we find

$$\begin{aligned}
\frac{\partial F}{\partial u} = & \frac{\frac{\psi(u)}{\psi'(u)^2}[2(\psi'(u))^2 - \psi(u)\psi''(u)]\left((u-v) + \frac{\psi(u)}{\psi'(u)}\right)}{\left((u-v) + \frac{\psi(u)}{\psi'(u)}\right)^2} - \\
& \frac{\frac{\psi^2(u)}{\psi'(u)}\left[1 + \frac{\psi'(u)^2 - \psi(u)\psi''(u)}{\psi'(u)^2}\right]}{\left((u-v) + \frac{\psi(u)}{\psi'(u)}\right)^2} + \\
& 2\varphi'(u) - (\delta - u)\varphi''(u) + \\
& \frac{\delta - u}{\delta - v}((\delta - u)\varphi''(u) - 2\varphi'(u))
\end{aligned}$$

Evaluating this expression at the point (μ,μ) and simplifying terms we obtain

$$\frac{\partial F(\mu,\mu)}{\partial u} = \frac{1}{\psi'(\mu)}[2(\psi'(\mu))^2 - \psi(\mu)\psi''(\mu)] - \frac{\psi'(\mu)}{(\psi'(\mu))^2}[2(\psi'(\mu))^2 - \psi(\mu)\psi''(\mu)] = 0$$

which is the result we require.

- $F_2(\mu, \mu) = \psi'(\mu) + \varphi'(\mu) > 0$. This again comes from the definitions of p, q, r and s. The fact that $\psi'(\mu) + \varphi'(\mu) \neq 0$ can be seen by observing that if $\psi'(\mu) + \varphi'(\mu) = 0$ then $\psi'(\mu) = \varphi'(\mu) = 0$ (since $\psi', \varphi' \geq 0$). But this implies that both ψ and φ have a horizontal tangent at μ which we know from their graphs cannot happen.

With these observations, we write the function F using Taylor's theorem in two variables centered at the point (μ, μ).

$$\begin{aligned} F(u,v) &= F(\mu,\mu) + (u-\mu)F_1(\mu,\mu) + (v-\mu)F_2(\mu,\mu) + \\ &\quad \frac{1}{2}\Big[(u-\mu)^2 F_{11}\left(\mu+\theta(u-\mu), \mu+\theta(v-\mu)\right) + \\ &\quad 2(u-\mu)(v-\mu)F_{12}\left(\mu+\theta(u-\mu), \mu+\theta(v-\mu)\right) + \\ &\quad (v-\mu)^2 F_{22}\left(\mu+\theta(u-\mu), \mu+\theta(v-\mu)\right)\Big] \end{aligned}$$

where $0 < \theta < 1$.

Now, we have also that $F(t_n, t_{n+1}) = 0$ (because we find t_{n+1} by solving $-p/(q - t_{n+1}) = 1 + r + s/(\delta - t_{n+1})$, where p, q, r and s are defined at t_n). Then, applying the expansion above to the point $(u, v) = (t_n, t_{n+1})$ we get
(recall that $F(\mu, \mu) = F_1(\mu, \mu) = 0$)

$$0 = (t_{n+1} - \mu)F_2(\mu,\mu) + \frac{1}{2}[(t_n-\mu)^2 F_{11} + 2(t_n-\mu)(t_{n+1}-\mu)F_{12} + (t_{n+1}-\mu)^2 F_{22}]$$

where F_{11}, F_{12} and F_{22} are evaluated at $(\mu + \theta(u - \mu), \mu + \theta(v - \mu))$. Rearranging the terms

$$\frac{\mu - t_{n+1}}{(\mu - t_n)^2} F_2(\mu,\mu) = \frac{1}{2}[F_{11} + 2\frac{t_{n+1}-\mu}{t_n - \mu}F_{12} + \frac{(t_{n+1}-\mu)^2}{(t_n-\mu)^2}F_{22}]$$

Using the fact that any continuous function is bounded in a compact set, and observing that $(t_{n+1} - \mu)/(t_n - \mu) \leq 1$, there is a constant K_1 so that

$$\frac{\mu - t_{n+1}}{(\mu - t_n)^2} F_2(\mu,\mu) \leq K_1$$

or a constant K so that

$$\mu - t_{n+1} \leq K(\mu - t_n)^2.$$

4.3 The initial approximation t_0

We have described an iterative method to find the solution of the equation $-\psi(t) = 1+\varphi(t)$. We assumed the existence of a starting point t_0 for the iteration which lies in the range $(0, \mu)$. Now we will show how we can choose this value, and the specification of the method is completed.

We know that at the root μ, $-\psi(\mu) = 1 + \varphi(\mu)$. Working with these functions ψ and φ, we can find a t_0 which is to the left of the root being sought. First we write the functions ψ and φ at μ as

$$\begin{aligned} \psi(\mu) &= -\frac{z_i^2}{\mu} + \sum_{j=1}^{i-1} \frac{z_j^2}{\delta_j - \mu} \\ \varphi(\mu) &= \frac{z_{i+1}^2}{\delta_{i+1} - \mu} + \sum_{j=i+2}^{n} \frac{z_j^2}{\delta_j - \mu} \end{aligned}$$

We then have the following inequalities

$$\begin{aligned} -\psi(\mu) &\geq \frac{z_i^2}{\mu} - \sum_{j=1}^{i-1} \frac{z_j^2}{\delta_j - \delta_{i+1}} \\ \varphi(\mu) &\leq \frac{z_{i+1}^2}{\delta_{i+1} - \mu} + \sum_{j=i+2}^{n} \frac{z_j^2}{\delta_j - \delta_{i+1}} \end{aligned}$$

(in the inequalities above we used the facts that for $j < i$, $\delta_j < \delta_i = 0 < \mu < \delta_{i+1}$ and for $j > i+1$, $\delta_i < \mu < \delta_{i+1} < \delta_j$) which combined with $-\psi(\mu) = 1 + \varphi(\mu)$ gives

$$\frac{z_i^2}{\mu} - \sum_{j=1}^{i-1} \frac{z_j^2}{\delta_j - \delta_{i+1}} \leq 1 + \frac{z_{i+1}^2}{\delta_{i+1} - \mu} + \sum_{j=i+2}^{n} \frac{z_j^2}{\delta_j - \delta_{i+1}}$$

Rearranging the terms we get

$$\frac{z_i^2}{\mu} \leq \frac{z_{i+1}^2}{\delta_{i+1} - \mu} + K \tag{4.26}$$

where

$$K = 1 + \sum_{\substack{j=1 \\ j \neq i, i+1}}^{n} \frac{z_j^2}{\delta_j - \delta_{i+1}}.$$

Looking at the graph of curves z_i^2/t and $(z_{i+1}^2/(\delta_{i+1}-t))+K$ (Figure 4.4) we observe that the intersection point of these two curves is to the left of μ (observe inequality (4.26)) and so we can choose t_0 to be the solution of

$$\frac{z_i^2}{t_0} = \frac{z_{i+1}^2}{\delta_{i+1}-t_0} + K \tag{4.27}$$

Figure 4.4 shows just the one possibility where $K > 0$. In fact, there are two possibilities

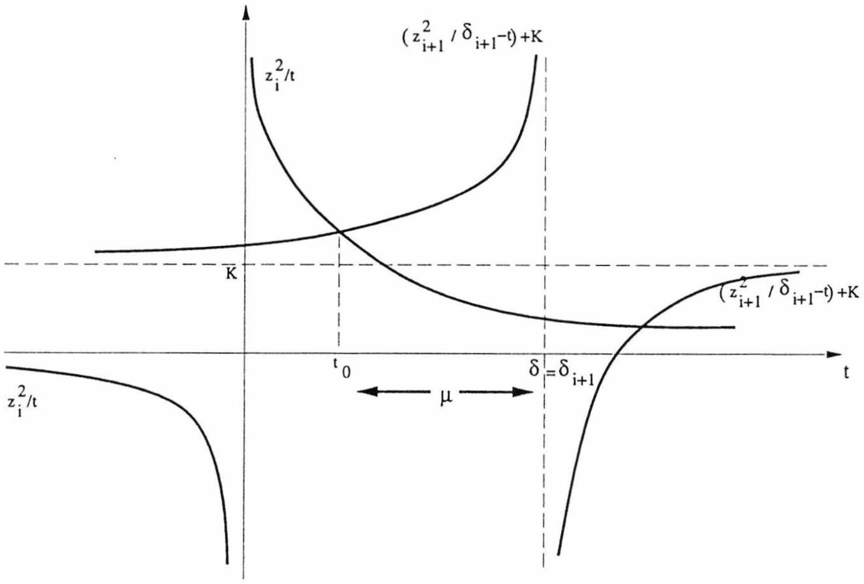
$(z_{i+1}^2 / \delta_{i+1}-t)+K$
z_i^2/t
K
$(z_{i+1}^2 / \delta_{i+1}-t)+K$
t_0
$\delta_i = \delta_{i+1}$
t
μ
z_i^2/t

Figure 4.4: Choosing t_0.

for K; $K > 0$ or $K \leq 0$. The graphs in Figure 4.5 illustrate these two cases and how the zeros of (4.27) behave.

As can be observed, we have to take the smaller root, when $K > 0$ and the bigger one in the other case. Equation (4.27) can be rewritten as a quadratic equation in the same way as (4.9) was transformed to find t_{k+1} from t_k. Here (4.27) takes the form $t^2 - at + b$, where $a = \delta_{i+1} + (z_i^2 + z_{i+1}^2)/K$ and $b = z_i^2\delta_{i+1}/K$. Thus the root to be used is

1. $K > 0$

$$t_0 = \frac{2b}{a + \sqrt{a^2 - 4b}}$$

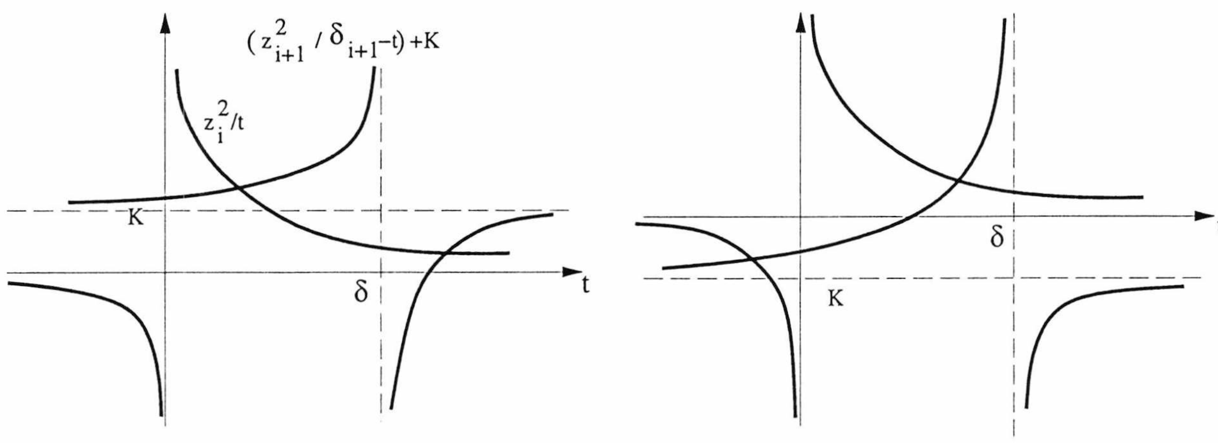

Figure 4.5: (a) $K > 0$, (b) $K < 0$.

2. $K < 0$

   - $a \geq 0$

$$t_0 = \frac{a + \sqrt{a^2 - 4b}}{2}$$

   - $a < 0$

$$t_0 = \frac{2b}{a - \sqrt{a^2 - 4b}}$$

4.4 Theory is different from practice

When we explained how the eigenvalues of matrix $D + \rho zz'$ are found, we proved that an eigenvalue λ could not be equal to any entry of D. In practice, however, as we work with finite precision arithmetic, this may fail. When a situation like this arises, we will encounter trouble in computing the corresponding eigenvector, because the formula for finding it (see (3.14)) involves division by the quantities $d_i - \lambda_j$, which can be zero in two cases.

As explained previously, the secular equation is transformed in such a way that we do not compute λ directly, but compute roots μ_j which then yield $\lambda_j = d_j + \rho\mu_j$. This transformation was suggested by Bunch *et al.* [13] in order to achieve more accuracy in the solutions. The suggested modification really improves the results, as Chapter 6 confirms, but it has also another important role, namely to permit the computation of the corresponding eigenvector when the numerical values of λ_i and d_i coincide.

This case, however, is not the only one where this difficulty can arise. Just as there is a possibility of λ_i coinciding with the left end of the interval in which it is located, so there is also the possibility of it coinciding with the other end of the interval, i.e, it can also happen that $\lambda_i = d_{i+1}$. Although the first case is cited by many of the authors already mentioned, the second case seems to have been ignored by most of them. The only reference we have found for this second case, after we encountered the difficulty ourselves, is in Gill and Tadmor [29]. Based on some ideas suggested by them, we developed an algorithm to overcome the difficulty which occurs when the eigenvalue is equal to the right extreme of the interval. Our first attempt to cope with this, was to re-compute the eigenvalue when we found $\lambda_i = d_{i+1}$, and this was satisfactory in terms of getting the program to work in such cases. In general terms, what the new algorithm does is to reformulate the computation so that we work with $\nu = (\lambda_i - d_{i+1})/\rho$ where we previously used $\mu = (\lambda_i - d_i)/\rho$. The quantity ν will be very small, and in practice the eigenvalue computed is still the same as d_{i+1}, but by making this change we are able to compute the eigenvector where we could not previously. The working of this new algorithm and the reason for computing μ instead of λ, is explained in the next sections.

4.5 Overcoming division by zero

Recall that μ_i is related to λ through $\lambda = d_i + \rho\mu_i$, and the corresponding eigenvector is given by

$$q = \frac{(D - \lambda I)^{-1} z}{\| (D - \lambda I)^{-1} z \|}. \tag{4.28}$$

From this formula, it can clearly be seen that, if effectively $\lambda = d_j$ for some j, there will be a problem with $(D - \lambda I)^{-1}$. If instead of using $D - \lambda I$, we use $\mathcal{D} - \mu I$, where $\mathcal{D}$ is a diagonal matrix whose elements are $\delta_1, \delta_2, \ldots, \delta_n$ and where μ is defined as before, then

$\mathcal{D} - \mu I$ is given by

$$\begin{bmatrix} \delta_1 - \mu & & & & \\ & \ddots & & & \\ & & \delta_i - \mu & & \\ & & & \ddots & \\ & & & & \delta_n - \mu \end{bmatrix} = \begin{bmatrix} \frac{d_1 - d_i - \rho\mu}{\rho} & & & & \\ & \ddots & & & \\ & & -\mu & & \\ & & & \ddots & \\ & & & & \frac{d_n - d_i - \rho\mu}{\rho} \end{bmatrix} \tag{4.29}$$

where in the expression above, we used the fact that $\delta_i = (d_i - d_i)/\rho = 0$ and $\mu = \mu_i$. The eigenvector $q = q_i$ is now given by

$$q = \frac{(\mathcal{D} - \mu I)^{-1} z}{\| (\mathcal{D} - \mu I)^{-1} z \|}. \tag{4.30}$$

Note that with this new formulation, the i^{th} diagonal entry of $\mathcal{D} - \mu I$ is no longer zero and there is no problem in computing q_i even if λ_i and d_i have identical numerical values (and likewise for the case $\lambda_i = d_{i+1}$ a similar reformulation is used – see the next section).

What remains to be answered is: with this formulation, are the results we get the same as before? The answer is affirmative and can easily be verified from the following expressions for

$$\mathcal{D} - \mu I = \begin{bmatrix} \frac{d_1 - \lambda}{\rho} & & & & \\ & \ddots & & & \\ & & \frac{d_i - \lambda}{\rho} & & \\ & & & \ddots & \\ & & & & \frac{d_n - \lambda}{\rho} \end{bmatrix} \tag{4.31}$$

$$y = \begin{bmatrix} \frac{z_1 \rho}{d_1 - \lambda} \\ \frac{z_2 \rho}{d_2 - \lambda} \\ \vdots \\ \frac{z_i \rho}{d_i - \lambda} \\ \vdots \\ \frac{z_n \rho}{d_n - \lambda} \end{bmatrix} = \rho \begin{bmatrix} \frac{z_1}{d_1 - \lambda} \\ \frac{z_2}{d_2 - \lambda} \\ \vdots \\ \frac{z_i}{d_i - \lambda} \\ \vdots \\ \frac{z_n}{d_n - \lambda} \end{bmatrix} \tag{4.32}$$

where, $q = y/||y||$. It can be seen, from these expressions that apart from multiplicative constants, the formula for the eigenvectors is the same as that first given in (4.28). This means that we can use (4.30) for computing the eigenvectors, as it is the same as (4.28). The advantage is that the i^{th} component of the i^{th} eigenvector no longer poses any numerical problems, since it is given by

$$\frac{\rho z_i}{d_i - \lambda} = \frac{z_i}{-\mu}$$

and although μ can be very small, it is never zero.

4.6 And if $\lambda = d_{i+1}$?

The case when the computed numerical value of λ and the numerical value of d_i coincide, is solved by the formulation shown in the previous section, but if the computed value of λ happens to be numerically equal to d_{i+1} (or $\mu_i = \delta_{i+1}$), then the approach taken above does not help, because we will still have to confront a division by zero in the $(i+1)^{st}$ position of the i^{th} eigenvector. So, what should be done in this case? The natural way of proceeding is to try a method similar to that used before, but now avoiding division by zero in the $(i+1)^{st}$ position of the corresponding eigenvector. We cannot use the μ computed as previously, because it originates at d_i and we will not be able to cancel any term in the denominator of the fraction where the division by zero occurs. The idea is, then, to compute a root that starts from d_{i+1} and not d_i.

We seek now, a root ν such that $\lambda_i = d_{i+1} + \rho\nu$, where we consider a negative ν. We use the notation ν, to distinguish the way we define it from the way μ is specified, but in the end they both generate roots of the secular equation. We now re-define the δ_j's as well, setting $\delta_j = (d_j - d_{i+1})/\rho$. Thus, the $(i+1)^{st}$ position of the i^{th} eigenvector (where the problem occurred before), is now given by

$$\frac{z_{i+1}}{\delta_{i+1} - \nu} = \frac{z_{i+1}}{-\nu}$$

which represents no difficulty for the calculations. The i^{th} entry can also be computed, because $\delta_i - \nu = (d_i - d_{i+1} - \rho\nu)/\rho = (d_i - (d_{i+1} + \rho\nu))/\rho$ and as in practice, $d_{i+1} + \rho\nu$ is identical to d_{i+1} and $d_i \neq d_{i+1}$ (assured by deflation), we run no risk of division by zero.

4.6.1 The method to find ν

Many of the details which we described relating to the computation of μ arise in a similar way here, but there are some vital differences, and because of these, we need to develop the new method in a little detail.

We compute a *negative* real number ν such that

$$\lambda_i = d_{i+1} + \rho\nu$$

and

$$d_i < d_{i+1} + \rho\nu < d_{i+1}$$

or

$$\frac{d_i - d_{i+1}}{\rho} < \nu < 0.$$

Calling the quantity

$$\frac{d_i - d_{i+1}}{\rho} = \tilde{\delta}_i$$

and defining in general

$$\tilde{\delta}_j = \frac{d_j - d_{i+1}}{\rho} \tag{4.33}$$

we have

$$\tilde{\delta}_i < \nu < 0 = \tilde{\delta}_{i+1}$$

and

$$\tilde{\delta}_j - \nu < \tilde{\delta}_j - \tilde{\delta}_i < 0\,,\ \text{for } j < i. \tag{4.34}$$

From expression (4.34) we obtain

$$\frac{1}{\tilde{\delta}_j - \tilde{\delta}_i} < \frac{1}{\tilde{\delta}_j - \nu}. \tag{4.35}$$

For $j > i$, the inequality relation between $\tilde{\delta}_j - \nu$ and $\tilde{\delta}_j - \tilde{\delta}_i$ persists, but this time both quantities are positive, which again makes inequality (4.35) true, but the form below will be of more use in the following discussion:

$$-\frac{1}{\tilde{\delta}_j - \nu} < -\frac{1}{\tilde{\delta}_j - \tilde{\delta}_i}. \tag{4.36}$$

4.6.1.1 Finding the initial approximation

We define $\tilde{\psi}$ and $\tilde{\varphi}$ in the same way as ψ and φ were defined, but using $\tilde{\delta}$ in place of δ

$$\tilde{\psi}(t) \equiv \sum_{j=1}^{i} \frac{z_j^2}{\tilde{\delta}_j - t} \tag{4.37}$$

$$\tilde{\varphi}(t) \equiv \sum_{j=i+1}^{n} \frac{z_j^2}{\tilde{\delta}_j - t} \tag{4.38}$$

Using inequalities (4.35) and (4.36) we get

$$\tilde{\psi}(\nu) = \sum_{j=1}^{i} \frac{z_j^2}{\tilde{\delta}_j - \nu} = \frac{z_i^2}{\tilde{\delta}_i - \nu} + \sum_{j=1}^{i-1} \frac{z_j^2}{\tilde{\delta}_j - \nu} > \frac{z_i^2}{\tilde{\delta}_i - \nu} + \sum_{j=1}^{i-1} \frac{z_j^2}{\tilde{\delta}_j - \tilde{\delta}_i} \tag{4.39}$$

$$\tilde{\varphi}(\nu) = \sum_{j=i+1}^{n} \frac{z_j^2}{\tilde{\delta}_j - \nu} = \frac{z_{i+1}^2}{-\nu} + \sum_{j=i+2}^{n} \frac{z_j^2}{\tilde{\delta}_j - \nu} \tag{4.40}$$

or

$$-\tilde{\varphi}(\nu) = \frac{z_{i+1}^2}{\nu} - \sum_{j=i+2}^{n} \frac{z_j^2}{\tilde{\delta}_j - \nu} < \frac{z_{i+1}^2}{\nu} - \sum_{j=i+2}^{n} \frac{z_j^2}{\tilde{\delta}_j - \tilde{\delta}_i} \tag{4.41}$$

Combining (4.39) and (4.41) and recalling that $\tilde{\psi}(\nu) + \tilde{\varphi}(\nu) = -1$ (i.e., $1 + \tilde{\psi}(\nu) = -\tilde{\varphi}(\nu)$), we get

$$1 + \frac{z_i^2}{\tilde{\delta}_i - \nu} + \sum_{j=1}^{i-1} \frac{z_j^2}{\tilde{\delta}_j - \tilde{\delta}_i} < 1 + \tilde{\psi}(\nu) = -\tilde{\varphi}(\nu) < \frac{z_{i+1}^2}{\nu} - \sum_{j=i+2}^{n} \frac{z_j^2}{\tilde{\delta}_j - \tilde{\delta}_i} \tag{4.42}$$

or

$$1 + \frac{z_i^2}{\tilde{\delta}_i - \nu} + \sum_{\substack{j=1 \\ j \neq i, i+1}}^{n} \frac{z_j^2}{\tilde{\delta}_j - \tilde{\delta}_i} < \frac{z_{i+1}^2}{\nu}. \tag{4.43}$$

Setting

$$\tilde{K} = 1 + \sum_{\substack{j=1 \\ j \neq i, i+1}}^{n} \frac{z_j^2}{\tilde{\delta}_j - \tilde{\delta}_i} \tag{4.44}$$

we can write (4.43) as

$$\frac{z_i^2}{\tilde{\delta}_i - \nu} + \tilde{K} \leq \frac{z_{i+1}^2}{\nu}. \tag{4.45}$$

Consider the functions

$$\frac{z_i^2}{\tilde{\delta}_i - t} + \tilde{K} \text{ and } \frac{z_{i+1}^2}{t}.$$

The two possible graphs for these curves, when $\tilde{K} < 0$ and $\tilde{K} > 0$ are shown in Figures 4.6 and 4.7. When $\tilde{K} = 0$, the graph assumes a similar form, but it is to be noted that there will just be one intersection point in the interval $(\tilde{\delta}_i, 0)$.

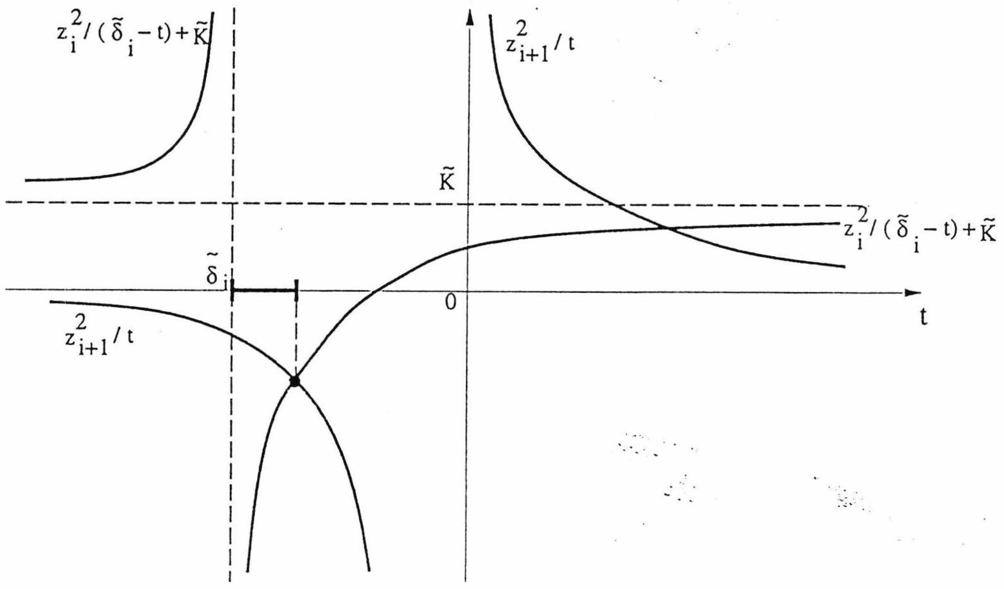

Figure 4.6: The possible locations of ν, when $\tilde{K} > 0$

As $\nu < 0$, we should consider intervals only in the negative axis. In the case of the second of these two figures (Fig. 4.7), we should take the interval emphasized in the figure, because $\tilde{\delta}_i < \nu < 0$. If we show that the sequence t_k of iterations is a decreasing one, we can choose the intersection points in the appropriate intervals, to be the starting iteration

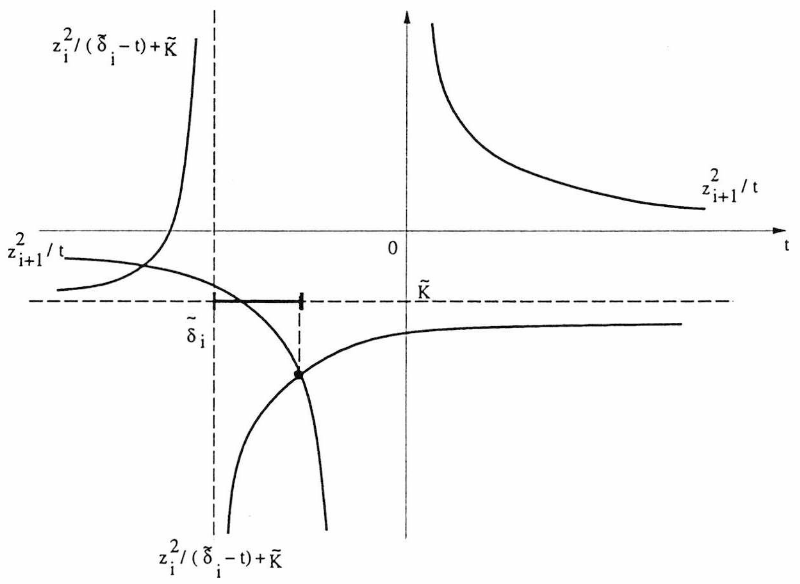
$z_i^2/(\tilde\delta_i - t)+\tilde K$
z_{i+1}^2/t
z_{i+1}^2/t
0
t
$\tilde K$
$\tilde\delta_i$
$z_i^2/(\tilde\delta_i - t)+\tilde K$

Figure 4.7: The possible locations of ν, when $\tilde{K} < 0$

points. Thus, if $\tilde{K} > 0$ we take the smaller root (the negative one) of equation

$$\frac{z_i^2}{\tilde{\delta}_i - t} + \tilde{K} = \frac{z_{i+1}^2}{t} \tag{4.46}$$

and if $\tilde{K} < 0$, we take the greater one. If $\tilde{K} = 0$, there is just one root given by

$$t = \frac{\tilde{\delta}_i z_{i+1}^2}{z_i^2 + z_{i+1}^2}.$$

As we have seen before, when $\tilde{K} \neq 0$, equation (4.46) leads to a quadratic equation of form $t^2 - at + b = 0$, where in this case

$$a = \tilde{\delta}_i + \frac{z_i^2 + z_{i+1}^2}{\tilde{K}} \text{ and } b = \frac{z_{i+1}^2 \tilde{\delta}_i}{\tilde{K}}$$

If $\tilde{K} > 0$, we take the smaller root, which is given in general by $t = (a - \sqrt{a^2 - 4b})/2$, and to avoid cancellation, we choose t as

$$t = \frac{a - \sqrt{a^2 - 4b}}{2}\text{, if } a < 0 \text{ or } t = \frac{2b}{a + \sqrt{a^2 - 4b}}\text{, if } a \geq 0$$

Note that when $a = 0$, $b < 0$, so that the expression above still gives a real value and thus, the intersection point.

