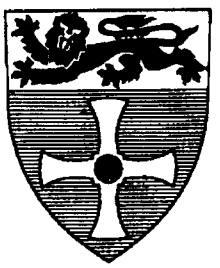
THE UNIVERSITY OF NEWCASTLE UPON TYNE
DEPARTMENT OF COMPUTING SCIENCE

UNIVERSITY OF
NEWCASTLE UPON TYNE



Parallel Algorithms for Numerical Linear Algebra on a Shared Memory Multiprocessor

by

Dogan Kaya

PhD Thesis

June 1995

Abstract

This thesis discusses a variety of parallel algorithms for linear algebra problems including the solution of the linear system of equations $Ax = b$ using QR and LU decomposition, reduction of a general matrix A to Hessenberg form, reduction of a real symmetric matrix B to tridiagonal form, and solution of the symmetric tridiagonal eigenproblem. Empirical comparisons are carried out using various different versions of the above algorithms and this is described in this thesis. We also compare three different synchronisation mechanisms when applied to the reduction to Hessenberg form problem. We implement Cuppen's method for computing both eigenvalues and eigenvectors of a real symmetric tridiagonal matrix T using both recursive and non-recursive implementations. We consider parallel implementations of these versions and also consider parallelisation of the matrix multiplication part of the algorithm. We present some numerical results illustrating an experimental evaluation of the effect of deflation on accuracy, comparison of the parallel implementations and comparison of the additional parallelisation for matrix multiplication.

A variety of algorithms are investigated which involve varying amounts of overlap between different parts of the calculation and collecting together updates as far as possible to make good use of the storage hierarchy of the shared memory multiprocessor. Algorithms using dynamic task allocation are compared with ones which do not. The results presented have been obtained using the C++ programming language, with parallel constructs provided by the Encore Parallel Threads package on a shared memory Encore Multimax (MIMD) computer. The experimental results demonstrate that dynamic task allocation can be sometimes very effective on this machine, and that very high efficiency is often obtainable with careful construction of the parallel algorithms even for relatively small matrices.

Acknowledgements

My deepest gratitude goes to my supervisor, Dr. Kenneth Wright, for his continuous encouragement throughout this research, for his excellent criticisms, superb editing and meticulous proofreading. He always made time for me, listening to my problems and provided me with abundant wisdom. I wish to thank him for everything that he taught me. Additionally, I wish to thank my thesis committee, Dr. Chris Phillips and Dr. Graham Megson, for their helpful suggestions and comments.

There are also a number of people that have helped out in one way or another. I am very grateful to all of them. There are some to whom I would like to give special thanks. My good friend Dr. Mesut Guner, who helped me during all stages of the preparation of the text. I would also like to thank Dr. M. Pakzad for reading the early draft of the thesis.

I am deeply grateful to my wife and my family at home for their consistent support and comfort. Now Bushra and Omer Faruk, time to play.

Finally, I acknowledge the financial support of the University of Firat in Turkey during my stay in the UK.

Dedication

This work is dedicated to the memory of my distinguished friend, Mehmed Kalkan, who was doing his Ph.D. in Politics at Durham University and passed away on 27 June 1992.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Parallel processing has emerged as a key enabling technology in modern computers, driven by the ever-increasing demand for higher performance, lower costs, and sustained productivity in real-life applications. A multiprocessor system provides a promising approach to high speed computing. This is a single computer system containing multiple processors which are capable of communicating and cooperating at different levels in order to solve a given problem [48]. More specifically, through some regular interprocessor network, processors exchange information to share the workload of a given program so that the computation can be performed in a parallel manner.

However, programming and compiling for multiprocessor systems is a far more complex task than traditional sequential programming. To efficiently and correctly utilise a multiprocessor, one has to investigate such issues as degree of parallelism, allocation of tasks, synchronisation between processors, as well as memory management.

1.1 Background

Many physical problems need to be expressed in terms of sets of quantities (called elements), which are conventionally arranged in an array of m rows and n columns. Such an array is *called a matrix.* Matrices provide a theoretical and practical way of approaching many types of problems, including the solution of linear algebraic equations, systems of linear differential equations, and many other applications [68].

This thesis discusses efficient serial and parallel methods for computing the solution of the linear system of equations $Ax = b$, reduction of a general matrix A to Hessenberg form, reduction of a real symmetric matrix B to tridiagonal form, and computing all eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Systems of linear algebraic equations occur in a variety of applications in practice and solving this problem is often one of the core components of many scientific computations. These equations are associated with many problems in engineering and science, as well as with applications of mathematics to the social sciences and the quantitative study of business and economic problems [7].

The development of efficient algorithms has received considerable attention in the literature [93,38,43,and 85]. A large number of papers have appeared in recent years describing various approaches to parallelising QR and LU factorisation on distributed memory and shared memory MIMD

architectures. These factorisations are certainly some of the most used of all numerical linear algebra computations.

Orthogonal transformations are a well-known tool in numerical linear algebra and are used extensively in decompositions such as the QR factorisation, tridiagonalisation, bidiagonalisation, Hessenberg reduction, and the eigenvalue or singular value decomposition of a matrix [90]. Widely used transformations for the QR factorisation of a matrix are Givens orthogonalisation, Householder orthogonalisation, and Gram-Schmidt orthogonalisation. Such factorisations may be realised on multiprocessors via plane rotations [24 and 78], elementary reflectors [4], or using the Modified Gram-Schmidt algorithm [90]. Several algorithms have been proposed in the past for the orthogonal factorisation of matrices, including those by Goles and Kiwi [37] on a shared memory SIMD computer, Zhou and Brent [101] on a distributed memory MIMD computer, and Wright [97] on a shared memory MIMD computer.

Modi and Clarke [67] have suggested a greedy algorithm for Givens reduction and the equivalent ordering of the rotations, but do not consider a specific architecture or communication pattern. Cosnard, Muller, and Robert [16] have shown that the greedy algorithm is optimal in the number of time steps required. Theoretical studies and comparison of such algorithms for Givens reduction have been given by Pothen, Somesh, and Vemulapati [74] and by Elden [30] and some of these algorithms have been implemented on current commercially available distributed memory multiprocessors [34].

Pothen and Raghavan [73] have compared the early work of Pothen, Somesh, and Vemulapati [74] on a modified version of a greedy Givens reduction with a standard row-oriented version of Householder transformations on a local memory system. Their tests seem to indicate that Givens reduction is superior on such an architecture. Wright [97], however, has implemented a number of algorithms on a shared memory machine for QR decomposition. These were compared and the ones using Householder transformations with multiple updates to columns were found to be very effective and better than the ones using Givens transformations.

Block algorithms developed for specific architectures rely on transferring large submatrices between different levels of storage. A numerical linear algebra library based on block methods was developed and its performance analysed in terms of architectural parameters in 1985 and early 1986 for a single cluster of the Cedar machine, and the multivector processors, Alliant FX/8 [77]. At approximately the same time, Calahan developed block LU factorisation algorithms for one CPU of the CRAY-2 [11]. In 1985, Bischof and van Loan [4] developed the use of block Householder reflectors in computing the QR factorisation and presented results on an FPS-164/MAX. It was shown in [4] that this block algorithm is as numerically stable as the classical Householder method. Most recently, Schreiber and van Loan [79] have considered a more efficient storage scheme for the product of Householder matrices. They describe the compact WY representation of the orthogonal matrix Q. Wright [97] stated that one idea popular with distributed memory machines is to use some sort of block method, though

for shared memory machines the advantages are not so obvious.

Jalby and Philippe [50] have considered the stability of the modified Gram-Schmidt algorithm and Gallivan et al [33] have analysed the performance of this algorithm as a function of block size which is presented along with experimental results on an Alliant FX/8 for single and double-level versions of the algorithm.