When $\tilde{K} < 0$, we must take the greater root which is given by $t = (a + \sqrt{a^2 - 4b})/2$. In this case, as $\tilde{K} < 0$, $a < 0$ and so for numerical reasons we use the equivalent formula

$$t = \frac{2b}{a - \sqrt{a^2 - 4b}}.$$

The problem of finding the initial estimate for ν is solved as long as we show that the sequence t_k generated from it is a decreasing one. Using the same method for finding the subsequent iterations as that used to compute μ, we can see, from the graph shown in Figure 4.8, that the sequence is decreasing and approachs the solution of $1 + \tilde{\psi} = -\tilde{\varphi}$. The next section explores this point in detail.

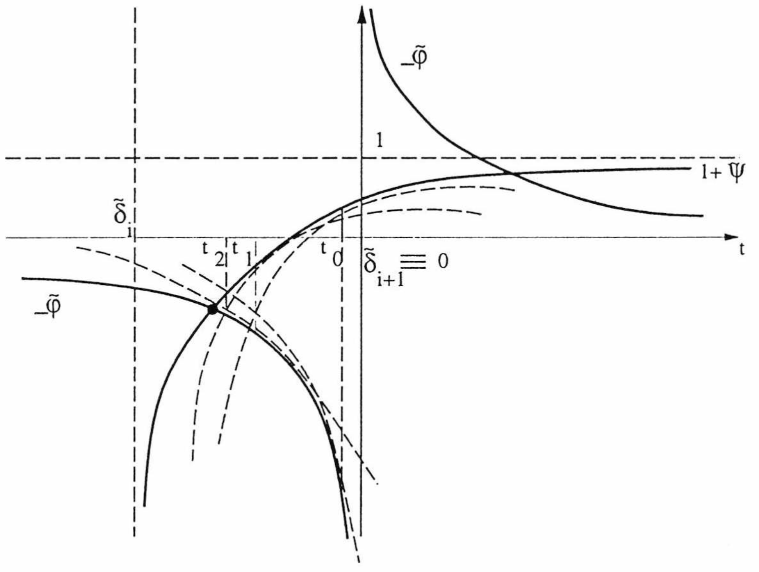

Figure 4.8: The sequence $t_k \longrightarrow \nu$

4.6.1.2 How to find subsequent iterations

The method for finding the next iteration point from a starting one, is entirely analogous to the one shown before in Section 4.1. What we want here is to find the value of t for which the equation $1 + \tilde{\psi}(t) = -\tilde{\varphi}(t)$ is true. As $\tilde{\psi}$ has an asymptote at $t = \tilde{\delta}_i$, we approximate $\tilde{\psi}$ by a function with the form $r + s/(\tilde{\delta}_i - t)$ and $\tilde{\varphi}(t)$ by $p/(q - t)$. Figure 4.8 displays these curves, their relation to $\tilde{\varphi}$ and $\tilde{\psi}$, and how the next iteration point is found. Exactly as with the method in Section 4.1, the approximating functions are made to coincide with functions $\tilde{\psi}$ and $\tilde{\varphi}$ and their derivatives at the point t_0 where the iteration starts:

$$\tilde{\psi}(t_0) = r + \frac{s}{\tilde{\delta} - t_0} \text{ and } \tilde{\psi}'(t_0) = \frac{s}{(\tilde{\delta} - t_0)^2} \tag{4.47}$$

$$\tilde{\varphi}(t_0) = \frac{p}{q - t_0} \text{ and } \tilde{\varphi}'(t_0) = \frac{p}{(q - t_0)^2} \tag{4.48}$$

This means that p, q, r and s are given this time by

$$p = \frac{[\tilde{\varphi}(t_0)]^2}{\tilde{\varphi}'(t_0)}, q = t_0 + \frac{\tilde{\varphi}(t_0)}{\tilde{\varphi}'(t_0)} \tag{4.49}$$

$$r = \tilde{\psi}(t_0) - \tilde{\psi}'(t_0)(\tilde{\delta}_i - t_0), s = \tilde{\psi}'(t_0)(\tilde{\delta}_i - t_0)^2 \tag{4.50}$$

The next iterate is found by computing the solution of

$$1 + r + \frac{s}{\tilde{\delta}_i - t} = -\frac{p}{q - t} \tag{4.51}$$

We deal with this equation in the same manner as previously, but we note that the variables p, q, r and s have a different meaning here. After some algebraic manipulations, we arrive at an equation $u^2 - au + b = 0$, where

$$a = \frac{1}{1+r}\left[\Delta(1 + \tilde{\psi}_0) + \frac{\tilde{\varphi}_0^2}{\tilde{\varphi}_0'}\right] + \frac{\tilde{\varphi}_0}{\tilde{\varphi}_0'} \text{ and } b = \frac{1}{1+r}\Delta\frac{\tilde{\varphi}_0}{\tilde{\varphi}_0'}(1 + \tilde{\psi}_0 + \tilde{\varphi}_0) \tag{4.52}$$

In the above expressions $\Delta = \tilde{\delta}_i - t_0$ and r is given by (4.50).

In contrast to what we had before, we cannot conclude from expression (4.50) that $1 + r$ is always positive or negative, so that two situations can arise. If we draw the graphs for the functions involved in (4.51), as we have done before, we can see that when $1 + r \geq 0$ we must take the smaller root, which is the negative one, but when $1 + r$ is negative, we must take the larger one, which is greater than $\tilde{\delta}_i$. In this last case, however, we need to show that the root is negative.

When $1 + r < 0$, we are considering the root given by $u = (a + \sqrt{a^2 - 4b})/2$ and clearly we need to show $a < 0$, otherwise u would be positive and consequently $t_1 > t_0$.

First note that $1 + r < 0$ implies that $1 + \tilde{\psi}_0$ is also negative,

$$0 > 1 + r = 1 + \tilde{\psi}_0 - \Delta\tilde{\psi}_0' > 1 + \tilde{\psi}_0$$

because $\Delta < 0$, $\tilde{\psi}_0' > 0$ and thus $-\Delta\tilde{\psi}_0' > 0$. This makes $1/(1 + \tilde{\psi}_0) > 1/(1 + r)$. Another observation is that $1+\tilde{\psi}_0+\tilde{\varphi}_0 > 0$. To see this, recall that at the root ν, $1+\tilde{\psi}(\nu)+\tilde{\varphi}(\nu) = 0$, so we have

$$1+\tilde{\psi}_0+\tilde{\varphi}_0 = -\sum_{j=1}^{n} \frac{z_j^2}{\tilde{\delta}_j - \nu} + \sum_{j=1}^{n} \frac{z_j^2}{\tilde{\delta}_j - t_0} = \sum_{j=1}^{n} \frac{z_j^2(\tilde{\delta}_j - \nu - \tilde{\delta}_j + t_0)}{(\tilde{\delta}_j - \nu)(\tilde{\delta}_j - t_0)} = \sum_{j=1}^{n} \frac{z_j^2(t_0 - \nu)}{(\tilde{\delta}_j - \nu)(\tilde{\delta}_j - t_0)}$$

which is positive, because $t_0 - \nu > 0$ and $(\tilde{\delta}_j - \nu)(\tilde{\delta}_j - t_0) > 0$. Now we return to the expression for a given by (4.52).

$$\begin{aligned} a &= \frac{\Delta(1 + \tilde{\psi}_0)}{1 + r} + \frac{\tilde{\varphi}_0^2}{\tilde{\varphi}_0'(1 + r)} + \frac{\tilde{\varphi}_0}{\tilde{\varphi}_0'} \\ &< \frac{\Delta(1 + \tilde{\psi}_0)}{1 + \tilde{\psi}_0} + \frac{\tilde{\varphi}_0^2}{\tilde{\varphi}_0'(1 + \tilde{\psi}_0)} + \frac{\tilde{\varphi}_0}{\tilde{\varphi}_0'} \\ &= \Delta + \frac{\tilde{\varphi}_0}{\tilde{\varphi}_0'} \left[\frac{\tilde{\varphi}_0}{1 + \tilde{\psi}_0} + 1 \right] \\ &= \Delta + \frac{\tilde{\varphi}_0}{\tilde{\varphi}_0'} \left[\frac{1 + \tilde{\psi}_0 + \tilde{\varphi}_0}{1 + \tilde{\psi}_0} \right] < 0 \end{aligned}$$

since $\Delta < 0$, and the other term is also negative, because $\tilde{\varphi}_0$ and $\tilde{\varphi}_0'$ are both positive. We conclude that if $1 + r$ is negative, then a must be also negative.

In the other cases there is no problem, so that the solutions of (4.51) are then given by

$$t_1 = t_0 + 2b/(a + \sqrt{a^2 - 4b}) \text{ when } 1 + r \geq 0 \text{ and } a \geq 0$$

$$t_1 = t_0 + (a - \sqrt{a^2 - 4b})/2 \text{ when } 1 + r \geq 0 \text{ and } a < 0$$

and

$$t_1 = t_0 + 2b/(a - \sqrt{a^2 - 4b}) \text{ when } 1 + r < 0$$

We see that in all cases, $t_1 < t_0$, because the quantity that is added to t_0 is negative.

Again, as before, we must show that the approximating functions satisfy the situation depicted in Figure 4.8, i.e., we must show that

$$\frac{-p}{q - t} > -\tilde{\varphi} \text{ and } 1 + r + \frac{s}{\tilde{\delta}_i - t} < 1 + \tilde{\psi} \tag{4.53}$$

4.6.1.3 $-\bar{\varphi} > -\tilde{\varphi}$ and other results

The proofs for the method to find ν are similar to those given for the corresponding method to find μ. The differences that occur between these two sets of proofs are very few and are related to the way the functions involved are defined for the second method. Thus we simply need to review how these alternative definitions affect the various steps in the former proofs, noting, for example, how changes of sign in the functions involved alter the details.

Result 1 Proving the first inequality of (4.53) is equivalent to showing that $\bar{\varphi} = p/(q-t) < \tilde{\varphi}$. The observations that were valid for the function ψ in (4.17) are slightly different here and we will summarise them just to remind the reader of all the points.

- $\tilde{\varphi}$ is a positive, increasing function.
- $1/\bar{\varphi}$ and $1/\tilde{\varphi}$ have the same tangent at $t = t_0$ and as $1/\bar{\varphi}$ is a line, the common tangent is given by $y = (-t+q)/p$.
- $\tilde{\varphi} > \bar{\varphi} \Leftrightarrow 1/\bar{\varphi} > 1/\tilde{\varphi} \Leftrightarrow 1/\tilde{\varphi}$ is below its tangent $\Leftrightarrow (1/\tilde{\varphi})'' \leq 0$.
- We must then prove $(1/\tilde{\varphi})'' \leq 0$ and as $(1/\tilde{\varphi})''$ is given by an expression similar to (4.19), just replacing ψ with $\tilde{\varphi}$ and its derivatives, we must show that $\tilde{\varphi}''\tilde{\varphi} - 2(\tilde{\varphi}')^2 \geq 0$ (because $\tilde{\varphi} > 0$), but this is done exactly as before.

The proof of the second inequality of (4.53) follows the same line as the one used to prove (4.18). We must just take care here, that $r + s/(\tilde{\delta} - t) < \tilde{\psi}$ is "transformed" into $r(\tilde{\delta} - t) + s - \tilde{\psi}(\tilde{\delta} - t) > 0$, because $\tilde{\delta} - t = \tilde{\delta}_i - t < 0$. With this in mind, we have again to show that (4.22) is positive. All the expressions for M given before are valid if we replace δ with $\tilde{\delta}_i$ and the δ_j's with $\tilde{\delta}_j$ with $j \leq i$. M is now given by

$$M = (t - t_0)^2(\tilde{\delta}_j - \tilde{\delta}_i)$$

For the present situation, however, $\tilde{\delta}_j < \tilde{\delta}_i$, which makes $M < 0$. As $\tilde{\delta}_j - t < 0$, this makes (4.22) positive and thus $r + s/(\tilde{\delta} - t) < \tilde{\psi}$.

Result 2 Here, we must show that the sequence t_k generated in the way described in Section 4.6.1.2 really converges to ν. Putting it more formally: if $t_0 \in (\nu, 0)$ is given, then $t_k \to \nu$ as $k \to +\infty$.

In the same way as before, we just need to observe that as the sequence t_k is decreasing, and is bounded below by ν, then it is convergent. It can be shown that it converges to the root ν.

Result 3 The proof of Result 3 for the second method is entirely analogous to that shown before (Result3, Section 4.2) and need not be repeated here. The differences are, as in the other two proofs above, the definitions for the variables p, q, r and s. With this in mind, all the observations about the function $F(u, v)$ still remain valid, along with the final result.

4.7 The computation of the last root

It may have been noticed that when we set out the method to find the roots of the secular equation, the last root, or the case $i = n$, was excluded. This was done for reasons of clarity in the description of the method, but it did not need to be so. In our implementation, we decided to compute the last root by the same means as all the others. With this approach we probably do more work than if we use the simplified version, but we have the advantage of computing all the roots in a common way. In what follows, we check that if we treat the final root in the same way as the others, we nevertheless obtain the same result for the next iteration as we would employing the simplified version.

As the last root λ_n is bounded by the quantity $d_n + \rho$, we set $d_{n+1} = d_n + \rho$ and compute the eigenvalue that is in the interval (d_n, d_{n+1}) or compute δ_{n+1} from this value for d_{n+1} and compute the root μ_n that is in the interval (δ_i, δ_{i+1}) (= (0,1)).

We must note that for $i = n$, $f(t) = 1 + \psi(t)$ and $\varphi(t) = \varphi'(t) \equiv 0$. This means that ψ is the only function to consider here, and we only need to consider its approximation $p/(q-t)$, so that we seek the solution of $1 + p/(q-t) = 0$, which is given by

$$t = p + q = t_0 + \frac{\psi_0}{\psi_0'}[1 + \psi_0].$$

This is the solution given by Bunch et al. [13], but we will see that working with the general formula we get the same result.

Recall that the next iteration t_1 is given by

$$t_1 = t_0 + \frac{2b}{a + \sqrt{a^2 - 4b}}$$

where a and b are given by equations (4.11) and (4.12), and $\varphi_0 = \varphi_0' = 0$. The values of a and b here simplify to

$$a = \Delta + \frac{\psi_0}{\psi_0'}(1 + \psi_0)$$

$$b = \Delta\frac{\psi_0}{\psi_0'}(1 + \psi_0)$$

so that

$$a^2 - 4b = \left[\Delta - \frac{\psi_0}{\psi_0'}(1 + \psi_0)\right]^2 \text{ and } a + \sqrt{a^2 - 4b} = 2\Delta$$

dividing $2b$ by the second value above and adding the result to t_0, we thus obtain the same result as the simplified version.

However, using our method to find ν, if we want to compute the last root, we cannot compute it in the same way as the other roots. This is because for the second method, p is given by $\tilde{\varphi}_0/\tilde{\varphi}_0'$ and this quantity is not defined, so that a and b cannot be computed above. The solution is to compute the root from the simplification

$$r + \frac{s}{\tilde{\delta} - t} = -1$$

which gives

$$t = \tilde{\delta} + \frac{\tilde{\Delta}^2\tilde{\psi}_0'}{1 + r}$$

4.8 Alternative proof of Result 1

We need to prove that $p/(q-t) > \psi(t)$. For this, consider the polynomial

$$\begin{aligned} h(t) &= \left(\frac{p}{q-t} - \psi(t)\right)(q-t)\prod_{j=1}^{i}(\delta_j - t) \\ &= p\prod_{j=1}^{i}(\delta_j - t) - (q-t)\sum_{k=1}^{i} z_k^2 \prod_{\substack{j=1 \\ j\neq k}}^{i}(\delta_j - t) \end{aligned} \tag{4.54}$$

In the following discussion we need to know the value of q in relation to the δ_j. Using expression (4.6) we have

$$\begin{aligned} q = t_0 + \frac{\psi_0}{\psi_0'} &= \frac{1}{\psi_0'}(t_0\psi_0' + \psi_0) \\ &= \frac{1}{\psi'(t_0)}\left(t_0\sum_{j=1}^{i}\frac{z_j^2}{(\delta_j - t_0)^2} + \sum_{j=1}^{i}\frac{z_j^2}{\delta_j - t_0}\right) \\ &= \frac{1}{\psi'(t_0)}\sum_{j=1}^{i}\frac{z_j^2\delta_j}{(\delta_j - t_0)^2} \\ &= \left(\sum_{j=1}^{i}\frac{z_j^2}{(\delta_j - t_0)^2}\delta_j\right) \Big/ \left(\sum_{j=1}^{i}\frac{z_j^2}{(\delta_j - t_0)^2}\right) \end{aligned}$$

Thus we have

$$\delta_1 < q < \delta_i = 0$$

(to establish the above inequality we have used the fact that $\delta_1 < \delta_2 < \cdots < \delta_i$).

We note the following features of the polynomial h

- h has degree i
- $h(t_0) = h'(t_0) = 0$

To study the location of the zeros of h we need to consider two cases for q

1. $\delta_l < q < \delta_{l+1}$ $(1 < l < i)$

   $h(\delta_k)$ and $h(\delta_{k+1})$ have opposite signs for $k \neq l$, and h has the same sign at

δ_l, q and δ_{l+1}, hence there are $l-1+(i-l-1)=i-2$ negative zeros.

(Note: $\delta_k \leq 0$ for $k \leq i$.)

2. $q=\delta_l$

   $h(\delta_k)$ and $h(\delta_{k+1})$ have opposite signs for $k, k+1 \neq l$ and $h(q)=0$, hence there are $(l-2)+1+(i-l-1)=i-2$ negative zeros.

As in both cases we have $i-2$ negative zeros and h has a double zero at t_0, we know that t_0 is the only positive zero. This means that h will not change sign for $t>0$ and will have the same sign as $h(\delta_i)=h(0)$ which is

$$\text{sign}(h(0))=(-1)^{i-1}$$

whence

$$\text{sign}\left(\frac{p}{q-t}-\psi(t)\right)=(-1)^{i-1}(-1)^{i+1}=(-1)^2=+1$$

and thus

$$\frac{p}{q-t}>\psi(t), \quad \text{for} \quad t>0, t\neq t_0.$$

The proof for $(r+s/(\delta-t))>\varphi(t)$ is exactly similar to this.

4.9 Alternative proof for Result 2

As mentioned before, we include the proof that follows, although not necessary for our development, because that which appears in Bunch et al. [13] is in error.

The sequence $\{t_k\}$ defined as before, converges to the solution μ_i. In other words $t_k \to \mu_i$ as $k \to +\infty$.

It suffices to prove that $\mu_i - t_k \leq \gamma^k(\mu_i - t_0)$ where $0<\gamma<1$. If we show that $t_2 - t_1 \geq \beta(\mu_i - t_1)$ for any $t_1 \in [t_0, \mu_i]$, with $0<\beta<1$, β independent of t_1 and t_2 derived from t_1 as before, then we have all that we need. If this happens we have

$$\mu_i - t_2 \leq \mu_i - t_1 - \beta(\mu_i - t_1) = (1-\beta)(\mu_i - t_1)$$

As any pair of consecutive iterations is obtained in the same way as t_2 is obtained from t_1, we know the result using t_1 and t_2 is true for any pair $\{t_k, t_{k+1}\}$ and thus obtain

$$\mu_i - t_{k+1} \leq (1-\beta)(\mu_i - t_k) \leq (1-\beta)^k(\mu_i - t_1) \leq (1-\beta)^{k+1}(\mu_i - t_0)$$

where t_1 is any real number in the interval $[t_0, \mu_i]$. To make the notation easier we will now use μ instead of μ_i.

Consider the intersection of the tangent line of $-\psi(t)$ at t_1 with the curve $1+r+s/(\delta-t)$ (which approximates $1+\varphi$). Call the abscissa of this intersection point τ (see Figure 4.9).

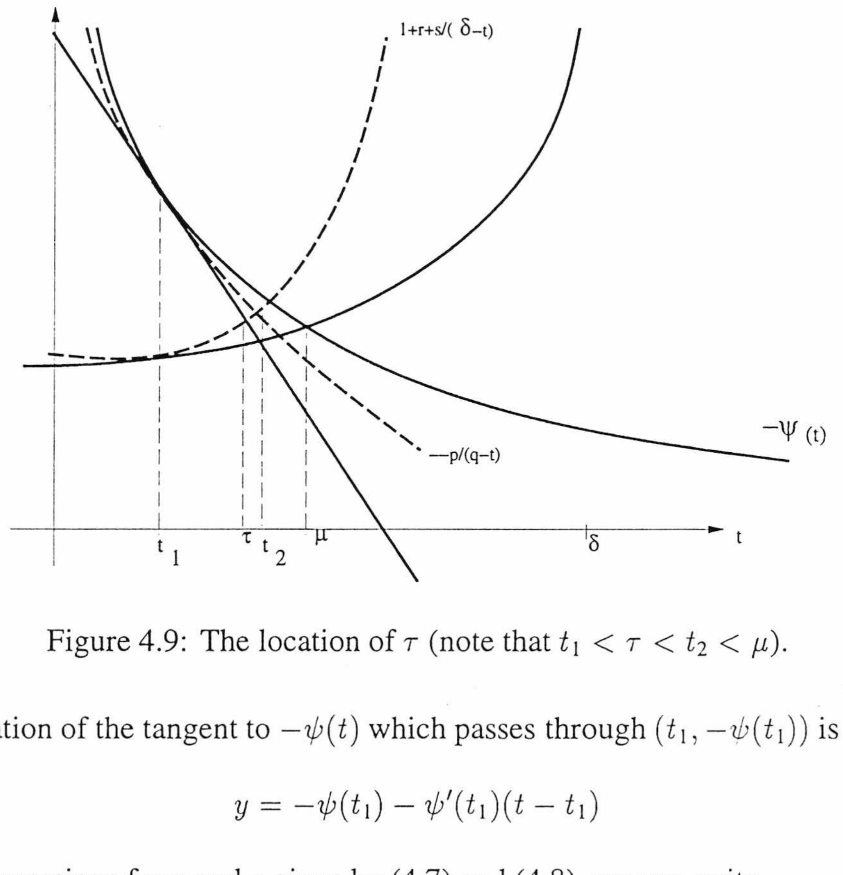

Figure 4.9: The location of τ (note that $t_1 < \tau < t_2 < \mu$).

The equation of the tangent to $-\psi(t)$ which passes through $(t_1, -\psi(t_1))$ is

$$y = -\psi(t_1) - \psi'(t_1)(t - t_1) \tag{4.55}$$

Using the expressions for r and s given by (4.7) and (4.8), we can write

$$1 + r + \frac{s}{\delta - t} = 1 + \varphi(t_1) - (\delta - t_1)\varphi'(t_1) + \frac{(\delta - t_1)^2\varphi'(t_1)}{\delta - t}$$

after rearranging the terms we get

$$1 + r + \frac{s}{\delta - t} = 1 + \varphi(t_1) + (\delta - t_1)\varphi'(t_1)\left(\frac{t - t_1}{\delta - t}\right) \tag{4.56}$$

As τ is the intersection between line (4.55) and curve (4.56) we must have

$$\begin{aligned} -[\psi(t_1) + \psi'(t_1)(\tau - t_1)] &= 1 + r + \frac{s}{\delta - \tau} \\ &= 1 + \varphi(t_1) + \varphi'(t_1)(\delta - t_1)\left(\frac{\tau - t_1}{\delta - \tau}\right) \end{aligned} \tag{4.57}$$

As observed before, we know from Figure 4.9 that $t_1 < \tau < t_2$ and thus $\tau - t_1 < t_2 - t_1$ so that if we can bound the quantity $\tau - t_1$ from below, by a constant multiplying $\mu - t_1$ we are done. For this we will use (4.57), but first we will make some observations that will aid our analysis.

From the graphs of the functions φ and ψ we can observe that (see Fig. 4.10)

- φ' is an increasing function in the interval $[t_0, \mu]$
- ψ' is a decreasing function in the interval $(t_0, \mu]$

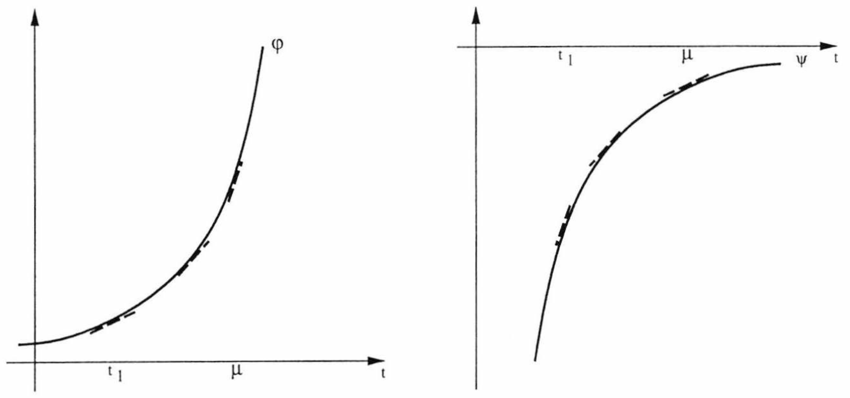

Figure 4.10: The curves φ and ψ and the behaviour of φ' and ψ'.