The various versions of the parallel LU factorisation algorithms have appeared in different contexts in the literature, some use distributed memory multiprocessor architectures [36,92,14,84] and some use shared memory [64]. The algorithms use several ways to organise the computations for calculating the LU factorisation of a matrix. The essential differences between the various forms are: the set of computational primitives required, the distribution of work among the primitives, and the size and shape of the subproblems upon which the primitives operate. Since architectural characteristics can favour one primitive over another, the choice of computational organisation can be crucial in achieving high performance. Of course, this choice in turn depends on a careful analysis of the architecture/primitive mapping. However other features are also important as will be discussed below.

One alternative method to LU decomposition which has been suggested for the solution of linear algebraic equations is QR decomposition. This algorithm is inherently stable and thus avoids the complication of pivoting. Since the operation count for QR decomposition is twice that of LU

decomposition, QR decomposition will only be competitive if the efficiency of LU decomposition with pivoting is less than half the efficiency of QR decomposition [92], but Geist and Romine [36] claim that parallel QR decomposition is not competitive to parallel LU decomposition.

We concentrate here on QR factorisation for solving systems of linear equations by using Householder transformation and the Givens method, and LU decomposition for solving systems of linear equations by using a method similar to Doolittle or to Crout reduction. These methods are the principal tools in the direct solution of linear systems of equations. Each method is based on a factorisation of the coefficient matrix of the system. We consider QR factorisation algorithms, namely, Householder and Givens transformations, which have been implemented as described in [97].

We will focus in particular on algorithms written in the C++ programming language. C++ is an object-oriented programming language which can provide various types of matrix classes (see chapter 2 for details). The use of C++ implies some storage organisation for array elements. The primitive arrays provided in C++ use storage by rows. As far as we are aware little work seems to have been done on parallelising these algorithms using C++. The comparisons in this thesis were carried out using both row and column representation of the matrix: this was made easy by the use of a C++ matrix class (see chapter 2 for details) which was altered internally. Comparisons were also carried out for versions with and without array bound checking and for versions with and without an inline function.

These extra comparisons were included to compare the algorithms under different conditions, and make spurious conclusions less likely. The code was written using the Encore Parallel Threads package (*THREADS*) [31], which provides mechanisms for synchronisation. The EPT routines can be accessed by C++ using the C linkage convention.

In [97] a number of algorithms for QR decomposition were compared and ones using Householder transformations with multiple updates to columns were found to be very effective. Most of the algorithms considered here for LU decomposition use a similar idea, though one simple implementation is also used for comparison. Some preliminary comparisons used pairwise row Gaussian elimination [82] in a similar manner to the Givens QR reduction considered in [97]. These algorithms gave significantly poorer times and were also less accurate than the column based algorithms and so are not considered here.

The use of the algorithm employed here to accomplish the LU decomposition is motivated primarily by Wright [97]. In the Householder algorithm, once a pivotal column has been completed no further changes are made to that column. In the usual Crout and Doolittle algorithms interchanges may take place in these columns corresponding to later pivotal columns. Here the algorithm is modified so that interchanges of the multipliers do not take place, with the programs organised in a similar way to Householder QR reduction. This variant of the interchange mechanism is mentioned by Gallivan et al [34], but not investigated in detail here.

Algebraic eigenvalue problems, either standard $Ax = \lambda x$, or generalised $Ax = \lambda Bx$, occur in wide variety of applications. This problem arises in many areas of physics, engineering, and science such as stability theory, the theory of vibrations, quantum mechanics, continuum mechanics, the analysis of electron orbits in atoms, the stability of structures, statistical analysis, and other areas [100]. In many cases, the problems are of very large order. For example, properties of certain quantum dynamical systems can be determined through statistical analysis of quantities computed from the eigenvalues or eigenvectors of symmetric matrices associated with those systems [51], [52].

Matrices arising in such applications sometimes have a tridiagonal form [61], and often have a banded form [51]. In addition, tridiagonal and bidiagonal matrices arise in the solution of general problems. That is, full eigensystems of dense matrices are usually computed by Jacobi methods [40], or by reduction of A matrix to tridiagonal form T by Givens rotations or Householder reflections followed by computation of the eigensystems of T [96].

A direct reduction of A to symmetric tridiagonal form T by Givens transformations or Householder transformations can be followed by computation of the eigendecomposition for the reduced matrix. Sparse symmetric eigenvalue problems are often handled by the Lanczos method [70] which itself produces symmetric tridiagonal eigenproblems. This thesis is concerned with methods for reducing a general matrix A to upper

Hessenberg form and a real symmetric matrix B to tridiagonal form.

It is well known that the methods for the reduction of a general matrix to Hessenberg form and a real symmetric matrix to tridiagonal form do not of themselves solve the eigenvalue problem, but this approach does reduce the problem to a form that can be manipulated inexpensively.

We start with the nonsymmetric case. There are several methods for reducing a general matrix to upper Hessenberg form, including some using Householder transformations and others similar to Gaussian elimination. Although those similar to Gaussian elimination are about twice as fast as those using Householder transformation, the latter method is more stable as it provides unconditional stability [9]. In chapter 5, we describe an evaluation of five parallel implementations using Householder transformations.

Similarly, the most common method for handling the symmetric eigenproblem consists of first reducing the symmetric matrix to tridiagonal form via Householder transformations. The algorithms for the reduction of a general matrix to Hessenberg form can also be used for the reduction of a symmetric matrix to tridiagonal form. We consider three parallel implementations of the reduction of a symmetric matrix to tridiagonal form in chapter 6.

Several algorithms have been developed for eigenvalue problems on parallel computers. The most robust of these methods are those that

rely first on reducing the symmetric matrix to tridiagonal form followed by handling the symmetric tridiagonal eigenvalue problem. Some works find as an example [26] and [65], use shared memory architectures. Other papers, such as [28],[55],[35], and [94], do not consider shared memory multiprocessor architectures. Dongarra and Sidani [25] consider the non-symmetric problem using shared memory but assume that reduction to Hessenberg form has already been carried out. Dongarra et al [27] consider algorithms for reduction to Hessenberg form using blocking to reduce data movement. All these papers give much relevant background to our work.

The Symmetric Tridiagonal Eigenproblem is an important problem in numerical linear algebra. There are various ways to solve this problem. Conventional methods are the shifted QR algorithm [10] and the bisection method based on the Sturm sequence(see [95] and [70]). In recent years, a divide-and-conquer technique has been developed by Cuppen [17]. Solving eigenproblems using rank-one modification was proposed by Bunch, Nielsen, and Sorensen [6], the work based on Golub [38]. Another approach using the divide-and-conquer technique is given by Krishnakumar and Morf [58]. Cuppen's method has attracted much attention. The idea has been extended and implemented using a variety of architectures, for example by Dongarra and Sorensen [26] and Ipsen and Jessup [49]. Watkins [93] states that Cuppen's method is highly parallelisable. Cuppen [17] claims this algorithm is asymptotically faster than the QR method by an order of magnitude. QR and Cuppen's methods are often used to compute all eigenvalues and eigenvectors of the matrix but the bisection method is normally used when

only a few of the eigenvalues and corresponding eigenvectors are required [49]. The Cuppen method uses a partitioning technique which reduces the original problem to smaller ones of the same type, by a rank-one modification. Cuppen [17] observed that there can frequently be some deflation in the updating process as the original matrix is rebuilt from the subproblems. Dongarra and Sorensen [26] implemented a further deflation technique to make the algorithm more efficient and more stable.

Gallivan et al [34] suggest that if only eigenvalues are desired (or all those lying in a given interval) or selected eigenpairs are desired, then bisection should be used (for example, see Wilkinson and Reinsch [96] or Parlett [70]). Such a combination has been adapted for the Illiac IV parallel computer in [60] and [46], and later for the Alliant FX/8 [65].

The implementation of Cuppen's algorithm in [26] always computes the eigenvalues to high accuracy, but some specific examples illustrate that it may not compute fully orthogonal eigenvectors (see [6], [17], and [26]). To resolve this problem, Kahan [54] suggests computing some key quantities more accurately using simulated extended precision. Sorensen and Tang [83] presented an alternative implementation scheme which was inspired by the earlier work of Kahan [54]. They showed that this method is stable but that it requires extended precision and so is machine-dependent [83]. Gu and Eisenstat [42] suggested an alternative method using the same rank-one modification as in [83], using a different approach for finding the eigenvectors. They showed that the new method is backward stable.

We implement Cuppen's method for finding all the eigenvalues and corresponding eigenvectors of the real symmetric $n \times n$ tridiagonal matrix T in this thesis. This method may be implemented recursively to produce a parallel counterpart to Cuppen's algorithm [17] as demonstrated in [26]. We discuss in addition to the recursive sequential algorithm two non-recursive sequential algorithms and their parallel implementations in order to compare these implementations in chapter 7. This chapter shows that non-recursive versions are generally the fastest and compete overall with recursive versions. In addition, we present and observe the effect of deflation on the accuracy of Cuppen's method, but this is not of primary concern in this thesis.

1.2 Motivation and Research Objectives

As mentioned before, the goal of this thesis is to consider efficient serial and parallel methods for computing the solution of the linear system of equations $Ax = b$, reduction of a general matrix A to Hessenberg form, reduction of a real symmetric matrix B to tridiagonal form, and computing all eigenvalues and eigenvectors of a real symmetric tridiagonal matrix T. The comparisons were carried out using the C++ programming language mainly using classes to represent the matrices. The parallel versions were implemented using the Encore Parallel Threads [31] package (EPT), which provides among other things the facility for the programs to explicitly create parallel "THREAD"s of execution using the "THREADcreate" function. It also provides mechanisms for synchronisation. The mechanisms used in this thesis are *"THREADjoin"s, locks, semaphores,* and *monitors.* These

mechanisms are the standard synchronisation mechanisms in the EPT package except for the lock mechanisms. The locks are provided in an extension to EPT. These mechanisms are used to provide mutually exclusive access to shared data.

Using the C++ language and shared memory computer system, the program does not have direct control of the allocation of either processors or storage. This is mainly a property of the operating system and not one of the language (the same is true using PASCAL or FORTRAN). The transfer of data between shared and cache memories is controlled by the hardware. Variables can be declared locally or globally, but in either case they will be stored in shared memory with possible copies in cache. The programmer can ask for a number of parallel threads, but again the actual allocation is controlled by the operating system and this may depend on current usage of the machine as it is time-shared.

This last point suggests that dynamic allocation of work may be particularly appropriate. However, it is difficult to measure its effectiveness as this is expected to be of most benefit under heavily loaded conditions, when timings are very variable. All the results in the sequel are based on the best times observed over a number of runs when the machine was lightly loaded. *Dynamic* allocation is likely to improve processor utilisation but does require inter-processor communication for control. Predetermined (or *static*) allocation is less flexible but avoids the need for this communication.

There are a number of considerations which were taken into account in

developing the parallel algorithms in this thesis. Firstly, and most obviously, THREAD waiting time should be kept low, both by avoiding synchronisation as far as possible and by keeping critical sections (see chapter 3 for details) as small as possible. Secondly, the storage hierarchy of the multiprocessor system implies different access times for the different parts of memory and there is the possibility of contention waits for access to shared memory [48], so that Threads [31] package (EPT), which provides data transfer should be avoided where possible. Thirdly, as the type of multiprocessor being used is normally set up in a multi-user mode, the algorithms should be adaptable so that serious degradation does not occur if the number of threads asked for is greater than the number of processors available.

In this thesis the experimental results to be presented show that dynamic task allocation can be very effective on shared memory machines, and that very high efficiency is obtainable with careful construction of the parallel algorithms even for relatively small matrices. The results also show that careful implementation taking these points into account produces significantly better times than those for simple parallel versions for all the algorithms presented here, and in most cases very good use of the parallel facilities is possible [98].

1.3 Structure of the Thesis

The structure of the thesis is as follows: The present chapter aims at presenting a brief review of the relevant algorithms. Chapter 2 outlines notation, assumptions, criteria for evaluating the numerical methods, and also gives an overview of object-oriented programming in C++.

Chapter 3 gives an introduction to some of the basic ideas in parallel computation including a review of the architecture of such computers as well as some fundamental concepts and also describes the shared memory multiprocessor used in the experiments. This chapter also deals with the parallel programming environment, an overview of the Encore Parallel Threads (EPT) package, and an introduction of inter-thread communications.

In chapter 4, we consider a number of different parallel algorithms for the QR and LU decomposition of a square matrix A. Algorithms based on both Givens and Householder transformations are considered for QR decomposition. For the LU decomposition we consider methods using both a unit lower triangular matrix L and a general upper triangular matrix U, and a unit upper triangular matrix and a general lower triangular matrix.

In chapter 5, we examine the reduction of a general matrix to upper Hessenberg form. We describe an evaluation of five parallel implementations using Householder transformations. We also consider a number of parallel implementations for comparing three different synchronisation mechanisms

(see chapter 3 for details) when applied to a particular problem.

In chapter 6, we examine the tridiagonalisation of an $n \times n$ real symmetric matrix, using Householder transformations. It is first written in a sequential form followed by parallel versions. This is similar to the reduction to upper Hessenberg form of a general matrix but has rather less scope for parallelisation. We compare the performance of the three implementations. These tests use both a simple Matrix class used for a general matrix and a special Symmetric Matrix class written so that only half the matrix is stored.

In chapter 7, we describe the implementation of Cuppen's method for finding all of the eigenvalues and corresponding eigenvectors of real symmetric $n \times n$ tridiagonal matrix T. We review the description of the divide-and-conquer method presented in [17], examine the arithmetic complexity of this method, discuss a number of sequential algorithms, both recursive and non-recursive. We consider parallel implementations of these versions, four of which are variant recursive versions and two different non-recursive versions. We also consider parallelisation of the matrix multiplication part of the algorithm. This chapter also provides some experimental results illustrating the effect of deflation on accuracy, as well as comparison of the parallel implementations with and without the additional parallelisation for matrix multiplication.

Finally, conclusions and suggestions for future research are presented in Chapter 8.

CHAPTER 2

Preliminaries

This thesis is concerned with the development of parallel algorithms for numerical linear algebra. We have designed different implementations of such algorithms and compared them empirically. All the comparisons have been carried out using the C++ programming language using the Encore Parallel Threads package on a shared memory multiprocessor (the Encore Multimax). The comparisons were carried out by measuring the elapsed time using each of the implementations. These algorithms are as follows:

- We firstly considered the QR decomposition of a matrix for solving systems of linear equations, that is using the decomposition of a square matrix into an orthogonal matrix Q and an upper triangular matrix R. In order to compute this we used Householder and Givens transformation methods which were first written in a sequential form followed by their parallel versions. Here the Q matrix was not stored.

- Secondly, we examined two methods for the LU decomposition of a matrix. One uses a unit lower triangular matrix L and a general

upper triangular matrix U and is similar to Doolittle reduction and GAXPY Gaussian elimination [40]. The other method uses a general lower triangular matrix L and a unit upper triangular matrix U and is similar to Crout reduction. A comparison of these methods is made using a number of variant parallel implementations. Results similar to QR decomposition were obtained. This work is described in [56].

- Thirdly, we considered the reduction of a general matrix to upper Hessenberg form using Householder transformations. A variety of parallel algorithms were investigated. Comparisons of performance were carried out between the five implementations. This work has been written up as a technical report [57].

- Fourthly, we compared the *lock, semaphore,* and *monitor* synchronisation mechanisms for algorithms which were otherwise the same. Two parallel implementations of the reduction to Hessenberg form were used in these experiments. We came to a clear conclusion about using the synchronisation mechanisms, which is that, in all cases, using "Locks" is more efficient than using "Monitors" or "Semaphores".

- Fifthly, we considered the tridiagonalisation of an $n \times n$ real symmetric matrix, using Householder transformations. This was first written in a sequential form followed by its parallel versions. This is similar to the reduction to upper Hessenberg form of a general matrix but has rather less scope for parallelisation. We compared the performance of the three implementations.