Furthermore

$$\begin{aligned} \frac{\psi(\mu) - \psi(t_1)}{\mu - t_1} &\geq \psi'(\mu) \\ \psi(\mu) - \psi(t_1) &\geq \psi'(\mu)(\mu - t_1) \end{aligned}$$

and similarly $\varphi(\mu) - \varphi(t_1) \geq \varphi'(t_0)(\mu - t_1)$

Now continuing from (4.57) and rearranging the terms we get

$$[\varphi'(t_1)\frac{\delta - t_1}{\delta - \tau} + \psi'(t_1)](\tau - t_1) = -1 - \varphi(t_1) - \psi(t_1) \tag{4.58}$$

Recalling that $\psi(\mu) + \varphi(\mu) = -1$ (because μ is the root) we can write (4.58) as

$$\begin{aligned}[\varphi'(t_1)\frac{\delta - t_1}{\delta - \tau} + \psi'(t_1)](\tau - t_1) &= \psi(\mu) - \psi(t_1) + \varphi(\mu) - \varphi(t_1) \\ &\geq \psi'(\mu)(\mu - t_1) + \varphi'(t_0)(\mu - t_1) \\ &= (\mu - t_1)(\psi'(\mu) + \varphi'(t_0)) \end{aligned} \tag{4.59}$$

From the locations of t_0, t_1, τ, δ and μ we know that $\delta - t_1 \leq \delta - t_0$ and $\delta - \tau \geq \delta - \mu$. This, along with the previous observations about φ' and ψ', gives the following inequality

$$\varphi'(t_1)\frac{\delta - t_1}{\delta - \tau} + \psi'(t_1) \leq \varphi'(\mu)\left(\frac{\delta - t_0}{\delta - \mu}\right) + \psi'(t_0)$$

which together with inequality (4.59) gives

$$(\tau - t_1)\left(\varphi'(\mu)\left(\frac{\delta - t_0}{\delta - \mu}\right) + \psi'(t_0)\right) \geq (\mu - t_1)\left(\psi'(\mu) + \varphi'(t_0)\right)$$

And this finally gives the inequality we were looking for

$$\tau - t_1 \geq \frac{\psi'(\mu) + \varphi'(t_0)}{\varphi'(\mu)(\frac{\delta - t_0}{\delta - \mu}) + \psi'(t_0)}(\mu - t_1)$$

because

$$\frac{\psi'(\mu) + \varphi'(t_0)}{\varphi'(\mu)(\frac{\delta - t_0}{\delta - \mu}) + \psi'(t_0)} < 1.$$

This last inequality is easily seen to be true, since

$$\psi'(\mu) + \varphi'(t_0) < \psi'(t_0) + \varphi'(\mu) < \psi'(t_0) + \varphi'(\mu)\frac{\delta - t_0}{\delta - \mu}$$

as $(\delta - t_0)/(\delta - \mu) > 1$.

Chapter 5

Implementing the Divide-and-Conquer method

This chapter describes work carried out on the implementation of the Divide-and-Conquer algorithm and issues related to the implementation.

5.1 Software and hardware environment

The Divide-and-Conquer (D&C) algorithm was implemented using the OCCAM2 language [49]. Occam is a language with built-in statements to address the concurrent execution of processes which interact with each other via the exchange of messages. It uses the synchronous, *rendez-vous* type model of communication.

The algorithm was implemented on a Meiko Computing Surface, a parallel computer based on Inmos transputers. Transputers are processors which have a built in FPU implementing the IEEE 754 floating-point arithmetic standard. Each transputer possess four links, which allow them to be joined to each other in a configuration chosen by the user. The machine installed at the University of Kent has a mixed pool of transputers including T414s and T800s, with 1, 2 and 4Mbytes of memory. The processor used in our implementation was the T800 transputer running at 20MHz. For the solution of relatively large eigenproblems we require at least 4Mbytes of memory. No more than 24 T800s with this amount of memory were available at any one time for our use. As the number of processors required by our general implementation is a power of two we have been able to run our program with at most 16 transputers. In addition, the maximum 4M bytes of memory available in each processor is still small for the D & C algorithm. At one point it is necessary to compute a product of matrices, and consequently, the order of complexity of the algorithm is n^3, where n is the order of the matrix as originally given. This meant our program could be tested with matrices of dimension at most 1000. Ipsen and Jessup [40] have reported on this requirement of memory, remarking that the Divide-and-Conquer algorithm requires more storage, and thus, can be used for smaller matrices than other algorithms such as bisection and multisection. They have been able to test the D&C algorithm with matrices of order up to 2000 on the iPSC-1/d5M, but have only provided the results for matrices of order up to 512. Dongarra and Sorensen [20] did not mention the maximum order they have run their tests, but they report results with matrices of order up to 500.

Our implementation of the Divide-and-Conquer algorithm was developed in several stages. To begin with, we wrote a simple prototype program using MATLAB [54] in which we were able to use with advantage mathematical routines already implemented within that package (e.g., the QR algorithm and matrix algebra routines). The form of this initial version was that it ran on a single processor. For this, the original matrix is partitioned into two parts and the eigensystem of each part computed separately and sequentially. This implementation allowed a better understanding of the method and permitted us to focus more on the method and its features than on the details of programming. MATLAB also, makes it easier to test the algorithm and to compare answers with those provided by its own library routines for solving eigenproblems. Subsequently, we wrote a simple version for three transputers, and then finally a completely general program which may be configured over a variable number of processors.

MATLAB Quoting the reference guide for MATLAB [54]: "MATLAB is a technical computing environment for high-performance numeric computation and visualization. MATLAB integrates numerical analysis, matrix computation, signal processing, and graphics in an easy to use environment where problems and solutions are expressed just as they are written mathematically–without traditional programming. "

The above quote describes MATLAB in general terms. From the user's point of view, MATLAB is a very useful tool for developing programs and testing new ideas. With all its built-in routines, it is very easy to perform experiments and to correct or confirm hypotheses. It is especially useful when working with matrices and vectors on which operations may be expressed in the same way as for scalars. With MATLAB, expressing a product of two matrices or of a matrix and a vector is as easy as expressing the product of two numbers. Even though we can use MATLAB interactively, we may also write small programs calling all MATLAB's library routines. As MATLAB has routines to compute the eigenvalues and eigenvectors of a matrix, it was also very useful for comparing results with those obtained by our implementation on transputers. This made it easier to debug our program at various stages of development. We also used the MATLAB version of the Divide-and-Conquer algorithm to verify the intermediate calculations given at each stage of the parallel version on the Meiko system. With MATLAB we could also observe, in an easy way, the behaviour of the eigenvalues of a certain matrix when some parameters are changed. In other words, even though we only used a small subset of what MATLAB has to offer, it made the development of our implementation of the Divide-and-Conquer algorithm and its subsequent testing, much easier.

5.2 The three-processor prototype implementation

Our three-processor implementation was written in a very simple way to allow us to gain experience of how the algorithm would behave in a parallel environment. The intention was that, if it delivered reasonable results, we could then expect to obtain a far better

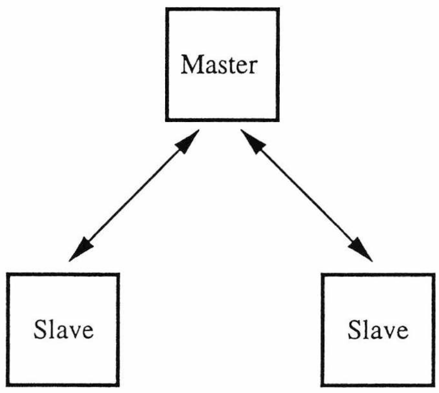

Figure 5.1: Three-processor implementation.

performance when the program was developed with more care taken to optimize processor activity.

In this version the processors are organized in a "master-slave" fashion where we employed one transputer as the "master" and the other two as "slaves". They are interconnected as shown in Figure 5.1. A limiting feature of our program, but imposed to keep things simple, is that the master is idle whenever the slaves are busy, and vice-versa. Of course, with this constraint we could not expect very high performance, but nevertheless we were able to obtain a good gain over the time taken by the QR-algorithm and efficiencies of 80% when compared with the sequential D & C (see next chapter).

The communication structure is also simple and the transfer of data occurs just between the master and the slaves – there is no communication between the two slaves.

5.2.1 Description of implementation

The master undertakes general tasks such as splitting the original matrix into two and sending each part to the slaves; it controls all the tasks as well as performing some of the computation.

First, the master computes (or reads from a file) the initial matrix data and sends part of this to each of the two slaves in parallel. Assuming that the dimension of the matrix n

is divisible by 2, each slave receives half of the original tridiagonal matrix. In practice the dimensions of each sub-problem are $dim1 = \lfloor n/2 \rfloor$ and $dim2 = n - dim1$. The slaves then compute the eigensystems of the partial matrices (T_1 and T_2) assigned to them, using the *QR algorithm*, and compute "half" of the vector z. They then report back their results to the master which amalgamates them by merging the two sets of eigenvalues received. This data constitutes the diagonal entries of the matrix D. The two halves of the vector z are assembled using the same ordering scheme employed to create D from the eigenvalues.

At this stage, the master performs tests on D and z to decide if deflation is necessary and then carries out the appropriate actions. These are the rotations needed to zero one or more elements of z (when the corresponding elements of D are equal) and the permutations to send any zero elements to the bottom of the vector, along with the corresponding reordering of the matrix D (stored as an array). When all the zero or near zero entries are banished to the bottom of the vector z, the routine to solve the secular equation is applied to the upper part of the matrix $D + \rho zz^t$ (comprising the distinct elements of D and the non-zero elements of z). In the initial implementation, this task is carried out by the master alone, but it could be divided and assigned to the two slave processors, because the computation of any root is completely independent of the others. The eigenvectors of $D + \rho zz^t$ are computed as described in Chapter 3 and subsequently half of them are sent to each of the slaves by the master processor. Each slave computes the product of matrices $Q_i(eigvecD)_i$ where $i = 1, 2;$, Q_i is the matrix of eigenvectors (of T_i) computed using the *QR algorithm* by processor i and $(eigvecD)_i$ is the part of the matrix of eigenvectors of $D + \rho zz^t$ which is assigned to each processor. The final eigenvectors are then sent back to the master and printed along with the eigenvalues.

We must note that in spite of talking about the *QR algorithm* as it is usually known, the implementation we employed is a version of the routine IMTQL2 (as it appears in [88]), an implementation of the implicit *QL* algorithm. IMTQL2 routine was written in ALGOL 60, and for our use on transputers, we have translated into OCCAM2.

As mentioned above, the program has two distinct processes: one for the master and one for the slaves. The structure of the master is that shown below:

```
... assemble the data for the diagonal and off-diagonal elements of the tridiagonal matrix to be worked on
... output to the slaves the index of the diagonal element to be modified ($n/2$ for server 1; and 1 for server 2)
    and the off-diagonal element which will be subtracted from this
... output to each of the slaves that portion of the diagonal and off-diagonal elements that each will need
... receive the eigenvalues ($d$) and the half vectors $z_1$ and $z_2$ – which together make up the vector z
... merge the two sets of eigenvalues and reorganize the elements of $z$ correspondingly
... test to check if two elements of $d$ are the "same" and if any $z_i$ are "zero", and deflate when necessary
... compute the eigenvalues of $D + \rho zz^t$ by the rational equation method (a call to the subroutine EIGDPZZ)
... compute the eigenvectors of $D + \rho zz^t$
... update the eigenvectors of $D + \rho zz^t$ "computing" $J \times \Pi 2 \times eigvecD$
... send the first dim1 lines of the eigenvectors matrix to one server and the remaining ones to the other
... receive the final eigenvectors
```

The structure of the slaves is much simpler, because in this initial implementation the majority of the tasks are done by the master.

```
... receive the index of the element to be modified and the off-diagonal element that is in position $dim1$
... receive the diagonal and off-diagonal elements of the tridiagonal matrix
... compute the eigensystem of its matrix using the QL algorithm
... output the eigenvalues computed
... compute its part of the vector $z$ and output the result
... receive "half" of the lines of the eigenvectors matrix of $D + \rho zz^t$
... compute the product of matrices $eigvecQL \times eigvec$ (where $eigvecQL$ is the eigenvectors matrix
    computed by the QL and $eigvec$ is the part of the eigenvectors matrix received)
... output the part of the eigenvectors matrix computed
```

5.3 The multiple-processor implementation

As a result of obtaining encouraging performance with our three transputers version, we turned our attention to a more general implementation. For this, we wanted a way of organizing tasks that would not entail leaving a processor completely idle while awaiting the result of some computation performed elsewhere (as was the case in our three transputer implementation). We were able to keep all of the processors occupied for most of the time. Some calculations are repeated in all the processors because we found the gains we obtained by distributing some tasks did not compensate for the penalty incurred to communicating the data.

5.3.1 The topology of the processor interconnections

The processors are connected as in a rectangular grid with four transputers in each row and as many rows as appropriate for the number of processors employed. Because of the way the algorithm works, the number of processors is always a power of two: this seemed to be the most natural choice for the method. We believe our implementation would work with any number of processors, but the organization of operations would be more complicated. The algorithm was tested with 4,8 and 16 transputers, because the latter was the largest number of transputers, with a memory of 4 MB, available for our grid. As the algorithm needs to deal with at least two dense matrices we could not employ other transputers that have less memory (2 and 1 MB). The processors are linked as in Figure 5.2 (which shows the case for 16 processors). The numbers indicate the number of the processor that is being represented, for easier reference and comparison with another topology.

As we did not have a large number of processors available, the organization of a grid with four processors in each row (and not eight, for example) seemed to be the best option. Our first multiple-processor implementation worked with just four processors and our intention with that version was that we could construct a basic "building block" and use it when more processors were available. For the four processors version it would probably be more

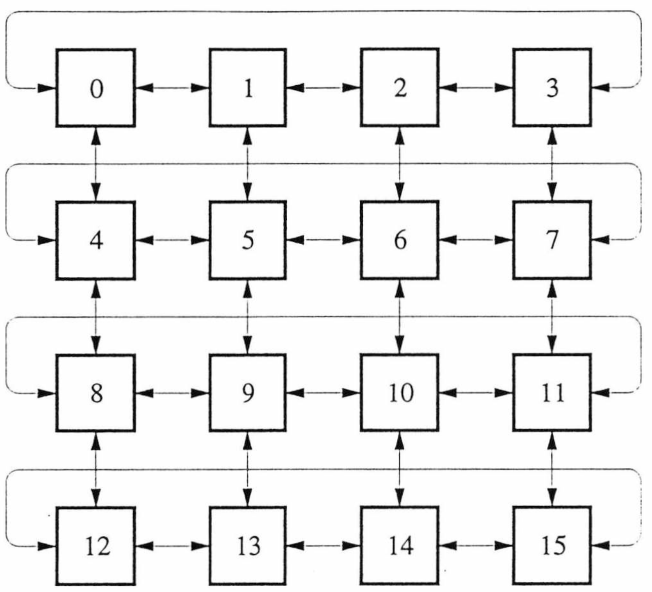

Figure 5.2: Multiple-processor Topology – 16 processors

efficient to use a 2 by 2 grid as the one shown in Figure 5.3-b, but this basic structure does not seem to lead to a good extension when a maximum of four links is available per processor (as is the case with transputers).

Figures 5.3 and 5.4 show the number of steps needed to send data from processor 0 to processors 4,5, 6 or 7, with one of the possibilities for effecting this example of communication. Using the 2×2 grids, it takes two steps to send data from processor 0 to processor 4, and three steps to send to the other processors in the same block (processors 5,6,7). Using the "full" grid (Figure 5.4), one step is necessary to reach processor 4; two steps to send to processors 5 and 7 (see Figure 5.4-b and d) and three steps to processor 6 (Figure 5.4-c), etc.. This means, that at this stage, and depending on the kind of communication needed, there could be some advantage in using the 2×2 grids, but if the intention is ultimately to employ a greater number of processors, this advantage is lost. It can easily be seen, that using the 2×2 grids (linked as shown in the diagrams), we would require four steps to send data from processor 0 to processor 8 and five steps to processors 9,10 or 11, while using the proposed topology we need two steps to send data to processor

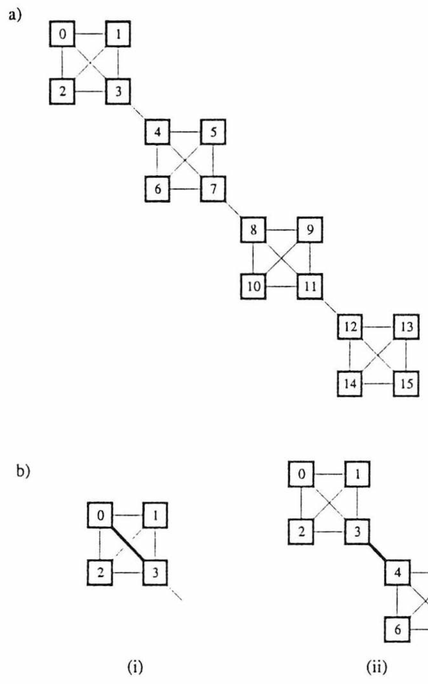

Figure 5.3: The "2 × 2 grid – topology" and the required steps to send data from processor 0 to any other processor

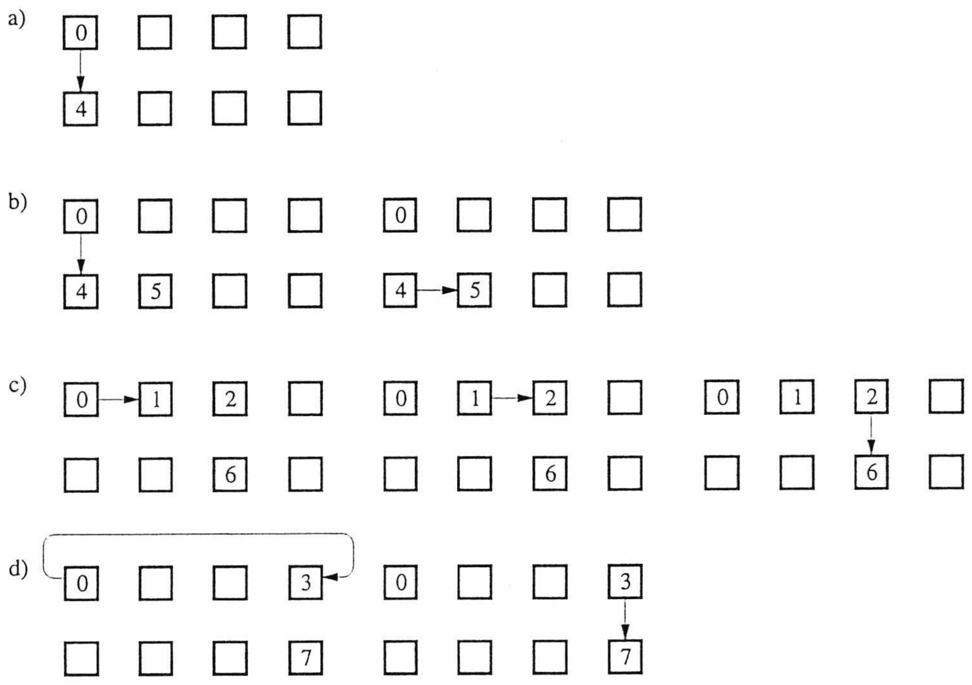

Figure 5.4: The proposed topology and the required steps to send data from processor 0 to any other processor – using 8 processors

8; three to processor 9 and 11; and four steps to processor 10. When we analyse the sending of data from processor 0 to more distant processors, the differences become greater. For example, from processor 0 to processor 12, six stages are required in the 2×2 grids and seven steps when sending to processors 13,14 or 15. Our preferred topology requires just three steps to send to processor 12; four to send to processor 13 and 15 and five to processor 14. We are taking as an example the number of communication steps needed for the path that goes from processor 0 to any other processor, but a similar analysis was carried out of the specific costs of communication between the different processors within the arrays, and the interconnection we have adopted proved the most effective.

One could envisage other ways of linking the 2×2 grids, but these are prohibited because of the limitation to a maximum of four of the number of links to an individual processor. We investigated alternative topologies, but either these were too complicated to program or they did not give any gain over our preferred arrangement described above. The proposed topology is as efficient or better than the more complex ones (as far as communication is concerned), but it has the advantage of being a simpler configuration. It allows the necessary interprocessor communications to be made using at most three steps for 8 processors and five steps for 16 processors.

We also chose to join the left-most processor (0,4,8, etc.) with the right-most one (3,7,11,...). This minimises the path of communication as much as possible, when communication is needed among all the processors in the same row. With the constraint that the total number of links available per transputer is four, this was the best that could be achieved in each row (whilst maintaining our idea of having an easily extendable topology). The processors situated in the first and last rows of the grid, with exception of processor 0 – which needs one link to connect with the I/O facilities – have yet one link available that we could use to join the extreme processors in each column and thus reduce the path of communication when all processors need to exchange information among themselves, but because of the I/O requirements in processor 0, this was not possible. Joining the extreme processors in each row instead of those in the same column is to be preferred, because communication among processors located in the same row has higher priority. Steps of low order already require this type of communication whereas communication up and down columns is only needed from the third step onwards (see Section 3.5). The problem is that we do not have the (ideal) situation where all processors would interconnect together, giving the minimum number of steps for communication.

5.3.2 Description of implementation

As mentioned earlier, in this implementation we do not have a processor acting as a master, dedicated to organizing the computations, but as the processor 0 is the only one which has access to I/O facilities, it assumes this role at times. The initial data is computed or read from a file by processor 0, which then makes necessary adjustments (calculation of dimensions and initial indices) and distributes this data to the other processors. If the matrix does not come from a file but is created from an algebraic formula, each processor could compute its corresponding component. The given tridiagonal matrix is partitioned into as many parts as there are processors.

As in the three-processors version, each transputer applies the QL algorithm to its local component matrix and computes the part of the vector z for which it is responsible. The results (both eigenvalues and the vector z) are merged by communicating them two-by-two. Only processors that are nearest neighbours need to communicate between themselves at this stage. In this way, processor 0 sends and receives messages to and from processor 1, and there is a similar interaction between processors 2 and 3, 4 and 5, and so on. The communication is performed in parallel, in other words, the data is sent and received at the same time by using a PAR command with the send/receive OCCAM instructions.

The main tasks executed in all processors are listed below.

1. input and output (if needed) of initial data: (diagonal elements (d); off-diagonal elements (e); off-diagonal element which will be subtracted from the appropriate diagonal element ($mod.e$); the dimension of the problem (n); the dimension assigned to that processor (dim))

2. call QL –using the local parts of the vectors d and e
3. compute part of z
4. compute ρ
5. output/input the eigenvalues computed by QL, the part of z computed and respective dimensions
6. merge the eigenvalues computed with those received (the same data is used in procs. 0 and 1; 2 and 3, etc.)
7. deflate (again, operating on the same data in pairs of processors, as in the previous item)
8. call eigdpzz (each processor computes a *different* set of eigenvalues – or roots μ – and works on a different range of eigenvalues)
9. send/receive μ
10. compute eigenvectors (each processor computes different parts of the matrix of eigenvectors)
11. send/receive partialsums (to compute norms)
12. merge/compute the eigenvalues computed in item 8 with the ones that were deflated (and the result will be the ordered d for the next step)
13. communicate $eigvecD$ (not applicable on the first step)
14. compute products of eigenvectors: $eigvec = eigvec1 \times eigvecD$

The second and all subsequent steps (that is, on all occasions, except the first, when this complete routine is executed) begin with task 3 of the above description, but tasks 3 and 4 are slightly different. Task 3 is not done by all processors, but just by those that have access to the last row of the first matrix of eigenvectors or the first row of the second matrix of eigenvectors. These processors compute part of the vector z and then communicate it to the other processors. The same applies with task 4: not all processors will have access to the off-diagonal element needed to compute ρ, so some will compute it and the rest will receive it. With the exception of this first difference and observing that the communication routines will differ from processor to processor (as explained later), the remaining tasks will be the same until task 12. Task 13 is not needed in the first step. Only in the first step is it unnecessary to communicate any part of the eigenvectors matrix, since each processor already has what it needs. Task 14 is slightly modified, because from the second step onwards, we will not have the eigenvectors computed by the QL, but we will use in their place the eigenvectors computed in the previous step, i.e., we set *eigvecl* $=$ *eigvec*. In step 1 *eigvecl* is the matrix of eigenvectors computed by the QL algorithm. For steps with $nstep > 2$, the main tasks are those described above, but there are some additional small calculations to control the number of rows involved in each step: Which are the processors that will compute ρ? What are those positions where we should start the storage of eigenvalues and eigenvectors? and some other details. In the last step of the algorithm, we do not need task 12, because the eigenvalues are ordered now simply for printing, but retaining this task does not increase running time very much and simplifies the code. After all computations are complete, the eigenvalues and eigenvectors are sent to processor 0, which computes residual norms, checks for orthogonality of eigenvectors and prints the results if that is desired.

5.3.3 Data Storage

As we are working with a symmetric tridiagonal matrix, the initial data is composed just of two vectors (d and e – the diagonal and off-diagonal matrix elements, respectively). These vectors are partitioned to correspond with the number, P, of processors in use. If this number exactly divides the dimension n of the matrix, then all processors will receive the same number of components to work on. Should P not divide n, the first set of processors will receive $\lfloor n/P \rfloor$ components of d and the final processors will receive $\lfloor n/P \rfloor + 1$ components. The number of processors that will work with the greater number of elements is the remainder when n is divided by P. In this way no one processor will have a much larger load than any other. Certainly, when the steps of the algorithm proceed, the difference between the sizes of the problems that are being worked on in each group of processors, will increase, but this difference will still be not very significant. Besides, the dimension with which each individual processor is working, remains the same throughout the algorithm. Once the partial dimensions are established, each processor in turn receives its component parts of the vectors d and e. Processor 0 will work with the first part of the vectors d and e, processor 1 will have the second part and so on. Suppose, for example, that we are working with 4 processors ($P = 4$) and the dimension of the matrix is 18 ($n = 18$). We then have $n/P = 18/4 = 4$ with remainder $r = 2$. This means processors 0 and 1 work on sub-problems with dimension $n = 4$ and processors 2 and 3 work with $n = 5$. Processor 0 will work with components d_0, d_1, d_2 and d_3 and e_0, e_1 and e_2, processor 1 will work with components d_4, d_5, d_6 and d_7, e_4, e_5, e_6, processor 2 with $d_8, d_9, d_{10}, d_{11}, d_{12}$, e_8, e_9, e_{10}, e_{11} and processor 3 with the remaining elements ($d_{13}, d_{14}, d_{15}, d_{16}, d_{17}$ $e_{13}, e_{14}, e_{15}, e_{16}$). Recall that in the tearing process the elements e_3, e_7 and e_{12} are used to modify appropriately the corresponding sub-matrices. The component e_3 is sent to processor 1 (of course processor 0 also uses it), e_7 is sent to processors 1 and 2 and e_{12} is sent to processors 2 and 3.

These initial dimensions, computed by processor 0 and later transferred to each processor, are maintained throughout the algorithm and are used to establish how many eigenvalues and how many rows of eigenvectors are computed at each step. As these partial dimensions can differ from one processor to another, the control of subsequent dimensions or how many elements should be received in each instance of communication is not straightforward, but it is known when needed. One alternative considered, was to maintain an array containing the partial dimensions of each processor. This does not pose a problem if the number of processors is small, but nevertheless it does require access to an array every time the information is needed. We opted for sending the required dimension, just before it is needed, because we had to send other data, and could transmit this information along with the rest in a single OCCAM protocol. All the same, we think that the other approach could be used without much degradation of execution time. An advantage of using an array for storing these dimensions would be a simpler program, and the data would be accessible at any time during processing. Again, these dimensions could be computed in each processor or could be transmitted by processor 0 to the other processors.