All these comparisons were carried out using both row and column representations of the matrix; this was made easy by the use of the simple C++ matrix class which was altered internally. Comparisons were also carried out between cases with and without array bound checking. These extra comparisons were included in order to compare the algorithms under different conditions, and to make spurious conclusions less likely.

- Finally, we implemented Cuppen's divide-and-conquer method to compute all of the eigenvalues and corresponding eigenvectors of an $n \times n$ real symmetric tridiagonal matrix. The method uses a divide-and-conquer technique which reduces the eigenvalue problem for a symmetric tridiagonal matrix to smaller problems of the same type by a rank-one modification. The algorithm can be parallelised using different schemes some being recursive and some non-recursive. We discuss a number of sequential and a variety of parallel approaches for the implementation of this algorithm.

Notations are introduced in section 2.1 along with assumptions about the matrices used. Definitions and theoretical results about the measures of quality employed are presented in section 2.2. In sections 2.3 and 2.4, we introduce measures of the sensitivity of a linear system and an eigenvalue problem, respectively. Real general matrices were used to test the Householder and Givens transformation methods, the Doolittle and Crout reduction, and the reduction of a general matrix to upper Hessenberg form. To test the tridiagonalisation phase we used real symmetric matrices. The

symmetric tridiagonal matrices used to test the eigenvalue problem of the divide-and-conquer method are given in section 2.5. Relevant features of the object-oriented programming language C++ are introduced in section 2.6. The overview of classes used here and inline functions are presented in subsections 2.6.1.

2.1 Notations and Assumptions

The following notation will be used throughout this thesis. Unless otherwise specified, a superscript T denotes transpose. All quantities are assumed to be real. We use the notation of Golub and van Loan [40] where $A(k, j : n)$ denotes the vector consisting of the elements of $A(k, i)$, $i = j, \ldots, n$, for the algorithms in this thesis.

A denotes an $n \times n$ general matrix having the QR decomposition $A = QR$, where Q is an $n \times n$ orthogonal matrix and R is an $n \times n$ upper triangular matrix. To solve $Ax = b$, we first use Gaussian elimination to factor the nonsingular matrix A as $A = LU$, where L is an $n \times n$ lower triangular matrix, U is an $n \times n$ upper triangular matrix, and where the permutations has been ignored.

T denotes an $n \times n$ symmetric tridiagonal matrix with the eigendecomposition $T = QDQ^T$ where D is an $n \times n$ diagonal matrix with the eigenvalues $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n|$ as its diagonal elements. The $n \times n$ matrix Q is orthogonal and has as its columns the eigenvectors $q_1, q_2, \ldots, q_n$

.

2.2 Measures of the Quality of a Method

In this thesis, different methods and implementations are compared empirically in terms of runtime and parallel efficiency.

- *Speed-up :* In evaluating a parallel algorithm for a given problem, it is quite natural to do it in terms of the best available sequential algorithm or corresponding one for that problem. The speedup S_p attained when using p processors to solve a problem instance of size N is defined by the formula

$$S_p(N) = \frac{T_s(N)}{T_p(N)}.$$

where $T_s(N)$ and $T_p(N)$ are the times for the sequential and parallel versions.

- *Overhead T_o :* The sum of the time spent by all processors with other processors, waiting for signals, time in starvation, etc. [81]. Overhead is defined by

$$T_o(N) = (p \times T_p(N)) - T_s(N)$$

where $T_s(N)$ and $T_p(N)$ are the times for the sequential and parallel versions and p is the number of processors.

- *Efficiency :* The efficiency E_p of a parallel algorithm is defined to be the speedup divided by p, which has the effect of scaling the speedup to a value usually between 0 and 1. Symbolically, efficiency is defined by

$$E_p(N) = \frac{T_s(N)}{p \times T_p(N)}$$

where $T_s(N)$ and $T_p(N)$ are the times for the sequential and parallel versions and p is the number of processors. In this thesis, the mean efficiency is calculated by averaging the efficiencies. Since $S_p(N) \leq p$, we have usually $E_p(N) \leq 1$ and an efficiency of $E_p(N) = 1$ corresponds to a perfect speedup of $S_p(N) = p$ [39].

- *Amdahl's Law :* Amdahl noted that the computation time can be divided into a parallel portion and a sequential portion, and no matter how high the degree of parallelism in the former, the speedup will be asymptotically limited by the latter which must be performed on a single processing element [2].

2.3 Measures of Sensitivity of Linear Systems

When we solve a system of linear equations or compute an eigenvalue problem we usually obtain an approximation of the exact result. The result will be affected by the roundoff errors made during the computation. The result produced by the algorithm is accepted as correct as long as the error of the computation is less than some specific value, where the error is the difference between the exact result and the computed result. In case the errors are not small, it is important therefore to ask what effect small changes or perturbations in the coefficients have on the solution of the system or the eigenproblem. How do these errors affect the accuracy of the computed solution?

Consider a linear system,

$$Ax = b \tag{2.3.1}$$

where A is a nonsingular matrix, and b is the nonzero right hand side vector. The system has a unique solution x. Suppose the system has $\hat{x}$ as computed solution of the perturbed system

$$A\hat{x} = \hat{b} \tag{2.3.2}$$

where $\hat{b} = b + \delta b$ and $\hat{x} = x + \delta x$. We hope that if δb is small compared to b, then δx is also small compared with x. The size of δb and δx relative to b and x are given by $\|\delta b\|/\|b\|$ and $\|\delta x\|/\|x\|$ respectively. We wish to relate $\|\delta x\|/\|x\|$ to $\|\delta b\|/\|b\|$.

Substituting equation (2.3.1) into equation (2.3.2) and multiplying by the matrix A^{-1} gives

$$\delta x = A^{-1}\delta b. \tag{2.3.3}$$

Whatever vector norm we have chosen, we will use the induced matrix norm to measure matrices. Using the properties of the vector norm and its induced matrix norm, equations (2.3.1) and (2.3.3) imply that

$$\|\delta x\| \leq \|A^{-1}\|\|\delta b\| \tag{2.3.4}$$

and

$$\|b\| \leq \|A\|\|x\|, \tag{2.3.5}$$

or equivalently

$$\frac{1}{\|x\|} \leq \|A\| \frac{1}{\|b\|} \tag{2.3.6}$$

Combining inequalities (2.3.4) and (2.3.6) yields an important inequality

$$\frac{\|\delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\| \frac{\|\delta b\|}{\|b\|} \tag{2.3.7}$$

which provides a bound for $\|\delta x\|/\|x\|$ in terms of $\|\delta b\|/\|b\|$. The factor $\|A\|\|A^{-1}\|$ is called the condition number of A and is denoted by $\kappa(A)$ [7].

The inequality (2.3.7) needs to be interpreted correctly. If $\kappa(A)$ is close to 1, then small relative changes in the components of the linear system of equations produce small relative changes in the solution. In this case we say that the linear system is *well-conditioned.* A linear system of equations is said to be *ill-conditioned* if $\kappa(A)$ is large and small changes in problem parameters may cause large changes in the solution.

The accuracy of a method may be measured by the residual in the computed solutions of the linear system (2.3.1). In [39], the *residual vector* of a computed solution $\hat{x}$ to the equation (2.3.1) is defined by

$$r = b - A\hat{x} \tag{2.3.8}$$

If r were zero, then $\hat{x}$ would be the exact solution of the linear system (2.3.1). Thus we would expect r to be small if $\hat{x}$ were a good approximation to the exact solution. If r were small, then $A\hat{x}$ effectively approximates the right hand side b. This is true in some cases, but if A is ill-conditioned, the size

of r can be very misleading. As an example, consider the system

$$\begin{pmatrix} 0.780 & 0.563 \\ 0.913 & 0.659 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.217 \\ 0.254 \end{pmatrix}, \tag{2.3.9}$$

and the approximate solution

$$\hat{x} = \begin{pmatrix} 0.341 \\ -0.087 \end{pmatrix}. \tag{2.3.9a}$$

Then, the residual vector is

$$r = \begin{pmatrix} 10^{-6} \\ 0 \end{pmatrix}. \tag{2.3.9b}$$

Now consider another very different approximate solution

$$\hat{x} = \begin{pmatrix} 0.999 \\ -1.001 \end{pmatrix}, \tag{2.3.9c}$$

and the corresponding residual vector

$$r = \begin{pmatrix} 0.0013\ldots \\ -0.0015\ldots \end{pmatrix}. \tag{2.3.9d}$$

By comparing the residuals (2.3.9b) and (2.3.9d) we could easily conclude that (2.3.9a) is the better approximate solution. However, the exact solution of (2.3.9) is (1,-1), so the residuals give completely misleading information.

If the matrix A is well-conditioned, the residual vector provides a valid estimate of the accuracy of an approximate solution $\hat{x}$. However, in general, small residuals do not always imply high accuracy.

When we solve a system of linear equations, we are concerned with knowing whether or not our computed solution is accurate. This calls for an error analysis which attempts to determine the effect of *round-off errors.* The round-off errors result from inaccuracy in computation. Computational procedures may consist of hundreds or even thousands of elementary operations and the cumulative effect of round-off is sometimes severe.

For systems of linear equations there is a way to analyse an error based on the residual and condition number. From (2.3.8) consider

$$r = A(A^{-1}b - \hat{x}) \tag{2.3.10}$$

so that, if $E = A^{-1}b - \hat{x}$ is the error in the approximate solution, then,

$$E = A^{-1}r. \tag{2.3.11}$$

This is the fundamental relation between the residual and the error. Then

$$\|E\| \leq \|A^{-1}\| \, \|r\| = \kappa(A)\frac{\|r\|}{\|A\|}, \tag{2.3.12}$$

so that the error is bounded by $\kappa(A)$ times a normalised residual vector. The estimate (2.3.12) shows that if $\kappa(A)$ and $\frac{\|r\|}{\|A\|}$ are both small, then the error is also small. On the other hand, from $r = AE$, we obtain

$$\frac{\|r\|}{\|A\|} \leq \|E\|$$

so that if $\frac{\|r\|}{\|A\|}$ is large, so is the error.

If the condition number is large, then small changes in the data may cause large changes in the solution depending on the particular perturbation. The practical effect of a large condition number depends on the accuracy of the data and the word length of the computer being used. If the data are measured quantities, however, the computed solution may not have any meaning even if computed accurately [39].

2.4 Measures of Sensitivity of Eigenproblem

The accuracy for the eigenvalue problem can also be determined by residual error in the computed solutions and by the orthogonality of the computed eigenvectors. For a symmetric tridiagonal matrix T with computed eigendecomposition $\hat{Q}\hat{D}\hat{Q}^T$, the quality of the solution can be measured using the residual $\Re$

$$\Re = \frac{1}{|\hat{\lambda}|_{max}} \max_i \|T\hat{q}_i - \hat{\lambda}_i\hat{q}_i\|_2$$

and a measure of orthogonality of the eigenvectors

$$\Im = \|\hat{Q}^T\hat{Q} - I\|_\infty,$$

where $\hat{D} = diag(\hat{\lambda})$ and $\hat{q}_i$ is the i^{th} column of $\hat{Q}$. The residual error is thus determined by the largest residual error for any single computed eigenpair.

Theorem 2.4.1 (see Theorem 2.1 in [53]).

Let $\hat{Q}\hat{D}\hat{Q}^T$ be the computed eigendecomposition of a symmetric tridiagonal matrix T. If $\Re \leq \epsilon_1$, and $\Im \leq \epsilon_2$, then there exists a matrix

E such that

$$T + E = \hat{Q}\hat{D}\hat{Q}^T, \text{ and } \|E\|_2 \leq \sqrt{n}\left[|\hat{\lambda}|_{max}\epsilon_2 + |\hat{\lambda}|_{max}\epsilon_1\sqrt{1+\sqrt{n}\epsilon_2}\right],$$

where $|\hat{\lambda}|_{max} = max(|\hat{\lambda}_1|, |\hat{\lambda}_n|)$.

Theorem 2.4.1 above shows that if the residual $\Re$ and orthogonality $\Im$ are small, then $\hat{Q}\hat{D}\hat{Q}^T$ is the exact eigendecomposition of a matrix $T + E$ nearly equal to T. In this result E is neither symmetric nor tridiagonal in general.

2.5 Test Matrices

In this thesis the algorithms for the QR and LU decomposition and the reduction of a general matrix to upper Hessenberg form were tested only with circulant matrices. Since the results are only concerned with timings and because these algorithms are independent of the test matrix apart from interchanges in the LU decomposition, only one type of matrix is needed to provide a satisfactory test. For the same reason the algorithms for the tridiagonalisation of an $n \times n$ real symmetric matrix were tested again using only one type of symmetric test matrix. If accuracy were an issue then many more types of matrices would need to be tested. The serial and parallel algorithms were tested on the collection of matrices given in this section. These are as follows:

1. *General Test Matrices:*

Only one type of test matrix was used. This was the circulant matrix

A. The $(n \times n)$ circulant matrix A is given by

$$A_{j,k} = \begin{cases} n+k-j+1, & \text{if } k < j \\ k-j+1, & \text{otherwise,} \end{cases}$$

all the elements of the matrix are real and different from zero [44]. If the right hand side vector b given by

$$b_j = n(n+1)/2$$

is used then the corresponding solution of the equations x is given by

$$x_j = 1,$$

for $j = 1, 2, \ldots, n$.

2. *Symmetric Test Matrices:*

For the symmetric matrix tests, the matrices B given by

$$B_{ij} = i + j + 1.31/(i+j),$$

for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$, were used.

3. *Tridiagonal Test Matrices:*

A number of test matrices were used to test Cuppen's algorithm as the amount of deflation in this method depends on the test matrix and this affects the timings.

The matrix $T[\beta, \alpha, \beta]$: The matrix with $\alpha = 2$ in each diagonal position and $\beta = -1$ in each off-diagonal position was used. It has eigenvalues given in [93] as

$$\lambda_k = 4\sin^2[k\pi/2(n+1)],$$

for $k = 1, 2, \ldots, n$. The eigenvector corresponding to the eigenvalue λ_k is $q^{(k)}$ given by

$$q_i^{(k)} = \sin[ik\pi/(n+1)],$$

for $i = 1, 2, \ldots, n$ and $k = 1, 2, \ldots, n$. The matrix $[-1, u, -1]$ has value -1 in each off-diagonal position and the value $u = i \times 10^{-6}$ in the i^{th} diagonal position, for $i = 1, 2, \ldots, n$. This has been chosen because the matrix undergoes little deflation when its eigenproblem is solved by Cuppen's divide-and-conquer method [17].

2.6 Object-Oriented Programming in C++

Object-oriented programming makes use of the *class* construct. One advantage of programming in an object-oriented language is that new types can be created through this *class* mechanism. In C++ terminology, a class is a data type that contains data and functions. An *object* is simply a user-defined class variable. Every object will be an instance of a class. We can define operations in the class and these operations can be performed on instances of that class. A class declaration determines how storage is to be used to represent an object and which operations are to be available to manipulate that storage. Language support for classes first came with

Simula 67 [15].

C++ was developed by Stroustrup in the early 1980s. Pohl [71] states that Stroustrup had two main goals: (1) C++ was to be compatible with ordinary C, and (2) it was to extend C using the class construct of Simula 67. The class construct is an extension of the C *struct*. The language, in an early form of the C++ programming language, is described by Stroustrup [89].

Booch [5] defines object-oriented programming as "a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships."

Object-Oriented Programming has grown from a radical concept of the 1960's to routine practice among serial programmers in late 1980's. Can it be as useful in parallel programming as in serial programming? Object-oriented programming involves simple components that can be tested independently and be used to assemble complex programs. Most of these simple (independent) components may be used in their own right in other programs.

The use of the object model helps us to exploit the expressive power of object-based and OOP languages. As Stroustrup points out, "it is not always clear how best to take advantage of a language such as C++.