5.3.3.1 The eigenvector matrices

Usually, when we think about computing eigenvectors, the idea that comes to mind is to calculate whole columns of elements. Our first approach was exactly this, and the computation of the set of eigenvectors was distributed between the processors in groups of columns. With this partition by columns, we have two alternatives for computing the products of the two eigenvector matrices. In either case, much communication is needed, because none of the processors has all the necessary information for performing the product. Maintaining the notation used previously for the matrices involved in these products, let us call

- *eigvecl* the eigenvectors matrix formed from the eigenvectors computed by the QL algorithm if we are in the first step, and in later steps, the matrix that comes from the previous step
- *eigvecD* the matrix computed in the current step using routine *compeigvec*, which computes the eigenvectors of matrix $D + \rho z z^t$, using formula (3.14)
- *eigvec* the matrix resulting from the product of these two matrices.

We suppose that P divides n exactly and that each initial sub-problem has dimension $dim = n/P$. We consider initially the first step of the Divide-and-Conquer algorithm. To exemplify, we will examine what happens on processors 0 and 1, as this is entirely similar, at this stage, to what happens with other pairs of adjacent processors. Figure 5.5 shows the shapes of matrices *eigvecl* and *eigvecD* and how they are stored, i.e., the upper $dim \times dim$ block of *eigvecl* is computed and stored in processor 0 ($P0$); the lower block is computed and stored in processor 1 ($P1$); the blanks represent blocks of null matrices, composed simply of zeros and, obviously, are not stored anywhere. The first dim columns of matrix *eigvecD* have been computed and stored in $P0$ and the remaining columns computed and stored in $P1$. Observing Figure 5.5 we can see that to compute the product *eigvec* $=$ *eigvecl* $\times$ *eigvecD*, we need to have the corresponding blocks of each matrix stored in

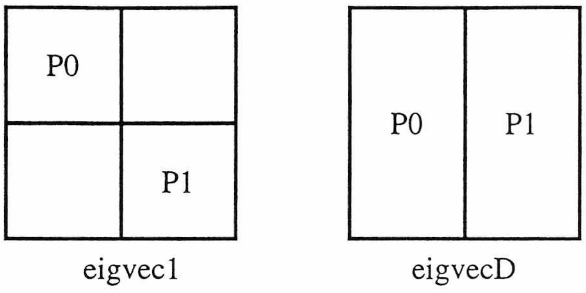

Figure 5.5: Distribution of *eigvec1* and *eigvecD* between P0 and P1.

the same processor. For example, to compute the first dim rows of *eigvec* we need the upper left $dim \times dim$ block of *eigvec1* (which is stored in $P0$) and the first dim rows of *eigvecD* (which is partitioned between $P0$ and $P1$). Just the upper half of *eigvecD* is needed, because the upper-right block of *eigvec1* is a null matrix. To accomplish this, i.e., to have all the necessary information in the same processor we have two alternatives:

Alternative 1: $P0$ receives the lower block of *eigvec1* (*eigvec1*$[dim+1:n; dim+1:n]$, in MATLAB notation) and then computes *eigvec*$[1:n;1:dim]$, the first *dim* columns of *eigvec*

Alternative 2: $P0$ receives the portion of the matrix *eigvecD* which it needs, *eigvecD*$[1:dim; dim+1:n]$, and thus is able to compute *eigvec*$[1:dim;1:n]$, the first *dim* rows of *eigvec*.

In the first way, the eigenvector matrix *eigvec* will be computed and stored by columns and in the second way it will be computed and stored by rows. Figure 5.6 shows how the matrices would be stored for these transfers of data, and the product and subsequent storage of *eigvec*. Again the notations $P0$ and $P1$ in the figures indicate that the corresponding block is stored in processors 0 and 1, respectively. In the case of alternative 1, this means that both processors will have both non-null blocks of *eigvec1*.

In the second step, four processors work together and need to exchange information. The matrix *eigvec* from the previous step is now *eigvec1* and, if we consider alternative 2

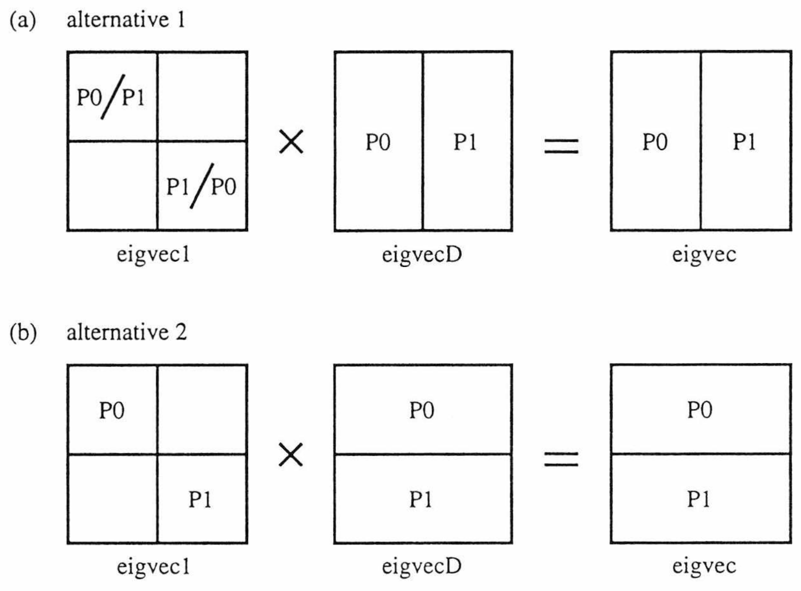

Figure 5.6: The parts of the matrices needed by each processor to compute the product.

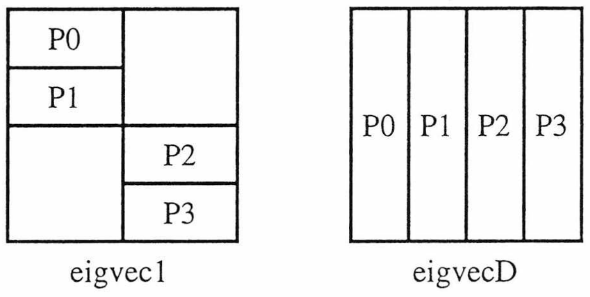

Figure 5.7: Distribution of *eigvec1* and *eigvecD* among P0, P1, P2 and P3 in step 2.

above, it is stored by groups of rows. If we still suppose the computation of *eigvecD* is by groups of columns, the distribution of these two matrices, in four processors, is represented in Figure 5.7.

Choosing alternative 2, we observe that we need data from all four processors to compute each group of rows. For example, in the case of processor 0, as it has a sub-matrix of *eigvec1* with dimension $dim \times 2dim$, it needs all the upper *2dim* rows of *eigvecD* to compute the first dim rows of *eigvec* (see Figure 5.8). As these upper $2dim$ rows are distributed among the four processors, P0 needs to receive data from P1, P2 and P3. The same thing happens

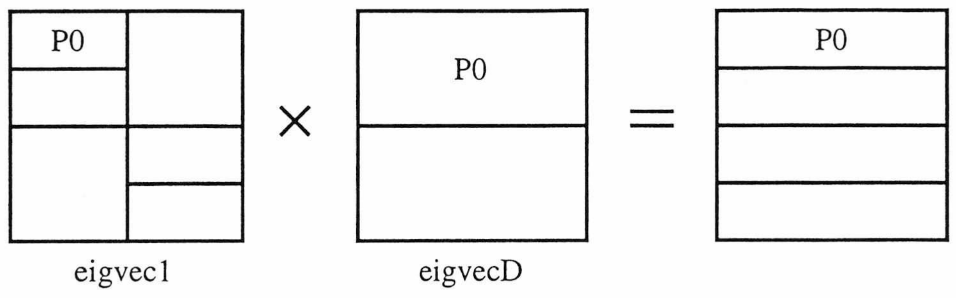

Figure 5.8: The parts of the matrices needed by processor 0 to compute the product.

with the other processors: each of them needs $2dim$ rows of *eigvecD* and so, each processor must exchange data with the other three. If we choose alternative 1, i.e, compute and store *eigvec* by columns, a similar analysis can be made and again we can verify that we need data from the four processors, to be able to compute one column of *eigvec*. In this way, using any of the two alternatives described, but computing *eigvecD* always by columns, we need to transfer data between two processors in the first step, and among four processors in the second step.

Even though this way of computing the eigenvectors column by column seemed to be the more natural one, because of the excessive communication it entails and the amount of storage necessary, we had to think of another way. As we tried to run bigger problems, we were confronted with a lack of memory, caused by trying to store so much data. The main reason of this was the matrices that are required by the algorithm. As the program was originally written, we needed to store three full matrices, one for each of the eigenvectors matrices and one for the resulting product. We could overwrite one of these with the product thus saving one matrix, but the two others would still have to be full matrices of order $n \times n$ and $n/P \times n$. We could use two matrices to store *eigvec1* instead of just one: one with a dimension of $n/2 \times n$ and another with dimension $n/2 \times n/P$ (and maintain the second matrix – *eigvecD* with dimension $n/P \times n$) and thus save some memory, but this would involve storing part of the eigenvectors in the smaller matrix and would make the program rather involved. In addition, the transfer of data (the computed eigenvectors) would be somewhat complicated to control.

To circumvent these difficulties, we decided to try the computation of eigenvectors in blocks of rows instead of blocks of columns. This idea came from observing the "shape" of the D&C method, mainly in its first step, because the matrices that arise from the QL are square matrices with half the dimension of the sub-problem that is being solved at that stage. This means that half of each rectangular block consists just of zeros, which, clearly, will not affect the part of the product corresponding to them. Observing this, it would be better if we could have the first half of the rows in processor 0 (for example) and the second half in processor 1, without having to transfer part of them (see Figure 5.6 – alternative 2). Making this modification introduced other problems, however, and it is not a perfect solution, but we think it is an improvement over the computation by columns. This new way poses the problem of having to communicate the values of μ for each processor involved in the step (see (3.14) for computing eigenvectors when the secular equation is solved), but this involves transferring less data than when computing the eigenvectors by columns. For instance, in the previous scheme (by columns), we had to communicate blocks of matrices of order $n/P \times n/P$ already in the first step, whereas in the current scheme (by rows), we do not need to communicate any part of the eigenvectors matrix in this first step. In the second step, as mentioned in the previous paragraph, the first scheme requires communication between four processors while the second needs to exchange data simply between two neighbouring processors (see the next section for more details concerning these requirements). Computing and storing the eigenvectors by rows means that we cannot immediately compute the norms necessary to normalize the eigenvectors matrix as soon as the computation of the eigenvectors is finished, rather we compute partial sums that are used later for the computation of these norms. Clearly, these partial sums must be communicated, but again, it is one vector and not sets of vectors as the first scheme requires.

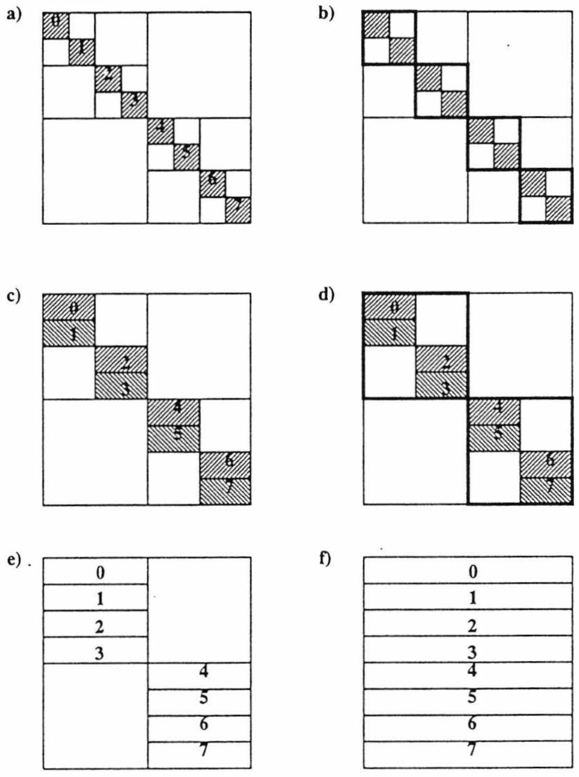

Figure 5.9: The storage of the eigenvectors matrices

5.3.3.2 Requirements for the products of matrices

In this section we explain in more detail, how the eigenvector matrices are partitioned for distribution among the processors and what effect this has on the computation of their products. Recall that we chose to make each processor compute groups of rows of the matrix *eigvecD* and that assumption underlies this section. Some of the topics included here are touched on again in other chapters, but we do so to emphasize these issues and to maintain the completeness of our explanation.

Figure 5.9-a shows the initial partition of the original matrix when we work with 8 processors. The eigensystems of the matrices marked as black squares in this figure are computed by the QL algorithm. The eigenvector matrix of each of these matrices assumes the same shape and each processor is responsible for the computation of one of them.

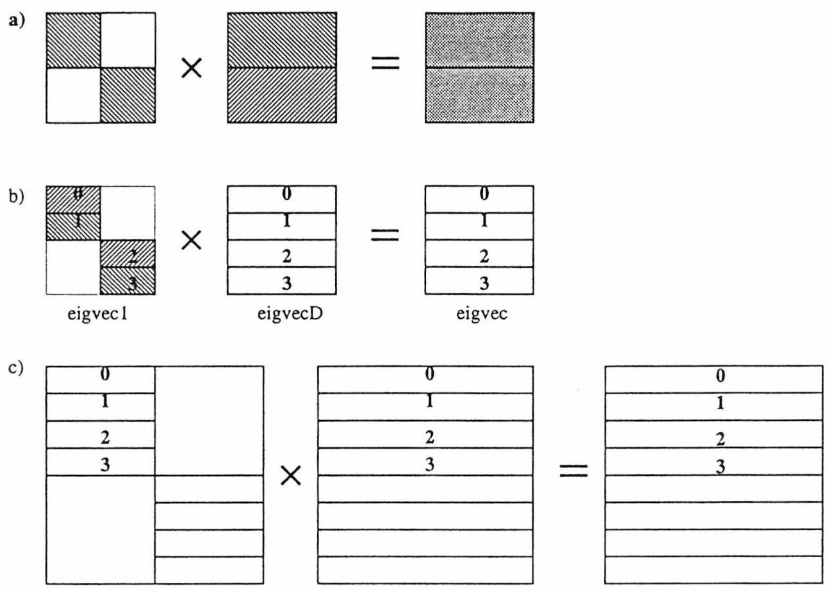

Figure 5.10: The product of *eigvec1* and *eigvecD*

After this first computation, the pairs of processors 0 and 1, 2 and 3, 4 and 5 and 6 and 7 combine their results as in Figure 5.9-b.

If we refer to those matrices via the processor number, processors 0 and 1 will use the *last* row of matrix 0 and the *first* row of matrix 1 to form the vector z (the other pairs of processors function similarly but again, for explanatory purposes, we will concentrate only on processors 0 and 1).

Now each pair of processors work together to compute the eigensystems of the "new" matrices (see Chapter 3). Processor 0 will compute the first half of the eigenvalues and the first set of rows of the eigenvector matrix, so that the eigenvectors matrix (*eigvecD*) will be stored as in Figure 5.9-c (recall that the previous (*eigvec1*) eigenvector matrices are stored as in Figure 5.9-b).

We now need to compute the product between these two sets of corresponding matrices (*eigvec1* and *eigvecD*), so that processors 0 and 1 will have to compute the product of matrices which have the format shown in Figure 5.10-a. As we can see from that figure, the upper half of the resulting matrix *eigvec* is computed from the upper matrices of *eigvec1* and *eigvecD* (as the top right block of *eigvecl* is zero). As these two matrices are both stored in processor 0, we have all the information that is needed to compute the upper half of *eigvec* in a single processor, not requiring any communication at this stage. The same happens for the bottom half of *eigvec*, where the required parts of *eigvecl* and *eigvecD* are now stored in processor 1.

After all the products are computed, the resulting eigenvector matrices (which will be *eigvecl* in the next step) are stored as in Figure 5.9-d. In this next step, information stored in four different processors will need to be exchanged. In this way, processors 0, 1, 2 and 3 (and 4, 5, 6 and 7) will need to communicate among themselves in order to compute the eigensystems of the two new bigger matrices. The first group of processors needs the last row of the eigenvector matrix which is in processor 1 and the first row of the sub-matrix stored in processor 2 (the second group needing the last row of the sub-matrix stored in processor 5 and the first row of the matrix stored in processor 6). The eigenvectors matrices will be computed and stored as in Figure 5.9-e.

Again the products between matrices represented in Figure 5.9-d and those represented in Figure 5.9-e need to be formed and the "shape" of these products is shown in Figure 5.10-b. To obtain the first quarter of the resulting matrix – *eigvec* – (which will be computed and stored in processor 0), we need portion 0 of matrix *eigvecl* and portions 0 and 1 of matrix *eigvecD* (note that all dimensions are compatible). To obtain the second quarter, we need portion 1 of *eigvecl* and again, portions 0 and 1 of *eigvecD*. The third quarter is computed from 2 of *eigvecl* and 2 and 3 of *eigvecD* and finally, the fourth quarter comes from 3 of *eigvecl* and 2 and 3 of *eigvecD*. This means that we need to exchange data between processors 0 and 1, and 2 and 3 in order to be able to compute the mentioned products (the same kind of computation is performed with the second set of processors).

After these computations, the eigenvectors set is now stored as is shown in Figure 5.9-f. In the case where eight processors are being used, we have arrived at the final step of the algorithm. In this step, all processors will work together to obtain the eigensystem of the original matrix. The information to compute z is now in the last row of the sub-matrix *eigvec1* stored in processor 3 and the first row of the matrix *eigvec1* of processor 4. This data needs to be transferred to the other processors. The eigenvectors will be computed as in Figure 5.9-f and again a product of eigenvectors matrices is needed as in Figure 5.10-c. In this figure we chose to represent only the portions stored in the first four processors to indicate that these four processors need to exchange information among themselves in order to be able to compute the first half of *eigvec*. For the first eighth of *eigvec*, part 0 of *eigvec1* is needed, along with the first four parts of *eigvecD* (first half). The second, third and fourth parts of *eigvec* also need the same upper half of *eigvecD* and the corresponding part of *eigvec1*. The fifth part of *eigvec* is obtained from part 4 of *eigvec1* and all the bottom half of *eigvecD*. The sixth, seventh and eighth parts of *eigvec* also need the bottom half of *eigvecD* as well as the corresponding parts of *eigvec1* (5, 6 and 7 respectively). This means that data from four processors need to be exchanged; processors 0 to 3 and 4 to 7 must communicate among themselves.

In general, when computing the matrix *eigvecD* in groups of rows, half the number of processors need to communicate among themselves to be able to compute the product *eigvec1* $\times$ *eigvecD* in the last step of the algorithm. However, if computing *eigvecD* column by column, all processors need to exchange information. After this final product of matrices, the algorithm is completed.

5.3.4 Communication in each step and related routines

As remarked earlier, the amount of communication needed increases as the algorithm progresses. This happens because as the method is structured, in the first step, the processors work in twos, in the second step groups of four processors work together on a sub-problem and in the third step, eight processors need to interchange information. The pattern goes on, the number of processors that need to work together doubles at each additional step.

Communication is needed in several parts of the algorithm, once partial results are obtained. Thus after the first sets of eigenvalues are computed using the QL algorithm, the

Figure 5.11: Communicating d and z; μ; partial.norm

values computed by one processor need to be "exchanged" with the ones obtained by its neighbour. The partial vectors z are also exchanged at this stage, which means that two vectors of dimension at most $\lfloor n/P \rfloor + 1$ each, will be sent and received. Figure 5.11 shows the pattern of communication for this first step using four processors, but the same kind of data transmission occurs in the other rows of the grid, if we are using a greater number of processors. (If just one row or a group of rows appears in the figures, this means that the same pattern is repeated in the other rows or groups of rows.)

After receiving the two vectors from its neighbour, the elements of d are ordered (and those of z are sorted accordingly) and the deflation routine is applied. These two routines are executed on the same data in both processors (the ones which exchanged information). The routine for finding the eigenvalues is called, and this time, though working on the same data (the ordered d and z) each processor computes its corresponding set of roots. At this stage, communication is again needed and the computed μ_i's are exchanged between these neighbouring processors. This is necessary, because we are computing the eigenvectors matrix by rows, and each processor must know all the eigenvalues for that step. The partial eigenvectors are computed, and as each processor "knows" just part of each eigenvector, it can compute just part of the sums necessary to compute the norms for normalizing these vectors. The partial sums are needed in each processor in order to compute the norm itself. Each processor computes the sums of squares of the elements of each column to which it has access. After this, partial sums are exchanged (as is shown in Fig. 5.11), and each processor computes the square root of the sum of these partial sums. At this step there is no need to communicate the partial eigenvectors of the matrix $D + \rho zz^t$ (because we are using just two processors for each sub-problem and we need the eigenvectors within one processor in this case – half the number of processors– to form the "final" eigenvectors matrix for this

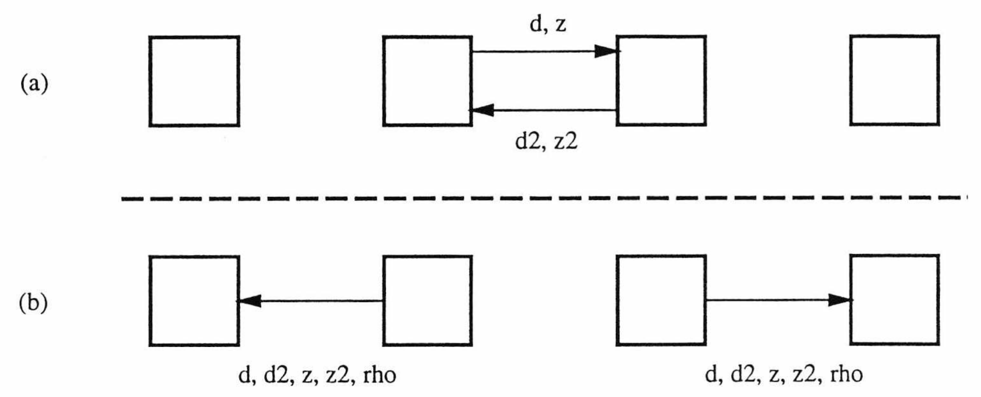

Figure 5.12: Communicating d and z in the second step

sub-problem –see previous section). The (partial) product of the two eigenvectors matrices (the one computed by the QL – *eigvec1* – and the one originated by the D & C – *eigvecD*) is formed in each processor, and this will generate the new z vectors.

We are now in what we call the second step. The partial vectors z will be present now in processors 1 and 2 (and 5 and 6, 9 and 10, etc.) – recall that the partial z's come from the last and first rows of the previous eigenvector matrices and as these are partially stored in each processor, the last row of one of those matrices will be in processor 1 and the first row of the other one will be in processor 2. These partial z's must be sent to the other processors in the same row, which do not have access to this information. This process needs two-stages of communication to be completed, because neither processor 1 nor any other processor in the same column has direct access to processor 3 (nor processor 2 to processor 0). Figure 5.12 shows the pattern of communication in this case. Recall that the vectors d and $d2$ here are the eigenvalues computed in the previous step, merged with the elements that were deflated before, which are eigenvalues themselves, to obtain a new ordered set.

When we refer to processor 0, 1, 2 or 3 in this second step, corresponding considerations apply to any processor located in the same column (but in different rows). In this step, the processors work in groups of four, all of them located in the same row, so all the communication needed here takes place within the row. Besides the transfers of data

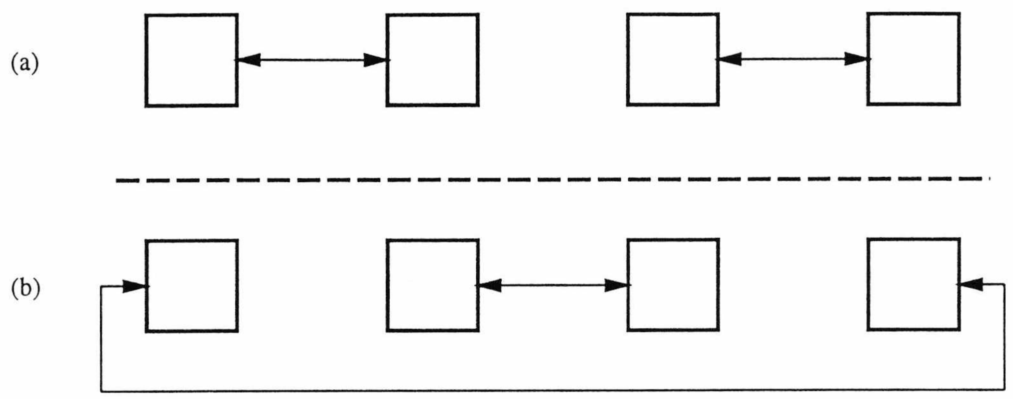

Figure 5.13: Communicating μ and $partial.norm$ in the second step

already mentioned when we described the first step, we also need to communicate the partial eigenvector matrices computed by each processor (*eigvecD*) which was not needed in the first step. However, as mentioned earlier, this exchange of information only needs to be done between pairs of adjacent processors, so that processors 0 and 1, 2 and 3 and so on, have to communicate. The way these partial matrices are transferred is shown in part (a) of Figure 5.13.

In this second step the method of communicating the partial vectors μ and the *partial.norms* is not the same as that used for transferring d and z, as was the case in the first step. This happens because the computation of ρ and z is performed only on the processors that have access to the information needed to compute these quantities, which are then transferred to the other processors. In the case of μ and *partial.norms* all the required information is initially available to all the processors; each processor has partial results which need to be communicated to the other processors involved in that stage of communication. The pattern for the communication of μ is shown in Figure 5.13, which represents the two stages required.

The third step requires three stages to communicate the vectors d and z and the scalar quantity ρ. The fourth step (which is first needed when using a minimum of 16 processors) requires four stages for this kind of communication. The pattern for these two steps follows the diagrams of Figures 5.14 and 5.15.

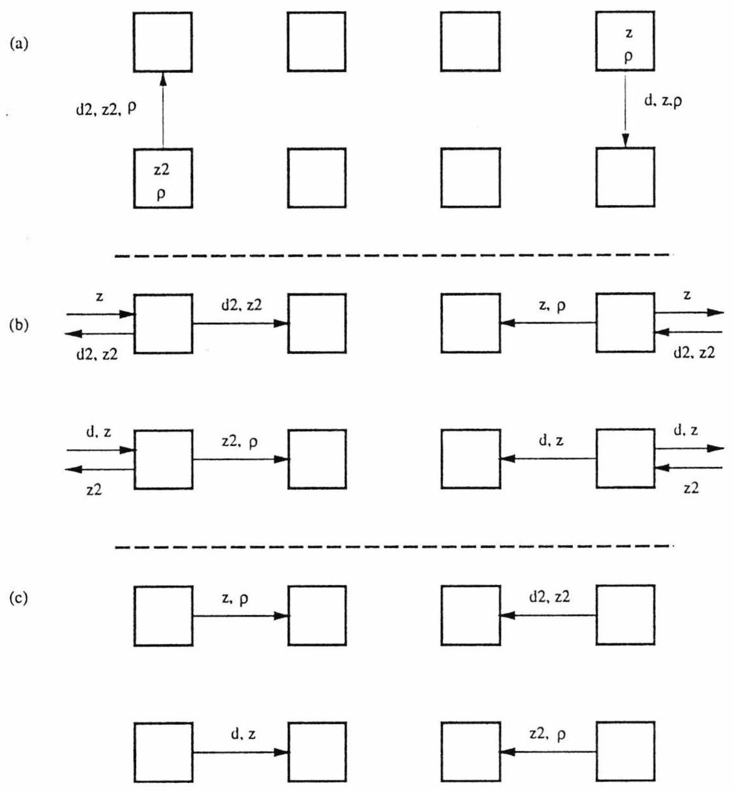

Figure 5.14: Communicating d,z and ρ in the third step

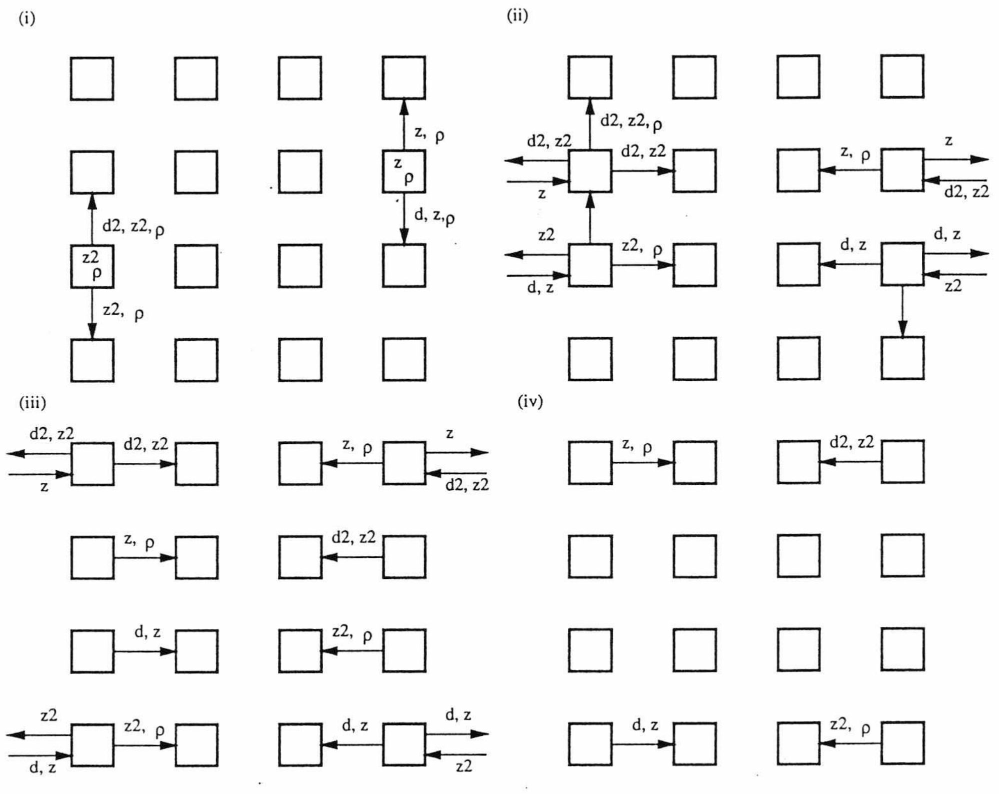

Figure 5.15: Communicating d,z and ρ in the fourth step

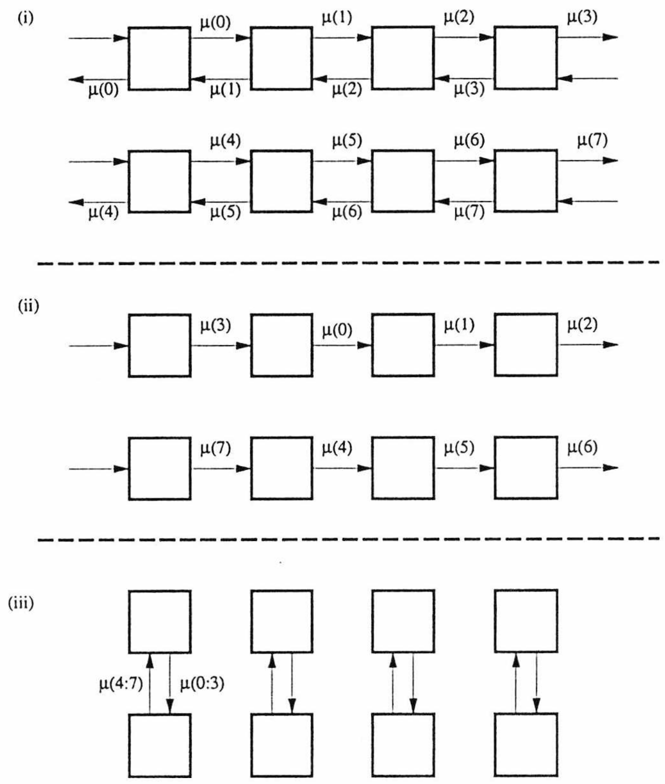

Figure 5.16: Communicating μ,$partial.norm$ and $eigvecD$ in the third step

Observe that we do not send d and $d2$ in all the possible directions, because some of the processors already have one of these quantities and some have the other. For example, if we consider 16 processors organized in four rows, rows 0 and 1 will have vector d (which came from the merge between eigenvalues computed and assigned from the previous step) and rows 2 and 3 will have $d2$. For further steps, the means of transferring this data is the same as for the fourth step, the difference is that it will take more stages to complete the sending of the variables to the extremes of the grid and patterns of communication will be repeated successively in the different rows.

The communication of μ and *partial.norm*, is also performed differently to that for d, ρ and z. Figure 5.16 shows this transfer of data.

In the first two stages, communication is restricted within the rows. Note that in the first stage, both horizontal links of each processor are used at the same time and in both directions to send and receive data in parallel to and from their immediate neighbours. The notation $\mu(p)$ in the figure indicates the sets of values of μ computed in processor p. The vertical links are not used simultaneously because this would not reduce the number of stages needed to do all the necessary communication and the programming is simpler this way. In the second stage, all processors transfer to the processor on the right, the information they received from the processor on the left in the previous stage. At the end of this second stage, all processors in a row will have all the data from the other processors located in the same row. Each then exchanges all this information with the processor that is immediately below or above it (located in the same column). The pattern to exchange the sets of rows of the eigenvector matrix (*eigvecD*) is represented by the first and second stages of Figure 5.16.

The communication in columns When four or more rows of processors (fourth step onwards) are involved in a communication process, the transmission of data is performed in several stages until every row has received all the data originating in the other rows. First, as described above, all the information from the same row is collected in each processor in that row (see stages i and ii of Fig. 5.16). After this first "collection" each processor sends its data to its immediate neighbours above and below. Figure 5.17 illustrates the process for four rows. Initially, the processors that are in row 0 send to the processors in row 1; processors in row 1 send to the corresponding processor in row 0 and row 2; processors in row 2 send to row 1 and 3, and finally processors in row 3 send to processors in row 2. When this stage is completed, processors in row 0 have their own data and the data from corresponding processors in row 1. Processors in row 1 have information from processors in rows 0 and 2; processors in row 2 have information from processors in rows 1 and 3 and processors in row 3 have information from processors in row 2 (as well as the data originating locally). Now, each processor sends upwards (when they can) data received from the processors below them, and send downwards data received from above. Simultaneously they are receiving new data from neighbouring rows, which will be sent

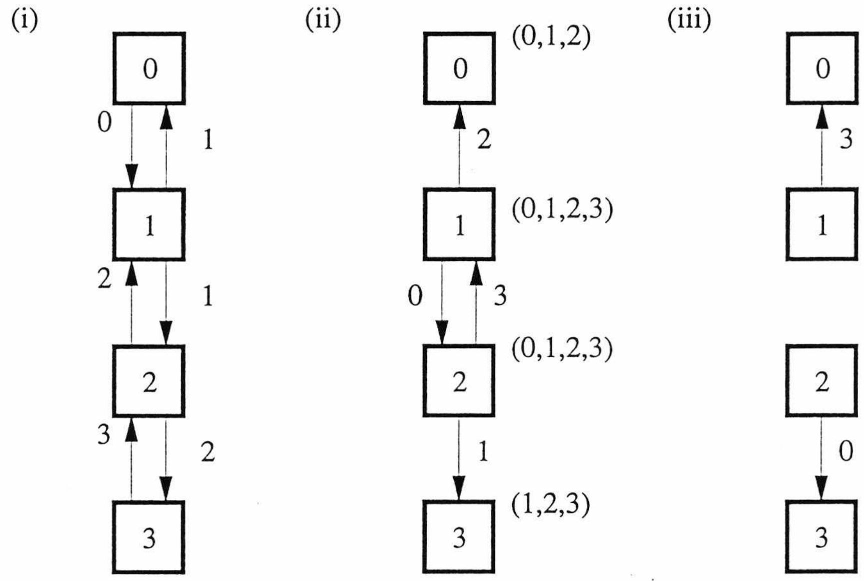

Figure 5.17: Communication in the columns

out at the next stage. This continues, until the data exchange is completed.

When more than four rows are involved in exchanging data, the mode of communicating is the same as that described above, but requires a greater number of stages to complete as data must traverse a larger array of processors. The process of communication just described is that needed for sending/receiving the different parts of the vector μ, the *partial sums* needed to normalize the eigenvectors, and the eigenvectors themselves. The only difference between the various types of data is that if four rows need to exchange data, then the transfer of eigenvectors will be needed between pairs of rows and not groups of four rows (as described in step 3). The number of processors involved in the exchange of the computed eigenvectors is always half the number of processors involved in the other types of communication in any specific step.

Another point to mention is that in this scheme of communication, where data is being sent and received at the same time, care must be taken when transferring the partial sums necessary for the computation of vector norms. We need to update the locally held sums as soon as we receive the partial sums from other rows, but clearly, we cannot do that by using the same variable for sending on the next stage, because the final sum would contain these partial sums added several times. The solution we found (other than writing a quite new routine for communication), was to use temporary variables to store what is received from above and below, and use these to send on the data.

5.4 How the modifications introduced in the original problem change the solution obtained

In our description of the method to find the eigensystem of the matrix $D + \rho z z^t$ we have assumed that the elements of D are ordered. Because the elements of D arise as the eigenvalues of T_1 and T_2, they are not ordered as they appear. Thus, we would have to do an initial sort (or merge) of these two sets of eigenvalues, so as to assemble them in order, and we must, of course, also carry out the corresponding reordering of the eigenvectors. In practice we do not modify the matrix of eigenvectors computed by the QL algorithm, because this would destroy the patterned structure it has with two separate blocks of eigenvectors. In addition, as the algorithm progresses, other factors affect the column ordering of the eigenvector matrix. This includes the rotations needed to introduce zero elements into z (in order to cope with coincident elements in D) and the permutations which send these and any other zeros to the end of the vector z. We are thus not working with $D + \rho z z^t$ but with a matrix of the form

$$\Pi_2^t J^t \Pi_1^t (D + \rho z z^t) \Pi_1 J \Pi_2$$

where Π_1 is the permutation matrix required to order the elements of D, J is the matrix accumulating the rotations arising because of coincided entries in D and Π_2 is the permutation matrix which sends the zero elements of z to the final part of the vector. Supposing X is the matrix of eigenvectors of $D + \rho z z^t$ and $\tilde{X}$ is the eigenvector matrix after it is modified by Π_1, J and Π_2, then

$$X = \Pi_1 J \Pi_2 \tilde{X}$$

and the matrix of eigenvectors of the initial matrix T is given by

$$Q\Pi_1 J\Pi_2\tilde{X}.$$

Clearly we wish to avoid performing these multiplications and the way we accomplished this is explained next.

5.4.1 How we implemented the permutations and the product of the matrices of eigenvectors

As remarked already the modifications introduced in the matrix $D + \rho zz^t$ were:

1. The permutation Π_1^t to order the elements of D.
2. The rotations (J^t) to transform the elements of z to zero when two or more of the d_i were considered to be equal.
3. The permutation (Π_2^t) to send the zero elements of z to the bottom of the vector and do the same modification in D

All these transformations are stored in vectors of adequate dimensions. Instead of computing the eigenvectors and then swapping them into their final positions, we compute them in their final order. In other words, as the elements of d and z had their positions changed by the merge used to order the elements of d (permutation Π_1^t), what we want is to use the "old" d's and z's, or the elements of these two vectors in their original position. The inverse (or transpose) of the rotations performed to zero out elements of z, also needs applying to the computed eigenvectors, in order to obtain the eigenvectors of the original $D + \rho zz^t$, i.e., the matrix $D + \rho zz^t$ before the modifications described in items 1, 2 and 3 above. The way we computed the eigenvectors will become clearer with the simple example shown below.

Consider the initial vectors d and z as given by $d = [d_1, d_2, d_3, d_4, d_5]^t$ and $z = [z_1, z_2, z_3, z_4, z_5]^t$, before any merge or rotation is performed. Recalling the formula for the eigenvectors, we need to compute

$$v = \begin{bmatrix} z_1/(d_1 - \lambda) \\ z_2/(d_2 - \lambda) \\ \vdots \\ z_i/(d_i - \lambda) \\ \vdots \\ z_n/(d_n - \lambda) \end{bmatrix}$$

All the other columns are the same, apart from changing the value of the eigenvalue λ used for each, so that in this explanation we will just consider one eigenvector.

Suppose that after the merge the vector d becomes $d = [d_1, d_3, d_4, d_2, d_5]^t$, thus the permutation Π_1, the transpose of Π_1^t, is represented by $[1, 4, 2, 3, 5]$. Suppose also that $d_1 = d_3$, so that we need to compute a rotation to zero the element of z in position 2. In this way we obtain a new vector $z = [z_1^*, z_3^*, z_4, z_2, z_5]^t$, where z_1^* and z_3^* indicate that these elements have been modified, $z_1^* = c \times z_1 - s \times z_3$ and $z_3^* = s \times z_1 + c \times z_3 = 0$. After this rotation, the newly created zero entry in z and the corresponding element of d are moved to the end of the vectors and the new arrangements for the vectors d and z are then $d = [d_1, d_4, d_2, d_5, d_3]^t$, $z = [z_1^*, z_4, z_2, z_5, z_3^*(= 0)]^t$. This permutation is represented by the array $[1, 3, 4, 5, 2]$ and its transpose is given by $[1, 5, 2, 3, 4]$ (Π_2). The program then computes

$$eigvecD[i][j] = z[ind[i]]/(d[ind[i]] - \lambda),$$

where $ind[i] = \Pi_2[\Pi_1[i]]$. Computing $ind[i]$ for $i = 1, 2, \ldots, 5$, we obtain $ind = [\Pi_2[1], \Pi_2[4], \Pi_2[2], \Pi_2[3], \Pi_2[5]] = [1, 3, 5, 2, 4]$. This means that each column j of *eigvecD* is computed as

$$eigvecD[j] = \begin{bmatrix} z[1]/(d[1]-\lambda) = z_1^*/(d_1-\lambda) \\ z[3]/(d[3]-\lambda) = z_2/(d_2-\lambda) \\ z[5]/(d[5]-\lambda) = z_3^*/(d_3-\lambda) \\ z[2]/(d[2]-\lambda) = z_4/(d_4-\lambda) \\ z[4]/(d[4]-\lambda) = z_5/(d_5-\lambda) \end{bmatrix}$$

We can see that with the exception of the first and third components, we obtain, in this way, exactly the eigenvector we needed to compute. In fact, these two components are also computed correctly, because the eigenvectors must be updated with the inverse of the rotations applied. This inverse is applied to the components that were originally changed by rotations and we compute the new $eigvecD[1][j]$ and $eigvecD[3][j]$ as

$$eigvecD[1][j] := c \times eigvecD[1][j] \tag{5.1}$$

and

$$eigvecD[3][j] := -s \times eigvecD[1][j] \tag{5.2}$$

This is because the formulae

$$eigvecD[1][j] = c \times eigvecD[1][j] + s \times eigvecD[3][j]$$

and

$$eigvecD[3][j] = -s \times eigvecD[1][j] + c \times eigvecD[3][j]$$

reduce to the above when we note that $eigvecD[3][j] = z_3^*/(d_3 - \lambda) = 0$. To see that (5.1) and (5.2) give the same results as computing these entries with the original d's and z's, recall that we chose c and s to ensure $s \times z_1 + c \times z_3 = 0$, that is $cz_3 = -sz_1$. But then $c \times z_1^* = c^2 z_1 - scz_3 = z_1$, so that $cz_1^*/(d_1 - \lambda) = z_1/(d_1 - \lambda)$. In a similar way, $-s \times z_1^*/(d_1 - \lambda) = z_3/(d_1 - \lambda)$, but as the reason we applied the rotation was because $d_1 = d_3$, then $eigvecD[3][j] = z_3/(d_3 - \lambda)$.

The rotation J is represented by two arrays c and s which store the sine and cosine values for each individual rotation and by an array which indicates the position of the first d_i for which $d_i = d_{i+1}$, after the merge is performed. In the example given above, this position would be 1. If, after the merge we also had $d[3] = d[4]$, for example, the other element of the array which holds the positions would be 3. Here we represent the components of the vector d by $d[i]$ to indicate that we are referring to the element of d that is in the i^{th} position, so as not to confuse it with the original elements of d that were denoted by $d_1, d_2, \ldots$ and could now be in different positions. With these three arrays (two with real components and one with integers), we can identify completely the computed rotations, and the integer array indicates over which elements the rotations are applied.

As another example, if the rotation J^t is given by

$$J^t = \begin{bmatrix} 1 & & & & & \\ & c_1 & -s_1 & & & \\ & s_1 & c_1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

we then have $pos = [2]$ and $c = [c_1]$, $s = [s_1]$.

There is another permutation, which is required after the computation of the eigenvectors, to merge the eigenvalues that were computed with the eigenvalues that are just assigned (and were previously elements of d). Again, these eigenvalues are physically merged, but the matrix of eigenvectors *eigvecD* is not changed. When we change the order in which the eigenvalues appear, we should also change the corresponding columns of the matrix of eigenvectors, so that the correspondence between eigenvalues and eigenvectors remains the same. In fact, when computing the product *eigvec1* $\times$ eigvecD, we select the correct columns of *eigvecD*. For instance, if column 1 should be moved to column 2, say, and in its place we should put the old column 5, then when column 1 is needed in the product, we take column 5. As an example, suppose the eigenvalues are given by

$[\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5] = [3, 5, 7, 8, 2]$ before the merge. They should clearly be re-ordered and become $[\lambda_5, \lambda_1, \lambda_2, \lambda_3, \lambda_4] = [2, 3, 5, 7, 8]$. If we represent this permutation again by an array Π_3 we would write it as $\Pi_3 = [5, 1, 2, 3, 4]$, which means that the element that is now in position 1 was in position 5 before; the element that is in position 2 was in position 1, and so on. Each entry (i, k) of the product of matrices *eigvecl* $\times$ *eigvecD* is then computed as

$$eigvec\mathrm{l}[i][k] \times eigvecD[k][\Pi_3[j]].$$

In this way, when we need what should now be column 4, we take $eigvecD[k][\Pi_3[4]] = eigvecD[k][3]$, which is the eigenvector corresponding to the eigenvalue $\lambda_3 = 7$.

During these operations, there is no need to use any extra vectors or variables to store the newly ordered eigenvectors and we do not physically transfer any entries. Certainly, we do need the integer arrays for saving the permutations involved, but these cost less to store than real arrays. There is also the overhead of having to access more arrays, which could entail some delay, but we believe the way we chose of implementing these permutations is better than having to change the positions of the elements of *eigvecD*.

Chapter 6

Results

In this chapter we present the results we obtained when we executed our implementation of the Divide-and-Conquer algorithm on our network of transputers. We also report on some experiments we have done to improve the accuracy of the solutions. In addition, we have developed a model for the complexity of the Divide-and-Conquer algorithm, when the initial matrix is partitioned into 2, 4, 8 or 16 parts. This model justifies in part (because it takes no account of the communication effects) the gains that one version of the algorithm has over another in which there is less partitioning.

6.1 Test Matrices

The Divide-and-Conquer algorithm we have implemented operates on real symmetric tridiagonal matrices. We wished to test the program with matrices which came from real world applications, but because of memory limitations on our transputer system and the unavailability of suitable matrices, we found just one example which could be used. Even this matrix was not originally tridiagonal, but it was transformed to tridiagonal form using a sequence of Householder transformations.

The matrices we used for testing the program were:

1. The $(1, 2, 1)$ matrix, i.e., the tridiagonal matrix which has all the diagonal elements equal to 2 and the off-diagonal elements equal to 1.

   This matrix was chosen partly for the purpose of comparing our implementation with those of other authors, because they tested their programs on it. Besides, this is a matrix for which there is a closed form for the eigenvalues and the eigenvectors (see [33]), and so we were able to check the results we obtained with the exact solution.

2. A matrix where the diagonal has all components equal to zero and the off-diagonal elements are given by $\sqrt{k(n-k+1)}$, $k = 1, 2, \ldots, n$ and the order of the matrix considered is $n + 1$.

   This matrix was taken from the collection of Gregory and Karney [33], and it has the nice feature that all the eigenvalues are integers. If n is odd the eigenvalues are $\pm n, \pm(n-2), \ldots, \pm 1$ and if n is even the eigenvalues are $\pm n, \pm(n-2), \ldots, \pm 2, 0$. These simple expressions for the eigenvalues makes the verification of the accuracy of our calculations very straightforward. This matrix is also used, because it differs from the $(1, 2, 1)$ matrix in that it has larger entries than that matrix. Each x_k increases as n increases. This provides a test for the behaviour of the Divide-and-Conquer algorithm when there are relatively large matrix entries.

3. The Wilkinson Matrices

   (a) The diagonal elements are $10, 9, 8, \ldots, 1, 0, 1, 2, \ldots, 9, 10$ and the off-diagonal elements are all 1.

   (b) The diagonal elements are $10, 9, \ldots, 1, 0, -1, -2, \ldots, -9, -10$ and the off-diagonal elements again, are all 1.

   These are well-known matrices created by J. H Wilkinson, to show that we can have symmetric tridiagonal matrices for which the eigenvalues are very close together, and yet the values of the off-diagonal elements are not small. The matrices introduced here have a fixed order, but larger matrices can be generated with similar properties (see [87] for details).

4. Matrices for which the diagonal elements are random numbers in the interval $(0, 1)$ and the off-diagonal elements are all 1.

5. Matrices for which the diagonal and off-diagonal elements are random numbers in the interval $(0, 1)$.

   These two types of test matrices were chosen in order to compare our results with other implementations and to test the program when the matrix entries are relatively small. They exhibit interesting behaviour, in that often a great deal of deflation is performed (see Chapter 3). These matrices also showed us the necessity of carefully computing the root μ, of the secular equation, when μ is very close to δ_{i+1} (see Section 4.6).

6. Matrices where the diagonal elements are given by $d_i = 2 + i^2$ and the off-diagonal elements are all equal to 1.

   We worked with this example, to observe the behaviour of the Divide-and-Conquer algorithm when the tridiagonal test matrix has large entries.

7. The tridiagonal matrix obtained from the so called "Platzman Problem" [62]. This is a matrix arising in a genuine application, to which we referred above, and it came to our attention from recent collection of test matrices [22]. It is a 362×362 matrix and it was tridiagonalized using the routine DSYTRD of LAPACK [2]. All the other matrices in this collection were either too large (it was necessary to limit our tests to matrices having order $\leq$ 1000), or were not real symmetric ones. A significant feature of this matrix is that the eigenvalues are very close together; earlier algorithms experienced difficulty in computing the eigenvalues accurately.

8. Matrices, obtained by combining the diagonal and off-diagonal elements from the examples given above. These were used as a means of obtaining a greater variety of

test matrices and had no other special purpose. In this way we formed, for example, a matrix with all its diagonal elements equal to 2 and with the off-diagonal elements the same as in item 2.

We will refer to these matrices by their item number, so that the matrix which has the diagonal entries all equal to 2 and the off-diagonal entries all 1, will be referred as Matrix 1, and the matrix described (as an example of possible combinations) in the last item, will be referred as Matrix 8.

6.2 Measures of performance for parallel algorithms

When assessing the performance of a parallel algorithm, we usually consider two different measures: *speed-up* and *efficiency*. The definition for the first of these differs a little from author to author, but its main objective is to give a measure of the gain in time of one algorithm over another. A rather general expression for the *speed-up*, as given by Modi [56] is

$$S_p = \frac{T_1}{T_p}$$

where T_1 is the time of the algorithm on one processor and T_p is the time of the algorithm distributed over p processors. It is in the definition of T_1 that opinions differ. Some authors take the view that the algorithm for which T_1 is being measured, should be the "best" algorithm for the problem that is being analysed. The difficulty is exactly this: What is the best possible algorithm? In our work we will use a more liberal approach for T_1. It will always be the time taken by the sequential algorithm we want to compare with our parallel algorithm; thus if we are comparing our implementation of the D&C with the QL, T_1 will be the time taken by the QL applied to the same matrix for which the D&C is executed, and so on.

We could also extend the definition of speed-up to measure the scalability of one version of the program when run on p_1 processors over the version executed on p_2 processors, so that

$$S_{p_1,p_2} = \frac{T_{p_1}}{T_{p_2}}$$

where T_{p_1} is the time the algorithm takes on p_1 processors and T_{p_2} is the time on p_2 processors, with $p_1 < p_2$.

Once the definition of speed-up is set up, the definition of efficiency offers no problems and is given by

$$E_p = \frac{S_p}{p}.$$

As its name suggests, efficiency measures how well the problem is being distributed over the p processors, that is to say how well the resources are being utilized.

We can also extend the definition of efficiency, as was done with the definition of speed-up, in the following way

$$E_{p_1,p_2} = \frac{S_{p_1,p_2}}{p_2/p_1}$$

so that we are assessing how efficiently we are using the version for p_2 processors over the one using p_1 processors. In this way we can assess how well one parallel implementation uses resources compared to the other.

Karp and Flatt [44] discuss speed-up, efficiency and other measures for performance in parallel processors. They also propose a new measure, which they claim shows other characteristics in the parallel implementation, not observed using the measures normally employed. It assesses the amount of sequential work done in the parallel implementations against the total time in one processor. They conclude that when this value is small, the parallel implementation is good. They also analyse how this ratio varies as the number of processors is increased and indicate when the behaviour can signal problems with the algorithm. Usually a fixed size problem is used when computing these measures, but their definitions can be extended to include the possibility of varying problem size. The measure proposed by Karp and Flatt is used to investigate problems arising from load imbalance and overheads imposed by the parallel implementation, and it does not allow for the superlinear effects which are a characteristic of our parallel algorithm. While load imbalance may appear in our algorithm due to the presence of deflation, the extent of which may vary from processor to processor, the computation of the eigenvectors is the dominant part of the calculation and does not suffer from load imbalance.

6.3 The three-transputers implementation

We report here the results we obtained with our three-transputers implementation. Even with this small number of processors and in spite of the structure used (see Chapter 5) we were able to get very good speed-ups and efficiencies when compared to the sequential QL algorithm. When we compared the parallel Divide-and-Conquer with its sequential version on a single processor, we still achieved efficiencies greater than 80%. In Table 6.1 we present the times obtained when computing the eigensystem of Matrix 2 using the QL and the parallel D&C algorithms. Notice that this matrix has the interesting feature that deflation is called for when its order, n, is even, while there is no deflation when n is odd. We have included two consecutive values of n in the table in order to display the difference in the times for each case.

Table 6.2 shows the speed-ups achieved when we compared QL with the sequential D&C and the speed-ups and efficiencies of the sequential D&C in relation to its parallel version on three transputers. The matrix used in this case was Matrix 1.