Significant improvements in productivity and code quality have consistently been achieved using C++ as 'a better C' with a bit of data abstraction thrown in where it is clearly useful. However, further and noticeable larger improvements have been achieved by taking advantage of class hierarchies in the design process. This is often called object-oriented design and this is where the greatest benefits of using C++ have been found" [87].

Hwang [47] states that the popularity of OOP is attributed to three application demands: First, there is increased use of interacting processes by individual users, such as using multiple windows. Second, workstation networks have become a cost-effective mechanism for resource sharing and distributed problem solving. Third, multiprocessor technology has advanced to the point of providing supercomputing power at a fraction of the traditional cost.

Perhaps the most widely known OOP language is C++, and so the algorithms, in this thesis have been implemented in this language in order to take advantage of its features and facilities. Lewis and El-Rewini [62] stress the point that OOP features information hiding and encapsulation, meaning that (*i*) each object hides the implementation details and also data from view of outside clients only a restricted set of methods is visible, and (*ii*) changes to the implementation of the object do not require changes to the code that uses the object, so long as the interface is stable. A modern programming language would be a poor one if the programmer had no means of building data items to match the conceptual items within the solution to

the problem being addressed. To re-phrase, data abstraction, the ability to create user-defined data types, is essential in any modern language.

Abstract data types are implemented in C++ through the *class* facility. Classes allow a programmer to control the visibility of the underlying implementation. What is public is accessible and what is private is hidden. Data hiding is one component of object-oriented programming. Classes have member functions, including those that overload operators. Member functions allow the programmer to code the appropriate functionality for the abstract data type [72].

A class is an extension of the idea of the *struct* construct in conventional C. The structure type allows the programmer to group together several pieces of data and treat them as a single data item. C++ *structs* behave as a class whose members are publicly exported by default, whereas classes' members are private to the class by default. In both *class* and *struct* access to variables and functions can be changed using the keywords *private* and *public*. The keyword *public* indicates the visibility of the members, and members of an object are accessible to any function having access to the declaration of that object class and scope access to the object itself. If the class is not used with the *public* keyword, the members are *private* to it. *Private* members are available for use only by other member functions of the class. *Public* members are available for use by any function within the scope of the object declaration. Privacy allows part of the implementation of a class type to be *hidden*. This restriction prevents unanticipated

modifications to the data structure. As the default for class is *private*, we need only use the keyword *public*.

The C++ class concept supports *data hiding*. Data hiding is a feature of object-oriented programming. When the representation of a type is hidden, some mechanism must be provided for a user to initialise variables of that type. A simple solution is to require a user to call some function to initialise a variable before using it. This is often done by a constructor when the variable is declared. Data hiding is a property of class objects whereby the internal structure of an object is hidden from the rest of the program, which can interact with the object only by sending it messages and receiving its replies using the public members of the class.

2.6.1 Classes

We used one matrix package and some matrix classes throughout this thesis. The package is called *newmat* and is intented for scientists and engineers who need to manipulate a variety of matrix types using standard matrix operations. The package includes the operations $*$, $+$, $-$, inverse, transpose, conversion between types, submatrix, determinant, Cholesky decomposition, Householder triangularisation, singular value decomposition, eigenvalues of a symmetric matrix, sorting, fast fourier transform, printing and an interface with "Numerical Recipes in C" [18].

The matrix classes are namely a simple matrix and vector class, and a symmetric matrix class. We wrote these classes to access the matrices and

the computations. This enabled the computation to be carried out using both row and column representations of the matrix. This was made easy by the use of the C++ class facility.

Classes help the programmer provide higher-level programming constructs than either functions or structs alone support. These constructs serve as abstractions, and what they abstract typically relates closely to the application for which the program is being written. The use of class emphasises this mapping from application domain abstractions to solution domain abstractions in a way that data structs alone cannot [15].

Newmat Matrix Package

The *newmat* matrix package is used for the manipulation of matrices, including the standard operations. A matrix is a two dimensional array of numbers. However, very special operations such as matrix multiplication are defined specifically for matrices. This means that a *matrix* package tends to be different from a general *array* package. The package is designed for version 2 of C++ by Davies [18].

The structure of matrix objects is described in the following way. Each matrix object contains the basic information such as the number of rows and columns and the status variable plus a pointer to the data array which is on the heap.

In this package, the elements of the matrix are stored as a single array.

Alternatives would have been to store each row as a separate array or a set of adjacent rows as a separate array, but large arrays may cause problems for memory management in smaller machines.

The *newmat* matrix package has a two-stage approach to evaluating matrix expressions which is used to improve efficiency and reduce the use of temporary storage. A first requirement is that a matrix expression is evaluated with close to the same efficiency as a hand-coded version. A second requirement is that temporary matrices generated during the evaluation of the expression are destroyed as quickly as possible.

The package does not have graceful exit from errors. All errors are treated as fatal. It is important to mention that in the *newmat* matrix package access to matrix element arrays involves array bound checking as well as access via functions.

Matrix and Vector Class

This section describes a *matrix* and *vector* class which is used for most of our experiments throughout this thesis. The class consists of two parts. One part is a one-dimensional and the other part is a two-dimensional array. A common mistake using bare C++ arrays is to access a subscript out of range elements. In C++ these problems can be taken care of by defining an array type in which bounds are tested.

An example of a matrix and vector class *matrix.h* which is particularly

convenient for the present thesis is given below. The class was altered internally to give storage of the matrix by rows or by columns and to include or not to include array bound checking. On the other hand, the matrix representation for C++ Data Arrays and the Newmat Matrix Package have no facility to give storage of the matrix by columns by internal alteration. Also we should note that there is no array bound checking when using C++ Data Arrays, while the Newmat Matrix Package always has array bound checking. Let us consider as an example a Matrix and Vector class that enforces data hiding and which can be declared as follows:

```
// File matrix.h
// Header file for class Matrix
#include < iostream.h >

//Definition for Matrix
class Matrix
{
    int m; int n; int sz; double * a;
    public:
    Matrix(int ma, int na);
    ~ Matrix();
    inline double& operator()(int i, int j);
};

//Definition for Vector
class ColumnVector
{
   int n; double * a;
   public :
   ColumnVector(int na);
   ~ ColumnVector();
   inline double& operator()(int i);
};
```

A C++ program consists of a number of source files. Each source file

is compiled separately into a machine-code file. The resulting machine-code files are then linked to one another and with any needed library files to yield a single executable file. Any program that uses the Matrix and Vector class will include this header file with the statement

#include "matrix.h".

(The names of header files written by the user are enclosed in quotation marks rather than angle brackets.) The header file must be included in all programs that create and use objects of this matrix class. The above class must be compiled and linked to any program using matrices of the class. We give the member function definition for the Matrix and Vector class as follows:

```
// File Matrix.c
// Source file for class Matrix
#include " matrix.h "
Matrix::Matrix(int ma, int na)
{
  m = ma;
  n = na;
  sz=m * n;
  a = new double[sz];
  for (int i = 0; i < size; i++)
  a[i] = 0.0;
}
Matrix::~Matrix()
{
    delete [] a;
}
inline double& Matrix::operator()(int i, int j)
{
   int pos = (j - 1) * n + i - 1;
   if ((pos < 0) || pos >= sz))                    // omit
   {                                               // for no checking
```

```
            cout<< " Matrix : subscript out of range "
                 << i << "  " << j << endl;
            exit(1);
        }
        return a[pos];
    }

    ColumnVector::ColumnVector(int na)
    {
      n = na;
      a = new double[n];
      for (int i = 0; i < n; i++)
      a[i] = 0.0;
    }

    ColumnVector::~ColumnVector()
    {
        delete [] a;
    }

    inline double& ColumnVector::operator()(int i)
    {
        if ((i < 1) || (i > n))          // omit for no checking
        {
            cout<< " ColumnVector : subscript out of range "
                 << i << endl;
            exit(2);
        }
        return a[i - 1];
    }
```

A C++ *constructor* can provide a way to automatically initialise data. A constructor is a member function whose name is the same as the class. The constructors "Matrix::Matrix(*int ma,int na*)" and "ColumnVector::ColumnVector(*int na*)" allow the programmer to build dynamically allocated arrays.