6.3.1 Residuals and orthogonality of the eigenvectors compared with the results obtained from the QL algorithm

The results obtained using the Divide-and-Conquer (D&C) algorithm were comparable to those obtained with QL in most of the problems. When the two sets of eigenvalues are compared, we find that they are practically the same. As for the orthogonality of the eigenvectors, the results obtained by D&C are a little inferior to those obtained by QL. This

	Time(s)			
n	QL	D&C	S_3	E_3
9	0.03	0.02	1.50	0.50
10	0.03	0.02	1.79	0.60
49	2.61	0.91	2.87	0.96
50	2.71	0.80	3.38	1.13
99	19.23	5.99	3.21	1.07
100	19.98	5.52	3.62	1.21
149	63.75	18.70	3.41	1.14
150	65.01	17.57	3.70	1.23
199	149.02	42.55	3.50	1.17
200	151.27	40.43	3.74	1.25
299	495.84	140.38	3.53	1.18
300	500.74	136.26	3.67	1.22

Table 6.1: Comparison between the QL algorithm and the parallel D&C

n	$T_{QL}/T_{D\&C}$	$S_{1,3}$	$E_{1,3}$
50	1.67	2.23	0.743
100	1.70	2.35	0.785
150	1.68	2.40	0.799
200	1.67	2.42	0.807
300	1.66	2.44	0.814

Table 6.2: QL versus sequential D&C and sequential versus parallel D&C

	$\|\|Tx - \lambda x\|\|$	
n	D&C	QL
100	4.4×10^{-15}	5.2×10^{-15}
200	1.2×10^{-14}	7.2×10^{-15}
300	6.4×10^{-15}	8.8×10^{-15}
400	1.2×10^{-14}	1.0×10^{-14}
500	8.8×10^{-15}	1.2×10^{-14}

Table 6.3: Residuals when using the D&C and the QL methods

	$\|\|Q^tQ - I\|\|$	
n	D&C	QL
100	6.7×10^{-14}	3.9×10^{-15}
200	8.2×10^{-13}	6.0×10^{-15}
300	4.3×10^{-13}	6.1×10^{-15}
400	2.2×10^{-12}	8.2×10^{-15}
500	9.5×10^{-13}	8.1×10^{-15}

Table 6.4: Orthogonality of the eigenvectors when using the D&C and the QL methods

is probably due to the means of computation of eigenvectors, which was not as accurate in this implementation as we were able to achieve later.

We assessed the norm of the residual in the same way as Dongarra and Sorensen [20]: first finding the 2-norm of each residual $Tx - \lambda x$ and then the maximum of these norms. We also used the same method to measure the orthogonality of the eigenvectors: computing the 2-norm of the columns of $Q^tQ - I$ and then the maximum of these norms, where Q is the matrix of eigenvectors. Tables 6.3 and 6.4 display the norms of the residual and the orthogonality of the eigenvectors, for Matrix 1, using the QL and the D&C algorithms.

6.4 The multiple-processor implementation

The multiple-processor implementation incorporates some features that were not present in our initial development of the sequential and three-processors version of the Divide-and-Conquer. For example, we compute the eigenvectors, using the formulae which involve μ and not λ as it was first done (see Chapters 3 and 4). This simple modification seemed to lead to better results in some cases (see next section for an example) especially in terms of the orthogonality of the eigenvectors. In fact, the computed eigenvalues did not seem to change from one method to the other, but the residuals and orthogonality improved, which confirms that we were able to compute more accurate eigenvectors by using the μ's and δ's instead of λ's and d's. One possible explanation for this improved behaviour is that effects due to bad scaling are minimised when the μ's are used, because their range is the interval $(0, 1)$, while the λ's can assume any value.

In general, the results we obtained were good and the residuals comparable to those obtained by the QL algorithm and in some cases better. There are some cases, however, for which the orthogonality between eigenvectors is worse than that recorded for the QL. It is to be noted that the QL algorithm, by its nature, tends to compute orthogonal sets of eigenvectors (see [60]), while the D&C does not share this characteristic. The cases in which the problems were found are, as to be expected, matrices for which the eigenvalues are very close together, as happens with the Platzman data (Matrix 7).

When the entries of the original matrix are relatively large, as in Matrices 2 or 7, we cannot expect to obtain residuals smaller than MACHEPS $\times$ *matrix norm*. More specifically, better results than those shown in Tables 6.5 (Matrix 8), 6.6 (Matrix 6) and 6.7 (Matrix 2), cannot be expected, since the values of the norms of the residuals are near optimal from this point of view.

6.4.1 The residuals and orthogonality in the multiple-processor version

Tables 6.5 to 6.8 display the norms of the residuals $Ax - \lambda x$ and the orthogonality measure of the eigenvectors (assessed as explained in Section 6.3.1). We also display in these tables corresponding data for the residuals and orthogonality using the QL algorithm, in order to compare the results supplied by the two algorithms. The values tabulated are based on the results when the algorithm was distributed over 16 processors, but similar results were observed when other numbers of processors were used. We also tested our programs with other matrices, but as the results obtained were very similar, the examples shown in these tables are enough to represent well the kind of performance we obtained.

6.4.2 Times, speed-ups and efficiencies

Figures 6.1 to 6.3 show the behaviour of the Divide-and-Conquer algorithm in terms of the time taken to complete the same task on various number of processors.

Figure 6.1 shows the curves for execution times against matrix size. One curve represents the times taken by the QL algorithm and the others, show the times for the Divide-and-Conquer algorithm using different numbers of processors. As already mentioned in the previous chapter, we have executed the program in 1,4,8 and 16 processors. The results shown are for Matrix 1, but other matrices gave similar curves. As can be seen from the graph, the gain (in time) of D&C over QL is significant. For the 1-processor implementation, the D&C is 1.7 times faster than QL with Matrix 1 and up to 1.2 times faster with Matrix 6. The latter is mentioned here, because its execution time for the QL algorithm is approximately half the time required by the other examples. Divide-and-Conquer also shows a reduced time for this matrix because more deflation is performed with this matrix, although the observed reductions are not as large as that for the QL.

Figure 6.2 shows the curves for speed-ups of Divide-and-Conquer compared to the QL algorithm when varying the number of processors used on a problem of fixed size. It must be noted that these are represented as continuous curves just to give a clearer idea

		D&C		QL	
matrix	n	$\|\| Ax - \lambda x \|\|$	$\|\| Q^tQ - I \|\|$	$\|\| Ax - \lambda x \|\|$	$\|\| Q^tQ - I \|\|$
1	50	1.40×10^{-15}	1.99×10^{-15}	4.30×10^{-15}	2.93×10^{-15}
	75	1.77×10^{-15}	3.93×10^{-15}	4.49×10^{-15}	3.27×10^{-15}
	100	2.24×10^{-15}	4.59×10^{-15}	5.17×10^{-15}	3.87×10^{-15}
	125	2.17×10^{-15}	7.79×10^{-15}	5.90×10^{-15}	4.87×10^{-15}
	150	2.35×10^{-15}	6.76×10^{-15}	6.34×10^{-15}	5.32×10^{-15}
	175	2.65×10^{-15}	5.34×10^{-15}	7.30×10^{-15}	4.78×10^{-15}
	200	3.14×10^{-15}	6.97×10^{-15}	7.18×10^{-15}	6.04×10^{-15}
	225	3.37×10^{-15}	7.82×10^{-15}	7.66×10^{-15}	6.44×10^{-15}
	250	3.33×10^{-15}	1.23×10^{-14}	8.03×10^{-15}	6.13×10^{-15}
	275	3.27×10^{-15}	1.59×10^{-14}	8.55×10^{-15}	6.09×10^{-15}
	300	3.39×10^{-15}	2.32×10^{-14}	8.76×10^{-15}	6.10×10^{-15}
	325	3.83×10^{-15}	4.06×10^{-14}	9.29×10^{-15}	6.52×10^{-15}
	350	3.65×10^{-15}	2.88×10^{-14}	9.15×10^{-15}	6.75×10^{-15}
	375	3.80×10^{-15}	3.77×10^{-14}	9.76×10^{-15}	7.43×10^{-15}
	400	3.81×10^{-15}	2.49×10^{-14}	1.00×10^{-14}	8.19×10^{-15}
	425	4.05×10^{-15}	2.72×10^{-14}	1.07×10^{-14}	8.11×10^{-15}
	450	4.49×10^{-15}	3.06×10^{-14}	1.07×10^{-14}	7.80×10^{-15}
	475	4.28×10^{-15}	8.97×10^{-14}	1.14×10^{-14}	8.53×10^{-15}
	500	4.32×10^{-15}	5.42×10^{-14}	1.17×10^{-14}	8.11×10^{-15}
8	50	3.16×10^{-14}	1.62×10^{-15}	1.15×10^{-13}	3.98×10^{-15}
	75	6.26×10^{-14}	1.76×10^{-15}	2.02×10^{-13}	4.29×10^{-15}
	100	8.52×10^{-14}	2.89×10^{-15}	3.02×10^{-13}	4.73×10^{-15}
	125	1.45×10^{-13}	2.39×10^{-15}	4.86×10^{-13}	4.51×10^{-15}
	150	1.75×10^{-13}	2.35×10^{-15}	7.20×10^{-13}	8.15×10^{-15}
	175	2.34×10^{-13}	3.33×10^{-15}	8.26×10^{-13}	5.50×10^{-15}
	200	2.70×10^{-13}	2.83×10^{-15}	1.07×10^{-12}	6.50×10^{-15}
	225	3.04×10^{-13}	3.12×10^{-15}	1.14×10^{-12}	6.74×10^{-15}
	250	3.94×10^{-13}	3.49×10^{-15}	1.80×10^{-12}	7.90×10^{-15}
	275	4.27×10^{-13}	4.01×10^{-15}	2.12×10^{-12}	8.34×10^{-15}
	300	5.42×10^{-13}	4.29×10^{-15}	1.69×10^{-12}	7.59×10^{-15}
	325	6.09×10^{-13}	5.21×10^{-15}	2.13×10^{-12}	8.31×10^{-15}
	350	6.26×10^{-13}	4.62×10^{-15}	2.23×10^{-12}	7.14×10^{-15}
	375	6.90×10^{-13}	4.51×10^{-15}	2.53×10^{-12}	1.00×10^{-14}
	400	7.34×10^{-13}	1.04×10^{-14}	3.28×10^{-12}	9.02×10^{-15}
	425	7.99×10^{-13}	9.11×10^{-15}	3.33×10^{-12}	8.49×10^{-15}
	450	8.57×10^{-13}	6.82×10^{-15}	3.42×10^{-12}	9.85×10^{-15}
	475	9.77×10^{-13}	6.97×10^{-15}	3.28×10^{-12}	9.69×10^{-15}
	500	1.07×10^{-12}	6.38×10^{-15}	4.60×10^{-12}	1.12×10^{-14}

Table 6.5: Residuals and orthogonality of D&C and QL.

		D&C		QL	
matrix	n	$\|\| Ax - \lambda x \|\|$	$\|\| Q^tQ - I \|\|$	$\|\| Ax - \lambda x \|\|$	$\|\| Q^tQ - I \|\|$
6	50	1.57×10^{-12}	1.77×10^{-15}	1.81×10^{-12}	3.10×10^{-15}
	75	3.41×10^{-12}	2.44×10^{-15}	5.45×10^{-12}	1.26×10^{-14}
	100	2.88×10^{-12}	2.44×10^{-15}	1.63×10^{-11}	2.26×10^{-14}
	125	6.91×10^{-12}	2.66×10^{-15}	2.36×10^{-11}	3.53×10^{-14}
	150	2.05×10^{-11}	2.88×10^{-15}	5.09×10^{-11}	3.95×10^{-14}
	175	1.36×10^{-11}	3.10×10^{-15}	4.36×10^{-11}	4.37×10^{-14}
	200	3.18×10^{-11}	5.99×10^{-15}	4.36×10^{-11}	5.06×10^{-14}
	225	3.05×10^{-11}	5.10×10^{-15}	1.01×10^{-10}	7.90×10^{-14}
	250	3.38×10^{-11}	7.10×10^{-15}	9.45×10^{-11}	7.77×10^{-14}
	275	7.54×10^{-11}	7.54×10^{-15}	3.20×10^{-10}	7.28×10^{-14}
	300	4.28×10^{-11}	8.65×10^{-15}	1.74×10^{-10}	7.72×10^{-14}
	325	7.44×10^{-11}	8.65×10^{-15}	2.47×10^{-10}	7.54×10^{-14}
	350	7.27×10^{-11}	1.06×10^{-14}	2.61×10^{-10}	7.83×10^{-14}
	375	1.28×10^{-10}	1.15×10^{-14}	3.63×10^{-10}	8.97×10^{-14}
	400	1.40×10^{-10}	1.24×10^{-14}	7.27×10^{-10}	9.59×10^{-14}
	425	1.70×10^{-10}	1.57×10^{-14}	6.98×10^{-10}	1.05×10^{-13}
	450	2.21×10^{-10}	1.33×10^{-14}	5.52×10^{-10}	1.10×10^{-13}
	475	2.00×10^{-10}	1.44×10^{-14}	8.44×10^{-10}	1.16×10^{-13}
	500	2.75×10^{-10}	1.66×10^{-14}	6.69×10^{-10}	1.25×10^{-13}
3b	21	1.63×10^{-15}	1.01×10^{-15}	3.34×10^{-15}	2.88×10^{-15}
3a	21	1.85×10^{-15}	1.56×10^{-15}	4.98×10^{-15}	1.91×10^{-15}
7	362	1.07×10^{-15}	4.00×10^{-9}	3.22×10^{-15}	2.99×10^{-14}

Table 6.6: Residuals and orthognality of D&C and QL.

		D&C		QL	
matrix	n	$\|\| Ax - \lambda x \|\|$	$\|\| Q^tQ - I \|\|$	$\|\| Ax - \lambda x \|\|$	$\|\| Q^tQ - I \|\|$
2	50	3.07×10^{-14}	1.37×10^{-15}	1.14×10^{-13}	3.15×10^{-15}
	75	5.75×10^{-14}	1.81×10^{-15}	3.28×10^{-13}	5.25×10^{-15}
	100	9.61×10^{-14}	2.74×10^{-15}	3.58×10^{-13}	4.94×10^{-15}
	125	1.48×10^{-13}	2.78×10^{-15}	4.76×10^{-13}	5.39×10^{-15}
	150	1.67×10^{-13}	2.61×10^{-15}	7.74×10^{-13}	5.76×10^{-15}
	175	2.69×10^{-13}	3.51×10^{-15}	7.68×10^{-13}	5.68×10^{-15}
	200	2.49×10^{-13}	3.74×10^{-15}	1.09×10^{-12}	5.73×10^{-15}
	225	3.05×10^{-13}	4.56×10^{-15}	1.22×10^{-12}	8.05×10^{-15}
	250	3.73×10^{-13}	3.79×10^{-15}	1.28×10^{-12}	8.46×10^{-15}
	275	4.39×10^{-13}	5.86×10^{-15}	1.69×10^{-12}	7.65×10^{-15}
	300	4.96×10^{-13}	5.89×10^{-15}	1.68×10^{-12}	1.07×10^{-14}
	325	5.60×10^{-13}	4.65×10^{-15}	2.32×10^{-12}	8.77×10^{-15}
	350	5.48×10^{-13}	6.27×10^{-15}	2.18×10^{-12}	9.16×10^{-15}
	375	7.13×10^{-13}	9.42×10^{-15}	2.44×10^{-12}	8.41×10^{-15}
	400	8.68×10^{-13}	4.34×10^{-15}	2.68×10^{-12}	8.74×10^{-15}
	425	8.51×10^{-13}	5.24×10^{-15}	2.87×10^{-12}	8.30×10^{-15}
	450	8.39×10^{-13}	6.44×10^{-15}	3.07×10^{-12}	1.10×10^{-14}
	475	9.44×10^{-13}	8.48×10^{-15}	3.55×10^{-12}	1.10×10^{-14}
	500	1.15×10^{-12}	7.86×10^{-15}	3.50×10^{-12}	9.14×10^{-15}

Table 6.7: Residuals and orthogonality of D&C and QL.

		D&C		QL	
matrix	n	$\|\| Ax - \lambda x \|\|$	$\|\| Q^tQ - I \|\|$	$\|\| Ax - \lambda x \|\|$	$\|\| Q^tQ - I \|\|$
4	50	2.27×10^{-15}	1.34×10^{-15}	3.88×10^{-15}	3.63×10^{-15}
	75	1.69×10^{-15}	2.34×10^{-15}	4.90×10^{-15}	3.42×10^{-15}
	100	2.27×10^{-15}	3.73×10^{-15}	6.99×10^{-15}	6.53×10^{-15}
	125	2.41×10^{-15}	3.57×10^{-15}	7.36×10^{-15}	4.99×10^{-15}
	150	3.35×10^{-15}	8.55×10^{-15}	6.64×10^{-15}	2.60×10^{-14}
	175	3.09×10^{-15}	1.13×10^{-14}	8.65×10^{-15}	1.10×10^{-14}
	200	3.88×10^{-15}	1.00×10^{-14}	9.46×10^{-15}	1.48×10^{-14}
	225	4.13×10^{-15}	6.15×10^{-15}	9.21×10^{-15}	2.24×10^{-14}
	250	3.58×10^{-15}	1.29×10^{-14}	1.53×10^{-14}	2.90×10^{-14}
	275	3.85×10^{-15}	8.62×10^{-15}	1.12×10^{-14}	2.92×10^{-14}
	300	4.42×10^{-15}	1.01×10^{-14}	1.02×10^{-14}	2.68×10^{-14}
	325	4.98×10^{-15}	1.34×10^{-14}	1.09×10^{-14}	3.93×10^{-14}
	350	4.91×10^{-15}	9.73×10^{-15}	2.15×10^{-14}	2.64×10^{-14}
	375	5.61×10^{-15}	1.96×10^{-14}	1.21×10^{-14}	2.53×10^{-14}
	400	5.51×10^{-15}	4.65×10^{-14}	1.46×10^{-14}	2.39×10^{-14}
	425	5.19×10^{-15}	1.14×10^{-13}	1.24×10^{-14}	4.81×10^{-14}
	450	5.72×10^{-15}	1.27×10^{-14}	1.55×10^{-14}	3.16×10^{-14}
	475	5.92×10^{-15}	5.97×10^{-14}	1.47×10^{-14}	2.97×10^{-14}
	500	5.92×10^{-15}	9.89×10^{-15}	1.52×10^{-14}	8.98×10^{-14}
5	50	8.69×10^{-16}	2.72×10^{-15}	4.57×10^{-15}	6.51×10^{-15}
	75	1.42×10^{-15}	2.27×10^{-15}	4.03×10^{-15}	1.38×10^{-14}
	100	1.84×10^{-15}	3.12×10^{-14}	4.31×10^{-15}	2.53×10^{-14}
	125	2.14×10^{-15}	2.79×10^{-15}	5.22×10^{-15}	1.20×10^{-14}
	150	4.84×10^{-15}	6.11×10^{-15}	6.76×10^{-15}	1.95×10^{-14}
	175	3.60×10^{-15}	4.26×10^{-15}	6.07×10^{-15}	2.45×10^{-14}
	200	4.19×10^{-15}	5.03×10^{-15}	6.86×10^{-15}	2.36×10^{-14}
	225	2.84×10^{-15}	7.57×10^{-15}	9.29×10^{-15}	2.63×10^{-14}
	250	3.77×10^{-15}	9.57×10^{-15}	9.67×10^{-15}	2.35×10^{-14}
	275	4.15×10^{-15}	3.34×10^{-14}	7.56×10^{-15}	2.40×10^{-14}
	300	4.94×10^{-15}	2.39×10^{-14}	8.42×10^{-15}	2.09×10^{-14}
	325	4.94×10^{-15}	7.37×10^{-15}	1.14×10^{-14}	4.11×10^{-14}
	350	4.12×10^{-15}	1.36×10^{-14}	1.41×10^{-14}	5.38×10^{-14}
	375	4.47×10^{-15}	8.29×10^{-15}	1.64×10^{-14}	5.43×10^{-14}
	400	4.53×10^{-15}	1.09×10^{-14}	1.17×10^{-14}	8.04×10^{-14}
	425	5.68×10^{-15}	1.44×10^{-14}	1.34×10^{-14}	6.58×10^{-14}
	450	5.25×10^{-15}	2.92×10^{-14}	1.02×10^{-14}	6.16×10^{-14}
	475	5.08×10^{-15}	1.53×10^{-14}	1.07×10^{-14}	5.95×10^{-14}
	500	5.73×10^{-15}	1.53×10^{-14}	1.10×10^{-14}	8.55×10^{-14}

Table 6.8: Residuals and orthogonality of D&C and QL.

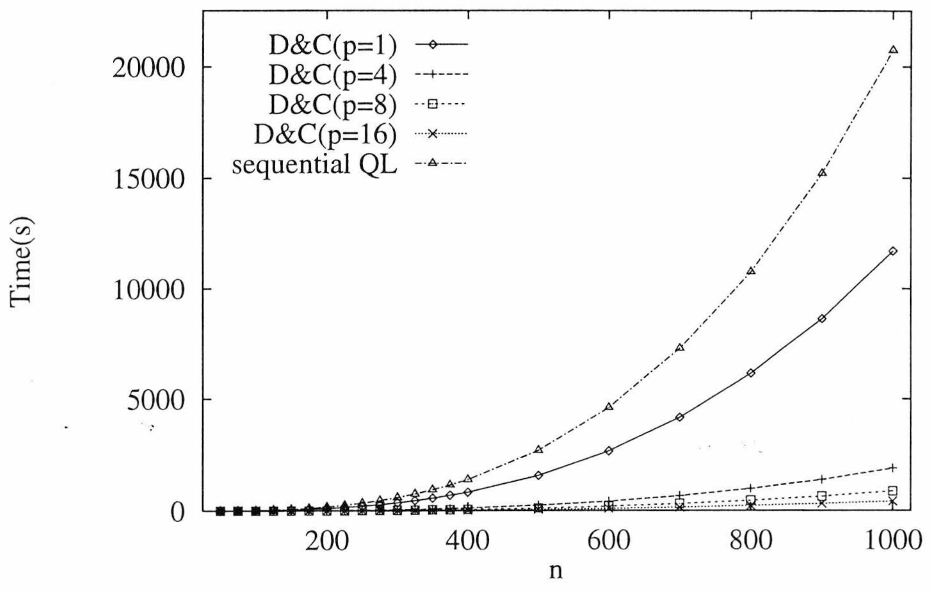

Figure 6.1: Execution times of D&C and QL implementations

of how the speed-up varies, even though the number of processors used is discrete. The number of processors for which the program was executed, is marked on each curve, in the corresponding position, with the symbol assigned for each different matrix size. Tests were performed for other sizes, but as their curves were similar to those shown here, they are, for clarity, not represented explicitly. The lines $S_p = p, 2p$ and $3p$ are also represented, to make it easier to compare the speed-ups achieved with the number of processors used. It can be seen that the speed-ups are always greater than the number of processors involved in the computation of Divide-and-Conquer, and for $n = 200$, 400 and 1000 the speed-up exceed $2p$ and even reach $3p$ for the case of $n = 1000$. This means, that if we take into account simply the times both algorithms take, Divide-and-Conquer has an advantage over QL even when executed in sequential mode. Cuppen [17] says that the gain obtained by Divide-and-Conquer is due to deflation, but we think that this is not the sole reason and we attempt to justify that in Section 6.7.

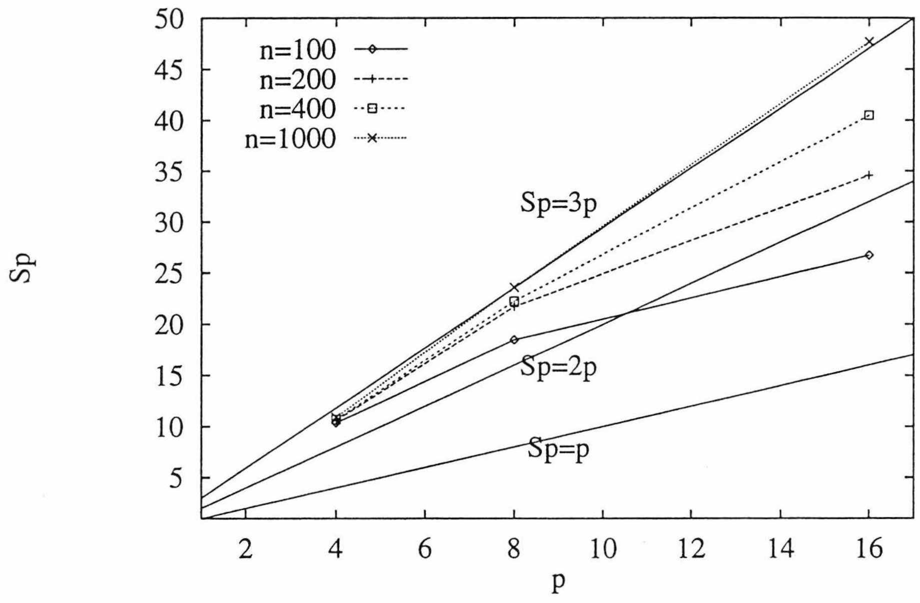

Figure 6.2: speed-up of the parallel D&C implementation with respect to QL

Figure 6.3 shows the speed-ups curves of the parallel versions of the Divide-and-Conquer algorithm in relation to the sequential version. Again, the range of matrices sizes is being restricted to give a clearer picture. The graphs demonstrate that the Divide-and-Conquer algorithm has very good scalability when we increase the number of processors, i.e., the speed-ups continue to increase as p increases, when an adequate matrix size is used. This means that we will probably get worse efficiencies if we use 32 or 64 processors to solve a problem of order $n = 100$, but we will obtain good values for $n = 200$, 400 or 1000 if we use 32 processors. The curve for $n = 200$ is increasing less than the curves for $n = 400$ and 1000, which make us believe that if were to increase the number of processors further, the efficiencies values would eventually decrease, on matrices of small order. From this analysis, if we had access to a greater number of processors with more memory, we feel confident we would have been able to solve problems with $n > 1000$ very efficiently.

In these graphs, the curves for $S_p = p, 2p$ and $3p$ are also represented to make comparisons easier. It can be seen from them that we always obtained efficiencies greater than 1.

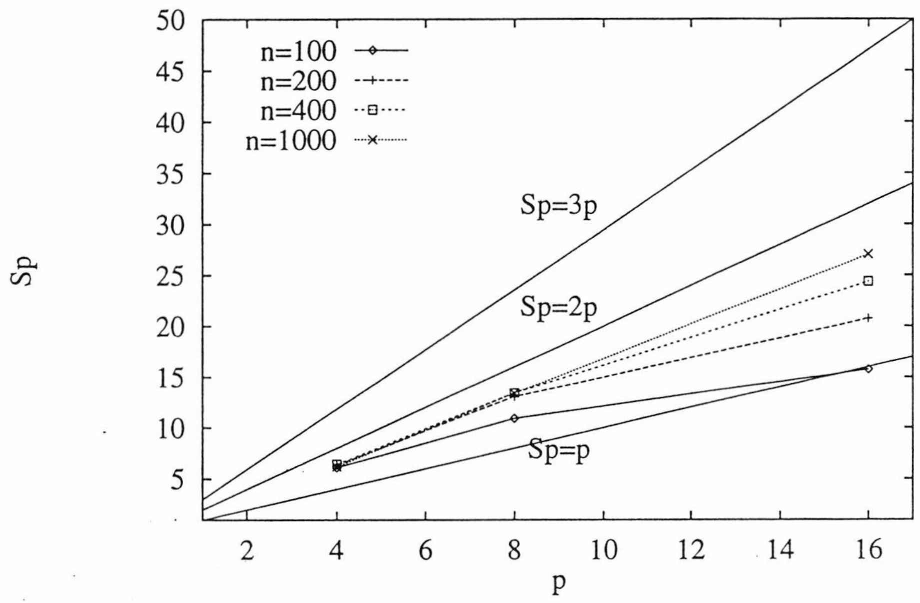

Figure 6.3: speed-up of the parallel D&C implementation

One possible explanation for this fact is that when we divide the problem into more parts, to be executed in different processors, we are really working with sub-problems, which have different execution times not adding up to the time spent on the whole matrix. As we will explain in more detail in Section 6.7, the parts of the algorithm which take the most time are the initial eigensystem computations done by the QL algorithm and the products of eigenvector matrices. When the sizes of the problems for which these computations are performed, are halved, say, the complexity of these computations is reduced significantly (see Section 6.7).

Table 6.9 shows some examples of the times taken by the Divide-and-Conquer when using 4, 8 or 16 processors and Table 6.10 the relations between them. The results shown in these tables were obtained with Matrix 6.

Table 6.10 shows that when we increase the number of processors we obtain efficiencies close to one. The behaviour of the efficiencies in this case is a little erratic and we cannot really ascertain if the trend is to increase or decrease with increasing n. This is probably

	Time(s)		
n	$P = 4$	$P = 8$	$P = 16$
100	1.93	0.89	0.50
200	14.29	6.90	3.34
300	47.22	22.04	10.99
400	110.79	53.26	27.20
500	215.75	101.55	51.62
600	373.22	179.64	86.37
700	590.90	280.93	136.88
800	879.97	424.00	209.42

Table 6.9: Times of the parallel Divide-and-Conquer for 4,8 and 16 processors

n	$E_{4,8}$	$E_{4,16}$	$E_{8,16}$
100	1.08	0.96	0.88
200	1.03	1.07	1.03
300	1.07	1.07	1.00
400	1.04	1.02	0.98
500	1.06	1.04	0.98
600	1.04	1.08	1.04
700	1.05	1.08	1.03
800	1.04	1.05	1.01

Table 6.10: Efficiencies of the parallel Divide-and-Conquer

due to variation in the sub-problems being solved when n and the number of partitions is changed, both of which affect the amount of deflation performed. What can be seen, and is natural to expect, is that as we increase the number of processors, the order of the problem we must solve on them, must also be increased if we wish to maintain a comparable level of efficiency.

6.5 Comparison with other implementations

In this section we present further results concerning times, speed-ups and efficiencies for various sorts and sizes of matrices, in order to compare them with results reported for other implementations of the Divide-and-Conquer method. We only consider implementations for symmetric tridiagonal matrices employing a version of Cuppen's algorithm [17]. These implementations are those developed by Dongarra and Sorensen [20], which uses a shared-memory machine, and Ipsen and Jessup [40], which uses a hypercube with 32 processors. Even though, in the latter case, we have not tested our program on as many processors, our implementation still compares favourably.

Table 6.11 shows the times taken by the D&C on 4 processors and speed-ups in relation to the QL algorithm for Matrix 1 and for Matrix 5. Table 6.12 shows the similar data for parallel Divide-and-Conquer in 8 processors.

Our results seem to be better or at least as good for most cases when comparing with the two implementations previously mentioned. In [20], the authors report on a speed-up of 15.2 in relation to the QL for a matrix of order $n = 150$ in an Alliant FX/8, a machine with 8 processors, while we obtained a speed-up of 21.25 for a problem of the same size and using the same number of processors. For the same size, but in a Cray X-MP-4, which has 4 processors, they obtained a speed-up of 4.5 while we achieved 10.54. As they did not say for which matrix they obtained these values, the comparison is possibly not appropriate, but as we obtained similar results for other matrices tested, we believe they are representative. For Matrix 1 the results we obtained were slightly better than

	Matrix 1			Matrix 5		
n	Time(s) D&C	S_4	E_4	Time(s) D&C	S_4	E_4
100	2.45	10.35	2.59	2.52	9.79	2.45
150	7.66	10.54	2.64	7.62	10.53	2.63
200	17.37	10.60	2.65	17.47	10.69	2.67
300	55.87	10.71	2.68	56.07	11.02	2.76
400	129.06	10.84	2.71	129.46	11.20	2.80
500	247.75	10.91	2.73	248.95	11.36	2.84

Table 6.11: Times, speed-ups and efficiencies of the 4 processor– Divide-and-Conquer algorithm against QL

	Matrix 1			Matrix 5		
n	Time(s) D&C	S_8	E_8	Time(s) D&C	S_8	E_8
100	1.38	18.38	2.30	1.53	16.12	2.02
150	3.80	21.25	2.66	3.71	21.63	2.70
200	8.47	21.74	2.72	8.44	22.12	2.77
300	26.86	22.27	2.78	26.45	23.36	2.92
400	60.07	23.30	2.91	60.34	24.03	3.00
500	117.45	23.02	2.88	113.75	24.75	3.09

Table 6.12: Times, speed-ups and efficiencies of the 8 processor– Divide-and-Conquer algorithm against QL

those of Dongarra and Sorensen, as they report speed-ups of 9.4, 15.4, 17.7 and 20.0 for $n = 100, 200, 300$ and 400 respectively, using 8 processors. They also report speed-ups of 60.7 for $n = 400$ in 8 processors for a random matrix, but do not specify which matrix this is, so we cannot compare our results with theirs in a meaningful way. Our results are comparable with theirs on Matrix 5 for $n = 100$ and $n = 200$, where they obtained speed-ups of 12.1 and 19.5, whereas we obtained 16.12 and 22.12 for the same sizes with 8 processors. However there is a larger difference in values for $n = 300$, where they report a speed-up of 38.8 compared with 23.36 in our implementation. They give as a reason for their figure, the amount of deflation that takes place with this matrix. We did not find any similar values to this and believe that this could be due to data communication overheads in our case, which are not a factor in their implementation which is for shared-memory machines.

The other implementation with which we can compare ours, is that of Ipsen and Jessup. Their results are for the hypercube for which data communication is necessary, thus the machines used for both sets of results are more alike, though there are still many differences. The data they supply is simply the execution times, which in general seem to be much higher than ours. They use a 5-cube which has 32 processors and using Matrix 1, obtained for $n = 100$, a time of 10.5 seconds whereas our implementation took 1.38 seconds in 8 processors and 2.45 seconds in 4 processors. For $n = 512$ the execution time of their implementation was 611.8 seconds, and for $n = 500$, we found a time of 117.45 seconds in 8 processors and 247.75 in 4 processors. It must be observed that one possible reason for these big differences is the fact that the hypercube used by Ipsen and Jessup – Intel's iPSC-1 – does not allow communication with more than one node at the same time, whereas transputers do. Also, the processors in the iPSC-1 are slower than transputers. We cannot compare the scalability of our implementation with theirs as they only report results for a fixed number, 32, of processors. To assess speed-ups they compare the sequential implementation of Dongarra and Sorensen (routine SESUPD) with theirs on a 32 processor hypercube. As we did not have access to their program, we can only compare with our own version for one processor. As we have previously seen (Figure 6.3), the speed-ups we found when comparing the parallel version of Divide-and-Conquer with the sequential one, were in general greater than the number of processors used. Ipsen and Jessup report for a matrix with all the off-diagonal terms equal to 1 and the i^{th} diagonal element equal to $i \times 10^{-6}$, speed-ups smaller than the number of processors, but close to that figure. For the $(1, 2, 1)$ matrix they got an efficiency of about 50%. Again, we cannot compare our results directly with theirs because Dongarra and Sorensen's SESUPD may be better optimised than our D&C, and so, take less time, which would give smaller speed-ups than those we obtained with our sequential version.

6.6 Some experiments and their results

We performed a variety of experiments in order to explore ideas for enhancing the accuracy of the solutions and in this section we report on some of these investigations and the results found.

We took up an idea of Bunch et al. [13] and Dongarra and Sorensen [20] in which we update the approximation for the root μ by adding corrections to the previous approximant instead of re-computing μ. If t_0 is the first approximation and $\tau_0, \tau_1, \ldots, \tau_{k-1}$ are the corrections to this initial approximation such that $t_1 = t_0 + \tau_0$, $t_2 = t_1 + \tau_1 = t_0 + \tau_0 + \tau_1$ and so on, are the next approximations, we have that μ is then given by $\mu = t_0 + \tau_0 + \tau_1 + \ldots + \tau_{k-1}$. In this way, they suggest that in place of calculating the difference $\delta_j - \mu$ (needed for the computation of the eigenvectors), one should compute $(((\delta_j - t_0) - \tau_0) - \tau_1) - \ldots - \tau_{k-1}$. This alternative formulation has the advantage of computing $\delta_j - \mu$ more accurately, because smaller and smaller quantities were being taken from δ_j, after each iteration. However, this modification did not seem to furnish any real improvement in the results obtained. We tested the modifications in the computation of the eigenvalue alone, and also with the modification made to δ (which affects the computation of the eigenvectors). In both cases, the results we obtained were almost the same as previously, at times the norms measuring residuals and orthogonality were a little worse and sometimes they were a little better. As an example, with Matrix 2 the residual we found for $n = 150$ was 3.8×10^{-13} and our measure of orthogonality of the eigenvectors was 1.5×10^{-14} before the modifications were done, while these were 4.0×10^{-13} and 1.8×10^{-14} respectively when they were included. For $n = 249$ the residual norms remained the same and the measure of orthogonality improved marginally from 1.49×10^{-14} to 9.6×10^{-15}. We tested these modifications on other matrices and for different values of n and found the results to be broadly similar. We conclude that it is not necessary to complicate matters by using this second way of computing the differences $\delta_j - \mu$. The tests we report were performed simply with the sequential version of the Divide-and-Conquer, because fewer changes in the program were needed in this case: had we obtained improved results we would have modified the parallel versions. However, these modifications would cause problems with the parallel versions, in that, as we compute the eigenvectors by blocks of rows and not columns, we would either have had to communicate the full sequence of τ_k for each μ_i, or we would have had to store all the modified δ_j's which would require another matrix for storage.

6.6.1 The computation of the last root

Our early implementations of sequential Divide-and-Conquer computed the last root μ_n by means of the formula $\mu_n = 1 - \sum_{j=1}^{n-1} \mu_j$. Although this approach seemed to work well, we obtained a very good improvement, in some cases, for the norm which measures the orthogonality of eigenvectors, when we changed the way we computed this last root, to use same routine as for the other roots. For example, in the case of Matrix 1, the residuals continued to be $O(10^{-15})$, but the orthogonality was improved by two orders of magnitude, going from 8.26×10^{-13} to 9.43×10^{-15} when $n = 249$. Both the orthogonality and residual results for Matrix 6 were also greatly improved. For example, for $n = 200$ the residual norm decreased from 3.14×10^{-8} to 6.55×10^{-11}; the orthogonality measure improved from 4.7×10^{-11} to 2.7×10^{-14}. For $n = 300$ the difference in the residual norms was smaller, an improvement from 9.1×10^{-9} to 2×10^{-10}, but the orthogonality was much enhanced, the measure changing from 1×10^{-11} to 4×10^{-14}. Similar changes were observed with other matrices; sometimes small or no changes to the residual norms, but improvements in the orthogonality of approximately one order of magnitude, e.g. from $O(10^{-14})$ to $O(10^{-15})$.

6.6.2 Computing μ instead of λ

Our first versions of the Divide-and-Conquer algorithm on 1 and 3 processors, in spite of computing μ_i, did not use the values obtained to compute the eigenvectors directly, but instead used the eigenvalues, i.e., $\lambda_i = d_i + \rho\mu_i$. With this approach we found results such as 6.6×10^{-12} for the residual norm for Matrix 2, when $n = 149$. On changing this computation to calculate $\delta_j - \mu_i$ in place of $d_j - \lambda_i$, we found that the results improved markedly, with a residual norm of 4.3×10^{-13}, and the orthogonality measure improved from 1.23×10^{-13} to 3.6×10^{-15}. For other sizes and other matrices similar differences occurred. Certainly our experiments verified that the proposal of Bunch et al. [13] should be used, not only to permit the calculation of all the eigenvectors, but also to achieve better accuracy.

6.6.3 Computing ν

The computation of ν instead of μ is necessary in some cases, as was shown in Sections 4.6. More precisely, if μ_i happens to coincide with δ_{i+1}, we need an alternative method in order to be able to compute the quantity $z_{i+1}/(\delta_{i+1} - \mu_i)$. This is solved by computing ν_i (see Section 4.6) in place of μ_i. In early implementations, we recomputed the root only when the problem $\mu_i = \delta_{i+1}$ occurred, in order to avoid computational problems with divisions by zero in the computation of the eigenvectors (we refer to this as approach 1). At a later stage, we employed the routine to compute ν whenever $f(\delta_{i+1}/2) < 0$, in which case the root μ_i would lie in the half-interval $(\delta_{i+1}/2, \delta_{i+1})$ and might possibly be close to δ_{i+1} (to which

	$\|\|Tx - \lambda x\|\|$	
n	Approach 1	Approach 2
112	9.1×10^{-10}	2.8×10^{-15}
163	8.3×10^{-10}	2.5×10^{-15}
200	2.0×10^{-9}	4.2×10^{-15}

Table 6.13: Differences in the residuals when computing ν by approaches 1 and 2

	$\|\|Q^TQ - I\|\|$	
n	Approach 1	Approach 2
112	2.0×10^{-9}	3.3×10^{-15}
163	2.0×10^{-9}	8.0×10^{-15}
200	3.0×10^{-9}	5.0×10^{-15}

Table 6.14: Differences in the orthogonality when computing ν by approaches 1 and 2

we refer as approach 2 or the general approach). With this more general formulation, the results we obtained were in the main similar to those obtained with approach 1, but in the case of Matrix 5 for which we were obtaining rather poor residuals, the results were greatly improved. Using the first approach we found a residual of 6.6×10^{-12} for $n = 57$, while with the second approach we obtained the much improved residual of 1.32×10^{-15}. The measure of orthogonality also improved from 5.2×10^{-11} to 1.9×10^{-15}. Other illustrative results, for various values of n, obtained for this matrix using the two approaches are shown in Tables 6.13 and 6.14.

The differences shown in these Tables, seem to be due to the more accurate calculation of eigenvectors when using the second approach. We came to this conclusion by comparing the eigenvalues computed in the two ways with those computed by the QL algorithm. The differences between the two sets of eigenvalues was the same in both cases (either approach 1 or 2). We obtained, for example, when $n = 200$, a maximum difference of 5.9×10^{-15} between the eigenvalues computed by the QL and those computed by the D&C. This difference did not vary when we used the second approach, which leads us to believe that the computation of the eigenvectors gives rise to the differences in accuracy noticed. This seems to show, that even though the computed eigenvalues do not differ, the values found for the μ's (via the ν's) are different and probably more accurate.

6.6.4 Changing the initial tolerance

As already mentioned, one issue that is still not completely resolved is the criteria to decide when to deflate. At first we believed we could relax the condition permitting deflation and that we could thus reduce further the size of the problem to be solved. However, we have not been able to prove that this is the case. We have performed some tests to attempt to verify this hypothesis. The tests seem to indicate that in certain cases we can relax the criteria for deflation, but for others, the results degenerate if we increase the permitted tolerance. Recall that the criteria used to ascertain if a pair (d_i, e_i) is considered a good approximation for an eigenpair is

$$|\rho z_j| \leq tol = macheps\,\eta(max(|d_1|, |d_n|) + |\rho|).$$

Sometimes the use of $macheps$, the machine-epsilon, in the expression above can be too stringent a requirement. Our experiments used the above inequality, but instead of $macheps$ we used initial tolerances with values of $10^{-15}, 10^{-14}, 10^{-12}, 10^{-10}, \ldots, 10^{-4}$. We found that two situations can occur when we change this tolerance: either the results remain basically the same, indicating that in this cases there is no problem in relaxing the criterion; or the results get worse at each increase in the tolerance. However, the indication that we can relax the criterion may simply mean that there is no increase in the amount of deflation performed (or in the number of z's that are considered to be zero), until we reach a tolerance of about 10^{-4}. This is the case for Matrix 1 or for Matrix 2. With these two matrices, we can employ tolerances of 10^{-15} or 10^{-8} and obtain the same residuals. For Matrix 2 with $n = 128$, the residual norms got slightly better when we increased the tolerance from 10^{-14} to 10^{-12}. These were respectively 1.71×10^{-13} and 1.28×10^{-13}. This last residual remained the same with a tolerance of 10^{-6}, deteriorating when the tolerance was set to be 10^{-4}. The measure for orthogonality of the eigenvectors did not change much. The execution times recorded seemed to decrease a little, but could not be considered significant.

The second kind of situation that can arise happens with Matrix 5 and Matrix 6. For these two matrices a large amount of deflation takes place, and this increases as we increase the initial tolerance (or relax the criteria for considering z_i to be zero). For $n = 96$ and with an initial tolerance of 10^{-15} we obtained a residual of 6.12×10^{-12}; using 10^{-14} as the initial tolerance, we obtained a residual of 8.46×10^{-11} and thereafter the results became worse. For tolerances of 10^{-12}, 10^{-10}, 10^{-8} the residuals were respectively 1.0×10^{-9}, 3.0×10^{-7} and 8.0×10^{-5}. The orthogonality, however, remained basically the same for all initial tolerances tested. Similar results occur with Matrix 5. The orthogonality cannot be considered very important here, because as we increase the tolerance, we are setting more eigenvectors equal to columns of the identity matrix, which should enhance orthogonality.

In the case of Matrix 6, with an initial tolerance of 10^{-15}, there is extensive deflation. For example, when $n = 200$, all but 8 eigenvalues are determined via deflation. However, if we increase the tolerance to 10^{-12}, then the number of fully computed eigenvalues reduces to 5, but more importantly, the norms of the residuals degrade from 3.2×10^{-11} to 3.1×10^{-8}. This matrix clearly shows, that there are cases where the criterion for deflation cannot be relaxed.

6.7 A model for complexity of the Divide-and-Conquer algorithm

We have constructed a model for the complexity of the Divide-and-Conquer algorithm, to justify the gains obtained when more processors (and more partitions) are used. It also shows why the sequential Divide-and-Conquer may perform better (faster) than the QR/QL algorithm. This model is not a definitive answer to the question of the super-linear speed-ups we obtained, in that the delays resulting from communications taking place are not considered in this model. It is important to realise that we cannot disregard the time taken for the transmission of data, because it is a significant fraction of the total execution time. However, this is not an easy factor to incorporate and we will leave it for future investigation.

In our model we consider just the terms of higher order in the counting of operations, and focus on multiplications rather than other arithmetic operations, because they appear much more frequently and in the parts of the algorithms critical to the determination of the execution time. Recalling that we use the routine *tql2* adapted from Wilkinson [88], and our objective is to compare the performance of QL with Divide-and-Conquer, we need to account for the rotations performed in order to obtain the eigenvectors. In the discussion that follows suppose that the matrix whose eigenvalues and eigenvectors we want to compute has order n. Then, the complexity of QL is $O(4n^3)$ (see Parlett [60] or check procedure *imtql2* in [88] to verify that if approximately 2 iterations —it is 1.7, on average, according to Parlett— are needed for each eigenvalue, then the main operations count is given by $8n[(n-1)+(n-2)+\cdots+2+1] = 4n^2(n-1) = O(4n^3)$). Sequential Divide-and-Conquer uses QL to compute the eigensystems of two matrices. In order to make our analysis clearer, we assume that both matrices have the same order, namely $n/2$. This gives an operation count, for this phase, of $2 \times 4(n/2)^3 = n^3$. The other part of the algorithm which is a major contribution to the computation time is forming the product of matrices of eigenvectors in the final part of the program (or of each step). Again this product is performed twice, with two different pairs of matrices. The dimensions of the corresponding matrices in each pair are the same and are respectively $n/2 \times n/2$ and $n/2 \times n$, which gives a total of $n/2 \times n \times n/2 = n^3/4$ operations for each matrix product. The other parts of the algorithm all have orders of complexity smaller than n^3, unless a great number of iterations is necessary in the root finding algorithm, which could then play an important role in the determination of computation time. In general, this did not seem to happen, and if we consider that 5 iterations are needed to find one root, then at most $27n^2$ multiplications are required to find all the roots (note that there are some preliminary operations that must be performed to compute each root). Assuming that this number of iterations is kept small, the complexity of the sequential Divide-and-Conquer algorithm is then given by

$$n^3 + \frac{2n^3}{4} = \frac{3n^3}{2}.$$

This means, that QL performs

$$\frac{4n^3}{3n^3/2} = \frac{8}{3} \cong 2.7$$

times more arithmetic operations than sequential Divide-and-Conquer (when we consider the initial problem split in two parts). This result shows that if we want to compute all the eigenvalues and eigenvectors of a tridiagonal matrix T, then the Divide-and-Conquer competes very favourably with QL in terms of execution time, at least in theory. This also shows that the gain of D&C over QL should not be considered as surprising as Dongarra and Sorensen [20] found. Furthermore, this gain is further increased when deflations take place, but deflation is not the only reason as Cuppen [17] and Dongarra and Sorensen [20] state.

If we review the practical results we found, we can see that in spite of achieving good speed-ups when comparing D&C with QL, we did not find as good gains as the theoretical result indicates. This is probably due to the simplicity of our model which does not take into account all the features of the algorithm.

6.7.1 The model for the parallel version

For the parallel versions of the Divide-and-Conquer algorithm, we consider a general model, for p processors, and then specialise it for each case. This model takes into account the varying operation counts arising during the different steps of the algorithm, which are directly related to the number of processors in use.

Letting p denote the number of processors and assuming the dimension n of the original matrix is exactly divisible by p, the size of each initial sub-problem is n/p. The counting of operations is then

- Step 1:
  - QR/QL algorithm:
  $$4\left(\frac{n}{p}\right)^3 \text{ multiplications}$$
  - products of matrices of eigenvectors:
  $$\frac{n}{p} \times \frac{2n}{p} \times \frac{n}{p} = \frac{2n^3}{p^3}$$
  - total number of multiplications:
  $$6\frac{n^3}{p^3}$$
- Step 2: problems now have dimension $2n/p$ (note that the QL is not used from this step onwards).
  - products of matrices of eigenvectors :
  $$\frac{n}{p} \times \frac{4n}{p} \times \frac{2n}{p} = \frac{8n^3}{p^3}$$
- Step 3: likewise:
  $$\frac{n}{p} \times \frac{8n}{p} \times \frac{4n}{p} = \frac{32n^3}{p^3}$$
- Step 4:
  $$\frac{n}{p} \times \frac{16n}{p} \times \frac{8n}{p} = 128\frac{n^3}{p^3}$$

Now, using these expressions for $p = 4, 8$ and 16, we have

- 4 processors ($p = 4$) The operations count is
  $$\frac{6n^3}{4^3} + \frac{8n^3}{4^3} = 14\frac{n^3}{4^3} = \frac{7n^3}{2^5}$$
- 8 processors ($p = 8$): The operations count is
  $$\frac{6n^3}{8^3} + \frac{8n^3}{8^3} + \frac{32n^3}{8^3} = 46\frac{n^3}{8^3} = \frac{23n^3}{2^8}$$

- 16 processors ($p = 16$): The operations count is

$$\frac{6n^3}{16^3} + \frac{8n^3}{16^3} + \frac{32n^3}{16^3} + \frac{128n^3}{16^3} = 174\frac{n^3}{16^3} = \frac{87n^3}{2^{11}}$$

Recalling that the operations count for sequential D&C is $3n^3/2$, we can forecast "speed-ups" using the expressions found above for each array of processors we used. Representing the complexity order by O we have

$$\begin{aligned}
\frac{O(seq)}{O(p=4)} &= \frac{3/2}{7/2^5} = 4 \times \frac{12}{7} \cong 4 \times 1.7 = 6.8 \\
\frac{O(seq)}{O(p=8)} &= \frac{3/2}{23/2^8} \cong 8 \times 2.1 = 16.8 \\
\frac{O(seq)}{O(p=16)} &= \frac{3/2}{87/2^{11}} \cong 16 \times 2.2 = 35.2 \\
\frac{O(p=4)}{O(p=8)} &= \frac{7/2^5}{23/2^8} = 2 \times \frac{28}{23} \cong 2 \times 1.2 = 2.4 \\
\frac{O(p=4)}{O(p=16)} &= \frac{7/2^5}{87/2^{11}} \cong 4 \times 1.3 = 5.2 \\
\frac{O(p=8)}{O(p=16)} &= \frac{23/2^8}{87/2^{11}} \cong 2 \times 1.1 = 2.2
\end{aligned}$$

The computations above explain why we can obtain super-linear speed-ups or efficiencies greater than 1 when using a parallel implementation of the Divide-and-Conquer algorithm. The model above does not include the cost of communications required by each implementation which can slow down the program and make the preceding values smaller in practice. In fact, communication really plays an important role in the total execution time taken by the algorithm and is responsible at least partially for the lower speed-ups we get. It must be noted that as the steps increase, the number of processors involved in communication also increase, this means that the issue of communication becomes more important as more processors are used.

A formal analysis or a model for communication is left for future work, where we would wish to determine whether the communication overheads would significantly alter the numbers obtained above.

Chapter 7

Conclusion

We have studied in this thesis the solution of the symmetric tridiagonal eigenproblem using Cuppen's Divide-and-Conquer method. In this study, new proofs for the convergence of the method for finding the roots of the secular equation have been devised. In our view, some of the aspects covered in these proofs facilitate the understanding of the mechanics of the method.

We have also described a method to compute a root of the secular equation when that root is almost coincident with the right endpoint of the interval within which it is located. This new formulation permits the computation of the corresponding eigenvector, which was not possible with the previous means of computing the roots. The convergence proofs employed for the first method were shown also to apply to the new alternative formulation. We then used in practice a combination of the two methods, which resorts to the original method when the root is known to be in the first half of the interval, but adopts the new method otherwise. With this arrangement we were able to obtain improved numerical results in previously troublesome cases.

A parallel algorithm based on this scheme was developed and implemented on a network of transputers. In this implementation, the processors are organized in a grid, using the four connections available on the transputers, taking advantage of both directions to communicate data. We believe that the algorithm should be fairly easy to port to other parallel machines available like the Intel Paragon XP/S, the TMC CM-5 and the Cray T3D, where such grid topologies are used.

The numerical results we obtained with our implementations were in general good as far as the accuracy of the solutions (measured by the norms of the residuals) and the orthogonality of the eigenvectors is concerned. However, we did encounter an example for which this orthogonality is not as good as one might hope. Recent papers, on which we report in the next section, present alternative ways of computing the eigenvectors, which seem to resolve the problem.

The Divide-and-Conquer algorithm is faster than the sequential QR/QL algorithm even when a sequential version of the former is used. The parallel implementations of Divide-and-Conquer naturally offer even greater gains. The Divide-and-Conquer algorithm has a good scalability when we increase the number of processors employed, exhibiting super-linear speed-ups when the order of the matrix is sufficiently large in relation to the number of processors.

We have devised a simple model of the complexity of the sequential and parallel Divide-and-Conquer. This model justifies the gains of Divide-and-Conquer over QR/QL and partly accounts for the superlinear speed-ups of Divide-and-Conquer mentioned above. It must be noted though that this is a partial justification because we have not included communication issues in our model.

In summary, we have obtained good results with our implementations of the Divide-and-Conquer method, not only with respect to the accuracy of the solutions, but also with respect to the scalability of the parallel versions. However, there are still some improvements that might possibly be made, and we will discuss some ideas for doing this in the next section.