The class creates the C++ data array by using *new* and removing the object by using *delete*. Each time a new object is defined, its class

constructor is automatically invoked. The statement

$$a = new\ double[sz];$$

invokes the C++ *new* operator to create an array of variables of type double and places the address of this variable in the pointer variable a. The pointer variable a is used as the base address of a dynamically allocated array whose number of elements is the same as the value of sz. The keyword *new* is an unary operator that takes as an argument a data type that can include an array size. It allocates the appropriate amount of memory to store this type from free store. Storage obtained by *new* is persistent and is not automatically returned on block exit. When storage return is desired, a *destructor* function must be included in the class. A destructor is identified by having the same name as the class name prefixed by the tilde symbol ($\sim$). Typically, a destructor uses the unary operator *delete* to deallocate storage allocated by *new* [71].

Inline Function

In C++ programming there is sometimes a need for many calls of functions that are very simple and small. Unfortunately, a certain amount of computational overhead is associated with each function call and return. In order to reduce the cost of calls and improve the speed of programs, C++ provides the *inline* feature to avoid function call overhead. A function may be made inline by explicit use of the *inline* and declaration keyword on the function definition, for example

```
inline double sqr (double x)
{
    return x * x;
}
```

The definition of an inline function must occur before it is used. Generally, this is accomplished by putting the inline function definition in a header file. The appearance of an inline declaration for a function, after any call to the function, is an error. The C++ programming language provides an alternative declaration of an inline function. The function has its complete definition placed in the class declaration. Any function that is defined (and not just declared) inside a class declaration is considered to be an inline function. The keyword inline does not have to be used.

The inline function modifier can be used to request that a function be expanded inline. This expansion avoids the overhead of a function call by expanding the body of the function at the point it is called. The compiler will attempt to inline expand the code of a function that is declared as such before it is used, where the declaration and use appear in the same source stream. Such inline expansion can result in large savings of central processing unit time [15].

An inline function may use extra space because the inline function duplicates the code for every function call and one may think that it automatically increases code space. This is not necessarily true because inline functions are designed for small functions in C++ such as matrix or vector subscripting. When a function call requires code to pass arguments, make the call, and handle the return value this code is not present for an inline function. If our inline function turns out to be smaller than the amount of code necessary for the ordinary call, we are actually saving space [29]. We shall discuss further the performance of inline functions in chapter 4.

CHAPTER 3

Parallel Computer Architectures and Programming

3.1 Parallel Computers

The basic idea in parallel computing is the execution of a program on two or more processors at the same time on a single problem and in a single system. Parallel computing may offer a number of advantages. Depending on the type of application and tools available, a single large job can be decomposed into several smaller tasks that can run simultaneously for faster running. This implies that two or more processors are operated simultaneously.

The motivation for using parallel computers is the hope that if one processor executes a task in time t then p processors can perform the task in time t/p. Clearly, the nearer this time is to t/p, the better the parallel algorithm. However, an execution time of t/p can be achieved only in very special situations.

3.2 Parallel Computer Architectures

Parallel architectures may be classified in a number of ways. Flynn [32] proposed a classification based on the multiplicity of instruction streams (IS) and data streams (DS) in a computer system. The instruction stream is defined as the sequence of instructions as performed by the machine and the data stream as the sequence of data called for by the instruction stream (including input and partial or temporary results). The classification has the following form:

- Single instruction stream - single data stream (SISD)
- Single instruction stream - multiple data stream (SIMD)
- Multiple instruction stream - single data stream (MISD)
- Multiple instruction stream - multiple data stream (MIMD)

Although this classification gives a general categorisation, the current development of computer architectures is more complicated and some architectures exhibit aspects of more than one category.

3.2.1 SISD Computer Organisation

The SISD computer is typically designed based on the von Neuman model as a single stream of instructions controlling a single stream of data (see figure 3.1). Instructions are executed sequentially but these may be overlapped in their execution stage (pipelining). Most present day SISD

uniprocessor systems are pipelined. A SISD computer may have more than one functional unit. All the functional units are under the supervision of one control unit [48]. This type of computer consists of three levels: the control unit (CU), the processor (P), and the memory modules (MM).

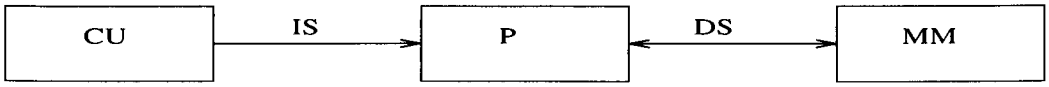


Fig. 3.1 SISD Computer Structure.

Example 1. Let us consider the implementation of a matrix multiplication algorithm on different computer architectures. The product of a $n \times p$ matrix A and a $p \times n$ matrix B is a matrix C whose elements are given by

$$C_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj} \tag{3.1}$$

for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$. There are $n^2 p$ cumulative additions and multiplications to be performed in equation (3.1).

In a conventional SISD uniprocessor system, the n^3 cumulative multiplications are carried out by a sequentially coded program with loops corresponding to the three indices to be used. The time complexity of multiplying two $n \times n$ matrices for this example is clearly $O(n^3)$.

3.2.2 SIMD Computer Organisation

A SIMD computer is an array processor model. It executes a single stream of instructions from a central control unit (CU) and operates on

several data elements simultaneously (see figure 3.2). There are a number of identical processing elements (P) each receiving the same broadcast instruction to be performed on their own data item. Each of the N processors have their own local memory (M) where they can store both program sections and data [1].

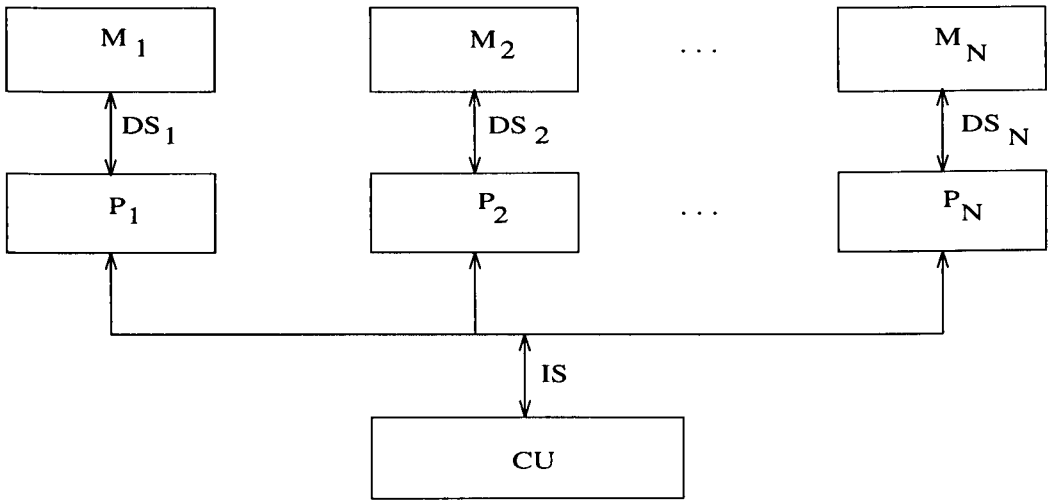


Fig. 3.2 SIMD Computer Structure.