7.1 Possible improvements on Divide-and-Conquer

In the last two years, new ideas have arisen for improvements with the Divide-and-Conquer method, and in the solution of the secular equation. For the latter, Li [46] has developed new ways of interpolating the functions ψ and φ (see expressions (4.2) and (4.3)). However, his aim is to speed up the algorithm for the computation of the roots of the secular equation reducing the number of iterations needed to find each one. As this is a minor part of the activity throughout the whole Divide-and-Conquer algorithm, we need not worry unduly about making this aspect faster. The author compares his new methods amongst themselves, and with that proposed by Bunch et al. [13], but only gives results relating to the number of iterations taken by each algorithm to find each root and does not comment at all on the comparative accuracy of the solutions. It is interesting to note that one of the new methods of interpolation described corresponds to our way of computing ν, but they either use the original means of computing μ or that involving ν, but not a combination as we do. Additionally, they use the secular equation formulated so as to compute λ, and they do not scale it, using the δ_j's, to compute μ, which we know can enhance the accuracy of the roots.

The problem of the lack of orthogonality for eigenvectors corresponding to very close eigenvalues has been investigated by Sorensen and Tang [71]. To address this problem they proposed the use of a simulated extended precision arithmetic in some critical parts of the algorithm. However, this simulation depends on which machine it is implemented and, thus, may not be portable. Accordingly, they propose as well another method, which modifies the manner of performing some calculations. However, their second method does not solve the problem of lack of orthogonality in all cases. The calculations that they are most concerned with, is the computation of the differences $d_j - \lambda_i$, because it was shown in [20] that if these quantities are accurately computed, then we obtain numerical orthogonality. Using their extended precision algorithm they were able to obtain enhanced measures for orthogonality in some cases where the original algorithm gave poor results. The small values for the residuals were similar using both algorithms. Their second adaptation, which reformulates the secular equation, was also shown to be better than the original Divide-and-Conquer in some cases, but there were also some examples in which its performance was much worse, not only with respect to orthogonality, but also in relation to the residuals obtained.

Gu and Eisenstat [35] propose another way of computing the eigenvectors of matrix $D + \rho z z^t$ which does not need to use the simulated extended precision suggested by Sorensen and Tang. Their idea is based on the fact that the computed eigenvalues are exact eigenvalues for another matrix $D + \rho \tilde{z}\tilde{z}^t$. They observe that even though the computed eigenvalues are close to the true ones, the eigenvectors computed using these approximate eigenvalues can be very different from the true ones and orthogonality may be lost. They propose computing the eigenvalues in the normal way (for example, using the method described in Chapter 4) and then with these values calculate the $\tilde{z}_i$'s. These are then used, together with the computed eigenvalues, in the usual formula (3.14) for the eigenvectors, and they observe that these computed eigenvalues and eigenvectors form an eigendecomposition for the original matrix $D + \rho z z^t$, provided the $\tilde{z}_i$'s are close to the z_i's, and for this, the computed λ_i's should be good approximations for the exact eigenvalues. All of this can be achieved as long as the stopping criterion for finding the eigenvalues is appropriate, i. e., they suggest the iterations are stopped when

$$|f(\mu)| \leq \eta n(1 + |\psi(\mu)| + |\varphi(\mu)|),$$

where η is a small multiple of the machine-precision and n is the order of the matrix. Note that the method for finding the eigenvalues is not specified here, so that this condition may be unnecessary if we use the method described in Chapter 4, for which quadratic monotonic convergence is assured. Gu and Eisenstat compare their new means for computing the eigenvectors with the original one given by Bunch et al. [13], and, also Dongarra and Sorensen [20], and find that their algorithm gives rise to improved measures for the orthogonality of the eigenvectors, while the size of the residuals is similar, or only slightly worse in some cases.

Gu and Eisenstat have also reported an alternative way of partitioning the initial tridiagonal matrix for which the eigensystem is sought (see [34]). Instead of decomposing T as in Cuppen's Divide-and-Conquer method, they propose an orthogonal transformation of T into an arrowhead matrix, i.e. a matrix of the form

$$H = \begin{pmatrix} \alpha & z^t \\ z & D \end{pmatrix},$$

where α is a real scalar, z is a column vector and D is a diagonal matrix. The eigensystem of this arrowhead matrix is then computed, in order to obtain the eigensystem of the original matrix T. As in the case of the isolation of a rank-one matrix in all of the foregoing, a secular equation is formed and the roots of this equation are the eigenvalues of the arrowhead matrix and, of course, of T. The eigenvectors are also computed by means of a formula similar to the one we have used previously. Again, as the computed eigenvalues are not exact roots of the secular equation, the computed eigenvectors can be very different from the true ones, and a similar approach to that described in [35] is used to obtain orthogonal eigenvectors. Here the computed eigenvalues are exact eigenvalues of a matrix

$$\hat{H} = \begin{pmatrix} \hat{\alpha} & \hat{z}^t \\ \hat{z} & D \end{pmatrix},$$

where the values for $\hat{z}_i$ and $\hat{\alpha}$ are calculated from the values of D and from the computed eigenvalues. Again, to guarantee that the $\hat{z}_i$'s are close to the z_i's and $\hat{\alpha}$ is close to α, which in turn makes the eigenvectors and eigenvalues of $\hat{H}$ good approximations for the true eigenvectors and eigenvalues of H, it is enough to ensure that the computed eigenvalues are close to the true eigenvalues. For this we must again take care about which criterion is used to judge when to stop the iterative calculation of the roots. These authors compare their new divide-and-conquer strategy with Cuppen's Divide-and-Conquer, with the QR/QL algorithm and with bisection/inverse iteration. They find that their algorithm is faster than Cuppen's and much faster than QR/QL. They attribute the advantage in execution time of their algorithm over Cuppen's to the way they perform deflation. The arrowhead divide-and-conquer is also, in general, faster than bisection, but there are cases in which bisection is slightly better. The residuals are in general about the same, for the authors's Divide-and-Conquer, Cuppen's algorithm and QR/QL, but they found some cases where their algorithm delivers more accurate results (by one or two decimal places) than the others. In Gu and Eisenstat's tests, bisection either had the same or enhanced accuracy than the other three methods. As for orthogonality, neither of the test matrices, used by the authors, display a serious orthogonality problem when employing Cuppen's method: for matrices for which the measure for orthogonality was $O(10^{-14})$ using Cuppen's, it was $O(10^{-15})$ with the arrowhead decomposition.

Other improvements on the Divide-and-Conquer method are suggested by Gill and Tadmor [29] who modify the algorithm so that it exhibits a complexity of $O(n^2)$ (instead of $O(n^3)$), and by Bini and Pan [8], who use evaluations of the characteristic polynomial and bisection to compute the eigenvalues. Both of these have been tested on sequential machines only.

7.2 Different approaches to Cuppen's Divide-and-Conquer

There are some other authors who also use divide-and-conquer techniques to solve the eigenvalue problem. The difference, with these investigations, is that the authors solve other kinds of eigenproblems (i.e., they do not employ their methods for finding the eigensystem of a symmetric tridiagonal matrix), or they solve the sub-problems by other means not involving the calculation of roots of a rational equation. In this section we mention some of these investigations, but we do not describe them in detail, only stating in general terms what kind of problems they apply to and what results have been obtained.

Krishnakumar and Morf [45] describe a divide-and-conquer scheme similar to that of Cuppen. They partition an initial matrix by means of rank-one decompositions, but then use Sturm sequences to compute the eigenvalues of the modified matrices. They develop formulae for the characteristic polynomials of the partial matrices, but instead of using polynomials of degree k to compute the next polynomial of degree $k+1$, they double the order of the next polynomial at each stage (i.e. use polynomials with degree k to compute the next with degree $2k$). Their algorithm is meant to be a parallel one, but the authors have only tested it in sequential mode. They do not give details on how the algorithm would perform in a parallel machine or how the different tasks would be partitioned among the processors. The tree structure of the algorithm is described, and how two sub-problems are combined to obtain the solution of a larger one. They compare the solutions obtained by their algorithm with those presented in [88], and usually find that both solutions agree with each other.

Gragg and Reichel [32] and Ammar et al. [1] compute the eigensystem of an unitary upper Hessenberg matrix H also by a divide-and-conquer approach. They use the fact that any such matrix with nonnegative subdiagonal elements can be written as a product of $(n - 1)$ Givens matrices and a diagonal matrix. In some applications this decomposition arises naturally, and Ammar et al.'s program uses the parameters of this decomposition: they do not compute the elements of H explicitly. Their divide-and-conquer technique partitions the initial matrix (by partitioning the product of Givens matrices) into problems with order $(n - 2)$ and 2, following formulas that can be found in [1]. The matrix with order $(n - 2)$ is again recursively partitioned in the same way, until all sub-matrices have order 2 (if n is even) or one of them has order 1 (if n is odd). The eigensystems of these 2×2 matrices can be found explicitly (by means of closed formulae for the eigenvalues and eigenvectors) and then the eigensystems of successively larger problems are computed using the eigensystems of the smaller matrices until we obtain the eigensystem of H (details of how this is done can be found in [32] and [1]). The authors compare their implementation with the appropriate QR/QL routines for computing the eigensystem of upper Hessenberg matrices and conclude that their algorithm exhibits comparable accuracy to that obtained with QR/QL, but is faster. However, more tests need to be done with larger matrices, as the reported results correspond only to matrices with order up to 50. One distinct advantage of their algorithm is that they are able to compute all the eigenvalues, and first components alone of the eigenvectors, which is all that the application they intend requires. This does not seem to be possible with QR/QL.

Arbenz [3] uses divide-and-conquer techniques to compute the eigensystem of a symmetric band matrix. He computes a matrix A_0 which is a part of the initial band-matrix A after an orthogonal transformation is applied to it. This matrix A_0 is a direct sum of two full square matrices A_1 and A_2. As in other divide-and-conquer techniques, the knowledge of the eigensystems of these two components is used for the solution of the eigenproblem of A. For further details, see [3]. The algorithm was tested on a shared-memory machine with 6 processors and compared with the EISPACK routines `bandr` paired with `tql2` and `tred2` paired with `tql2`. The first of each pair of these pairs of routines, reduces the original matrix to tridiagonal form, either by exploiting the fact that it is a band matrix, or by treating it as an arbitrary full symmetric matrix without any special structure. This divide-and-conquer algorithm is slower than the EISPACK routines for matrices with order up to 200, when run in sequential mode. When the order of the matrix is 300 and the semi-bandwidth is 6, the speed-ups obtained using a larger number of processors are good (greater than the number of processors). For smaller orders, the speed-ups are always smaller than the number of processors employed. The accuracy (residuals and orthogonality) is worse than that obtained by the EISPACK pairs. One of their algorithms loses 3 to 4 decimal places in the measures for residuals and orthogonality. An improved version of Arbenz's algorithm obtains better results, but these results are still not as good as those obtained by `tql2`.

Finally, there is a divide-and-conquer algorithm for solving the generalized eigenproblem: for details see Borges and Gragg [10].

7.3 Future work

As is to be observed throughout this work, there are still some questions that need to be answered and results that potentially can still be improved.

One question that remains, is of deciding what is the best criterion for determining when to do deflation without degrading the accuracy of the results. The tests we have performed indicate that this is not simply a matter of changing the initial tolerance for deciding when z is appropriately small (see Section 3.4), but should include some additional scaling besides those already being used.

Another aspect that must still be pursued is how to improve the orthogonality between eigenvectors, when the corresponding eigenvalues are very close. We have encountered problems with the Platzman matrix, and there are undoubtedly other examples where the same phenomenon occurs. One possible solution for this problem is to implement the suggestion given by Gu and Eisenstat [35] described above, which recomputes the z_j's (that appear in $D + \rho zz^t$) and uses these to compute the eigenvectors.

Also following Gu and Eisenstat [34], it would be interesting to implement their alternative scheme for decomposing the initial tridiagonal matrix and to investigate if it is really worthwhile by contrast with Cuppen's Divide-and-Conquer. We note that their results were based on execution on a sequential machine. It remains to be seen if the gains obtained by their algorithm can be extended to a parallel implementation.

Another point that must still be investigated is the use of the Divide-and-Conquer method as a way of computing the eigenvalues alone and not the corresponding eigenvectors as well, which was the way we proceeded in the present work. Our feeling is that Divide-and-Conquer should be employed only when all the eigenvalues and corresponding eigenvectors are sought, but this issue has not yet been studied. If the size of the initial sub-problems is not large, it may be advantageous to use Divide-and-Conquer to compute only the eigenvalues, but the saving over the full version of Divide-and-Conquer is only in the computation of the products of matrices of eigenvectors, because the eigenvectors of rank-one matrices are still needed in order to determine the vector z. This lowers the complexity of the algorithm to $O(n^2)$, if the initial sub-problems are sufficiently small, but then the algorithm may not be competitive with QL.

The parallel reduction of a full dense matrix to a tridiagonal one has also to be studied and incorporated into the algorithm, in order to obtain a more general algorithm fit to solve a wider range of matrix eigenvalue problems. For this we could either employ the already mentioned routine of SCALAPACK but adapt it to use on transputers, or we could employ techniques described in [19], [21] and [9].

The Divide-and-Conquer algorithm still need to be more thoroughly compared with other algorithms to ascertain which algorithm is appropriate for which problem. For this, and also to establish more definitively the performance of Divide-and-Conquer, more test matrices should be used, including perhaps those employed in Gu and Eisenstat's tests [34], in order to compare our results with theirs. The adoption of a routine to tridiagonalize a matrix A (as commented on the previous paragraph), would also allow a broader comparison between Divide-and-Conquer and other methods.

A more complete model of complexity, which includes communication issues, should be developed, to verify if the partial conclusions we obtained with our simplified model are still valid when communication is considered. But we note that as far as the model of complexity of the sequential version of D&C is concerned, the conclusions in relation to the comparison with the QR/QL algorithm hold in any case.

Finally, many more issues remain to be studied. These include alternative means of solving the sub-problems that arise in the Divide-and-Conquer partitions not using the secular equation; investigating how the Divide-and-Conquer method extends to symmetric-band matrices or other symmetric matrices which have a special structure; investigating how the method behaves when the matrix is not real. In addition, the search for alternative methods for solving the eigenproblem which are suitable for parallel implementations should continue in order to make available a greater variety of methods from which can be selected the tool most appropriate to the particular application in hand.

7.4 Final remarks

To summarise, we have studied a divide-and-conquer method for solving the symmetric eigenvalue problem for the theory of which we have presented new proofs of convergence.

The method applies to tridiagonal matrices, and is very suitable for parallel implementations, displaying good scalability when the number of processors employed is increased. Moreover, the parallel implementations have superlinear speed-ups when compared with the sequential version of the algorithm. The numerical results obtained were also good: accuracies (measured by the norms of the residuals of the solutions) of order $O(10^{-15})$ compared with a machine-precision of $O(10^{-16})$ were achieved for matrices whose entries are of moderate size (for matrices with larger entries, a similar result was found, but it must be viewed in relative terms, by taking the norm of the matrix into consideration). The orthogonality between eigenvectors was in general $0(10^{-15})$, but we have encountered an example for which this is not achieved and where the measure for orthogonality needs to be reduced. There is already available a strategy, devised by Gu and Eisenstat, which seems to work well, but still remains to be tested with our example. When this orthogonality issue is fixed, and after more testing has been done to confirm the improvement of the algorithm, we believe that the algorithm described in this thesis will be competitive with the QR/QL algorithm not only with respect to execution times, but also in respect of overall accuracy, including the orthogonality of eigenvectors in most, if not all, situations.

Bibliography

[1] G.S. Ammar, L. Reichel, and D.C. Sorensen. An implementation of a divide and conquer algorithm for the unitary eigenproblem. *ACM Transactions on Mathematical Software*, 18:292–307, 1992.

[2] 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*. SIAM, Philadelphia, 1992.

[3] P. Arbenz. Divide and conquer algorithms for the bandsymmetric eigenvalue problem. *Parallel Computing*, 18:1105–1128, 1992.

[4] S. Barnett. *Matrices – Methods and Applications*. Clarendon Press, Oxford, 1990.

[5] A. Basermann and P. Weidner. A parallel algorithm for determining all eigenvalues of large real symmetric tridiagonal matrices. *Parallel Computing*, 18:1129–1141, 1992.

[6] A. Basilievsky. *Applied Matrix Algebra in the Statistical Sciences*. North-Holland, New York, 1983.

[7] M. Berry and A. Sameh. An overview of parallel algorithms for the singular value and symmetric eigenvalue problems. *J. of Computational and Applied Mathematics*, 27:191–213, 1989.

[8] D. Bini and V. Pan. Pratical improvement of the divide-and-conquer eigenvalue algorithms. *Computing*, 48:109–123, 1992.

[9] C. H. Bischof and X. Sun. A framework for symmetric band reduction and tridiagonalization. Preprint MCS-P312-0692, Mathematics and Computer Science Division, Argonne National Laboratory, July 1992.

[10] C. F. Borges and W. B. Gragg. A parallel divide and conquer algorithm for the generalized real symmetric definite tridiagonal eigenproblem. In L. Reichel, A. Ruttan and R. S. Varga, editor, *Proceedings of the conference in numerical linear algebra and scientific computation*, pages 11–29, Berlin, 1993. Walter de Gruyter.

[11] C.A. Brebbia and A.J. Ferrante. *Computational Methods for the Solution of Engineering Problems*. Pentech Press, London, 1979.

[12] R.B. Brent and F.T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Stat. Comp.*, 6:69–84, 1985.

[13] J.R. Bunch, C.P. Nielsen, and D.C. Sorensen. Rank-one modification of the symmetric eigenproblem. *Numerische Mathematik*, 31:31–48, 1978.

[14] W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W Leland. Massively parallel methods for engineering and science problems. *Communications of the ACM*, 37(4):31–41, 1994.

[15] S. C. Chapra and R. P. Canale. *Numerical Methods for Engineers*. Mc Graw–Hill Publishing Company, New York, 1985.

[16] J. Choi, J.J. Dongarra, D.W. Walker, and R.C. Whaley. SCALAPACK Reference Manual – Parallel Factorization Routines (LU, QR and Cholesky) and Parallel Reduction Routines (HRD, BRD and TRD) – (Version 1.0BETA, December 31, 1993). Report ORNL/TM-12470, Oak Ridge National Laboratory, April 1994.

[17] J.J.M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik*, 36:177–195, 1981.

[18] G.J. Davis, R.E. Funderlic, and G.A. Geist. *A hypercube implementation of the implicit double shift QR algorithm*, pages 619–626. Hypercube Multiprocessors. SIAM, Philadelphia, 1987.

[19] J.J. Dongarra, D. C. Sorensen, and S. J. Hammarling. Block reduction of matrices to condensed form for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27:215–227, 1989.

[20] J.J. Dongarra and D.C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. Stat. Comput.*, 8(2):s139–s154, 1987.

[21] J.J. Dongarra and R.A. van de Geijn. Reduction to condensed form for the eigenvalue problem on distributed memory architectures. *Parallel Computer*, 18:973–982, 1992.

[22] I.S. Duff, R.G. Grimes, and J.G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection. Report TR/PA/92/86, CERFACS, October 1992.

[23] A. Edelman. Large dense numerical linear algebra in 1993 – the parallel computing influence. *International Journal of Supercomputer Applications*, 7(2):113–128, 1993.

[24] M.P.G. Fachin and B.J. Vowden. Parallel computation of tridiagonal eigensystems – a transputer implementation of a Divide and Conquer method. Technical Report Series UKC/IMS/P92/4a, Institute of Mathematics and Statistics, University of Kent at Canterbury, April 1992.

[25] J. Götze, S. Paul, and M. Sauer. An efficient Jacobi-like algorithm for parallel eigenvalue computation. *IEEE Transactions on Computers*, 42(9):1058–1065, 1993.

[26] K.A. Gallivan, R.J. Plemmons, and A.H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, 1990.

[27] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension.* Springer-Verlag, Berlin, 1977.

[28] G.A. Geist and G.J. Davis. Finding eigenvalues and eigenvectors of unsymmetric matrices using a distributed-memory multiprocessor. *Parallel Computing*, 13:199–209, 1990.

[29] D. Gill and E. Tadmor. An $O(N^2)$ method for computing the eigensystem of $n \times n$ symmetric tridiagonal matrices by the divide and conquer approach. *SIAM J. Sci. Stat. Comp.*, 11:161–173, 1990.

[30] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore, 1989.

[31] A.R. Gourlay and G.A. Watson. *Computational Methods for Matrix Eigenproblems*. John Wiley & Sons, London, 1973.

[32] W.B. Gragg and L. Reichel. A divide and conquer method for unitary and orthogonal eigenproblems. *Numerische Mathematik*, 57:695–718, 1990.

[33] R.T. Gregory and D.L. Karney. *A Collection of Matrices for Testing Computational Algorithms*. Wiley-Interscience (John Wiley and Sons), New York, 1969.

[34] M. Gu and S.C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. Research Report YALEU/DCS/RR-932, Dept. of Computer Science, Yale University, November 1992.

[35] M. Gu and S.C. Eisenstat. A stable and efficient algorithm for the rank-one modification of the symmetric tridiagonal eigenproblem. Research Report YALEU/DCS/RR-916, Dept. of Computer Science, Yale University, September 1992.

[36] S.J. Hammarling. *Latent Roots and Latent Vectors*. Adam Hilger, London, 1970.

[37] D. J. Harris. *Mathematics for Business, Management and Economics – a systems modelling approach*. Ellis Horwood Limited, Chichester, 1985.

[38] D. Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20(4):740–777, 1978.

[39] P. Hlawiczka. *Matrix Algebra for Electronic Engineers*. Iliffe Books, London, 1965.

[40] I.C.F. Ipsen and E.R. Jessup. Solving the symmetric tridiagonal eigenvalue problem on the hypercube. *SIAM J. Sci. Stat. Comp.*, 11:203–229, 1990.

[41] A. Iserles. Private communication.

[42] A. Jennings and J.J. McKeown. *Matrix Computation (second edition)*. J. Wiley & Sons, Chichester, 1992.

[43] T.Z. Kalamboukis. The symmetric tridiagonal eigenvalue problem on a transputer network. *Parallel Computing*, 15:101–106, 1990.

[44] A. H. Karp and H. P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, 1990.

[45] A.S. Krishnakumar and M. Morf. Eigenvalues of a symmetric tridiagonal matrix a divide-and- conquer approach. *Numerische Mathematik*, 48:349–368, 1986.

[46] R.-C. Li. Solving secular equations stably and efficiently. Research Report, Department of Mathematics, University of California at Berkeley, April 1993.

[47] T.-Y. Li and N.H. Rhee. Homotopy algorithm for symmetric eigenvalue problems. *Numerische Mathematik*, 55:265–280, 1989.

[48] T.-Y. Li, H. Zhang, and X.-H. Sun. Parallel homotopy algorithm for the symmetric tridiagonal eigenvalue problem. *SIAM J. Sci. Stat. Comp.*, 12:469–487, 1991.

[49] INMOS Limited. *Occam 2 reference manual*. Prentice-Hall, New York, 1988.

[50] S. Lo, B. Philippe, and A. Sameh. A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem. *SIAM J. Sci. Stat. Comp.*, 8:s155–s165, 1987.

[51] M. Lu and Xiangzhen Qiao. Applying parallel computer systems to solve symmetric tridiagonal eigenvalue problems. *Parallel Computing*, 18:1301–1315, 1992.

[52] A. Magid. *Applied Matrix Models*. J. Wiley & Sons, New York, 1985.

[53] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves 1: simple intersections. *ACM Transactions on Graphics*, 13(1):73–100, 1994.

[54] The MathWorks, Inc. *MATLAB for Sun Workstations*, January 1990.

[55] A. McIntosh. Private communication.

[56] J. J. Modi. *Parallel Algorithms and Matrix Computation*. Oxford University Press, Oxford, 1990.

[57] J.M. Ortega and W.G. Poole Jr. *An Introduction to Numerical Methods for Differential Equations*. Pitman, Marshfield, 1981.

[58] M.H.C. Paardekooper. A quadratically convergent parallel Jacobi process for diagonally dominant matrices with distinct eigenvalues. *J. of Computational and Applied Mathematics*, 27:3–16, 1989.

[59] M.H.C. Paardekooper. A quadratically convergent parallel Jacobi-like eigenvalue algorithm. In D.J. Evans, G.R. Joubert and F.J. Peters, editor, *Parallel Computing 89*, pages 173–242, Amsterdam, 1990. Elsevier Science Publishers.

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

[61] G. Pini. A parallel algorithm for the partial eigensolution of sparse symmetric matrices on the CRAY Y-MP. *Parallel Computing*, 17:533–561, 1991.

[62] G. W. Platzman. Normal modes of the atlantic and indian oceans. *Journal of Physical Oceanography*, 5(2):201–221, 1975.

[63] Y. Saad. *Numerical Methods for Large Eigenvalue Problems.* Manchester University Press, Minneapolis, 1991.

[64] A. Sameh. On Jacobi and Jacobi-like algorithms for a parallel computer. *Mathematics of Computation*, 25(115):579–590, 1971.

[65] A.H. Sameh and D.J. Kuck. A parallel QR algorithm for symmetric tridiagonal matrices. *IEEE Transactions on Computers*, C-26(2):147–153, 1977.

[66] T. Schreiber, P. Otto, and F. Hofmann. A new efficient parallelization strategy for the QR algorithm. *Parallel Computing*, 20:63–75, 1994.

[67] S.R. Searle. *Matrix Algebra for the Biological Sciences.* J. Wiley & Sons, New York, 1966.

[68] J.R. Shewchuk. An introduction to the Conjugate Gradient method without the agonizing pain. Report CMU-CS-94-125, School of Computing Science, Carnegie Mellon University, March 1994.

[69] G. M. Shroff. A parallel algorithm for the eigenvalues and eigenvectors of a general complex matrix. *Numerische Mathematik*, 58:779–805, 1991.

[70] C. A. B. Smith. *Biomathematics.* Charles Griffin and Company Limited, London, 1969.

[71] D.C. Sorensen and P.T.P. Tang. On the orthogonality of eigenvectors computed by divide-and- conquer techniques. *SIAM J.Numer.Anal.*, 28:1752–1775, 1991.

[72] G.W. Stewart. A parallel implementation of the QR algorithm. Technical report, Department of Computer Science and Institute for Physical Science and Technology, University of Maryland.

[73] G.W. Stewart. *Introduction to Matrix Computations.* Academic Press, Inc., Orlando, 1973.

[74] G. Strang. *Linear Algebra and its applications*. Harcourt Brace Jovanovich, San Diego, 1988.

[75] B. Sumner. Private communication.

[76] P. N. Swarztrauber. A parallel algorithm for computing the eigenvalues of a symmetric tridiagonal matrix. *Mathematics of Computation*, 60(202):651–668, 1993.

[77] P. Tervola and W. Yeung. Parallel Jacobi algorithm for matrix diagonalization on transputer networks. *Parallel Computing*, 17:155–163, 1991.

[78] M. Trott. Private communication.

[79] H.A. van der Vorst. Private communication.

[80] R.C. Ward. The QR algorithm and Hyman's method on vector computers. *Mathematics of Computation*, 30:132–142, 1976.

[81] J. Warsa. Private communication.

[82] D.S. Watkins. *Fundamentals of Matrix Computations*. John Wiley & Sons, New York, 1991.

[83] D.S. Watkins. Some perspectives on the eigenvalue problem. *SIAM Review*, 35(3):430–471, 1993.

[84] G. Watson. Private communication.

[85] L.T. Watson. Private communication.

[86] J. S. Weston and M. Clint. Two algorithms for the parallel computation of eigenvalues and eigenvectors of large symmetric matrices using the ICL DAP. *Parallel Computing*, 13:281–288, 1990.

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

[88] J.H. Wilkinson and C. Reinsch. *Handbook for Authomatic Computation, volume II, Linear Algebra.* Springer-Verlag, Berlin, 1971.