As an example, let us consider the implementation of the matrix multiplication algorithm (*Example 1*) on an SIMD computer with n processing elements. Hwang and Briggs [48] have stated that the algorithm structure depends heavily on the memory allocations of the A and B matrices in the processing elements' memories. Column vectors are stored within the same processing element memory. This memory allocation scheme allows parallel access to all the elements in each row vector of the matrices. Based on this data allocation, the two *parallel* operations correspond to *vector load* for initialisation and *vector multiply* for the

inner loop of additive multiplications. If there are n processors the time complexity has been reduced to $O(n^2)$. Therefore, the SIMD algorithm is n times faster than the SISD algorithm for matrix multiplication. It should be noted that the *vector load* operation is performed to initialise the row vectors of matrix C one row at a time. In the *vector multiply* operation, the same multiplier a_{ij} is broadcast from the control unit (CU) to the processing elements to multiply all elements of the i^{th} row vector of B i.e. b_{ik} for $k = 1, 2, \ldots, n$. In total, n^2 vector multiply operations are needed in the double loops. The successive memory contents in the execution of the above SIMD matrix multiplication program are illustrated in [48]. Each *vector multiply* instruction implies n parallel scalar multiplications in each of the n^2 iterations. A number of parallel SIMD computers are on the market, including the AMT DAP-610, the Thinking Machines CM-2, and the MasPar MP-1 [22].

3.2.3 MISD Computer Organisation

The MISD computer is the third classification of Flynn which involves multiple instruction streams controlling a single data stream. It consists of N processors (P), each receiving different instructions from its control unit (CU) and performing operations on the same data stream which is received from the memory module (MM) at each step (see figure 3.3). The conventional view is that such a machine has not yet appeared, although there is also a view that pipelined vector processors belong to this class rather than to SIMD or SISD [2].

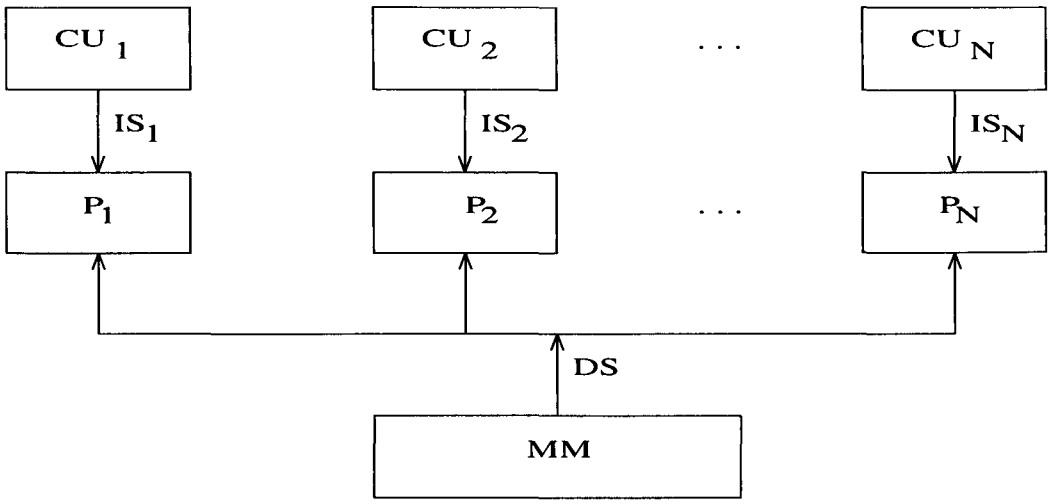


Fig. 3.3 MISD Computer Structure.

3.2.4 MIMD Computer Organisation

A MIMD computational model corresponds to a multiple stream of instructions each of which are applied to separate data items. This class is very broad because it comprises all multiprocessor systems. There are two basic types of MIMD architectures, namely *distributed memory* multicomputer systems (loosely coupled) and *shared memory* multiprocessor systems (tightly coupled).

The fundamental difference between the two systems is in the design of the system memory. The defining characteristic for these two systems is the communication mechanism provided by the underlying hardware: the shared memory system assumes hardware support for shared-memory, while the distributed system makes no such assumption. Interprocessor communication in distributed systems is usually handled by a local area network and takes place through message passing, whereas in shared

memory systems communication is achieved via a *common bus* or an *interconnection network*. (In the sequel, we will use the terms 'shared memory' and 'common memory' interchangebly.) The processors in a shared memory system communicate with each other through shared variables in a common memory. A major advantage of a shared memory system is potentially very rapid communication of data between processors. A serious disadvantage is that different processors may wish to use the common memory simultaneously, in which case there will be a delay until the memory is free. This delay, called *contention time*, can increase as the number of processors increases.

3.2.4.1 Shared Memory Systems

A shared memory system is composed of autonomous computing units which are usually used to execute a single task together. This is achieved by having a common memory. All processors can access any part of the common memory because it is a single shared memory and accessible to all processors (see figure 3.4). Communications among the processors are accomplished through reading from and writing to the common memory. The simplest implementation of the shared memory model is to connect processors and common memory modules by a single bus. This is called a bus-based shared memory multiprocessor. Bus-based shared memory multiprocessor systems representative of this structure are the Encore Multimax, the Sequent Symmetry, the Flex/32, and the Alliant FX/8 [21].

The multiprocessor system used to perform the experiments described in this thesis is a bus-connected shared memory Encore Multimax computer running the UMAX operating system. The machine (locally called Newton) has 14 NS32532 processors, each with 256 Kb processor cache memory.

The bus-based shared memory multiprocessor systems have a common bus. An advantage of this is that a very small number of connection lines are used, but there may be contention (*bus contention*) for use of the bus by different processors; this can become a severe problem as the number of processors increases. The common bus is a key system element of shared memory systems. It provides a common communication path and carries instructions and data between the common memory, processors, and I/O subsystem devices. An illustration of a multiprocessor system using the bus-based shared memory structure is shown in figure 3.4.

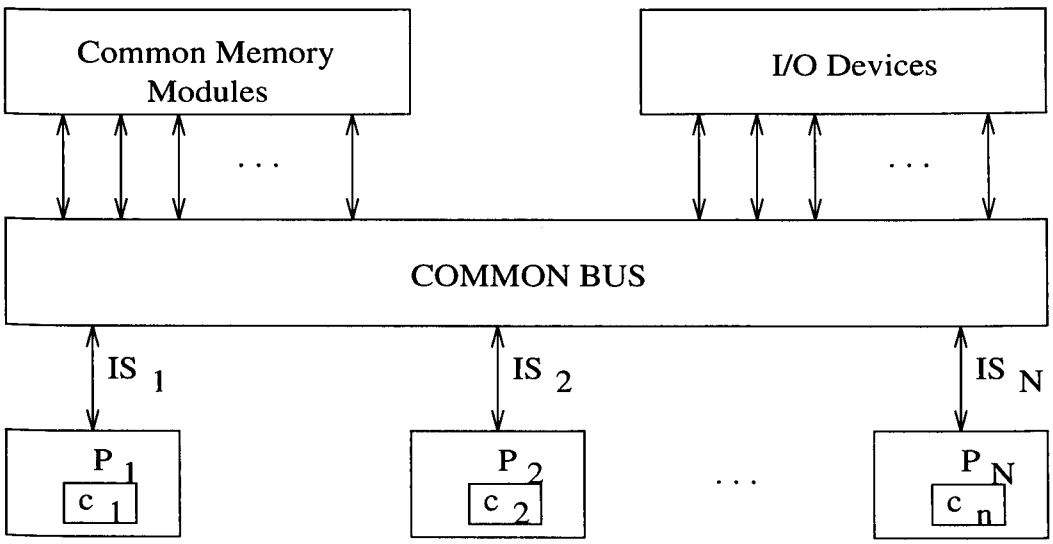


Fig. 3.4 Bus-Based Shared Memory Structure

The common memory may be organised as several memory banks. A memory bank is a unit of interleaved memory allowing only a single read or write at a time. It may be the case that some processor attempts to access a memory bank which is being accessed by another processor. If this is the case the common bus provides a mechanism for resolving the problem. In order to minimise memory contention, processors often have large local *cache* memories (c in figure 3.4) to reduce the number of memory requests. If suitable cacheing strategies are employed then the shared memory architecture works very well in practice, allowing the current generation of multiprocessors to utilise up to thirty processors [86]. Caches are high-speed memory units used as a buffer, and placed between the processors and common memory to capture those portions of the contents of main memory currently in use. Since cache memories are typically five to ten times faster than main memory, they can reduce the effective memory access time if carefully designed and implemented.

When a processor makes a memory request, it generates the address of the desired word and searches the cache for the reference. If the item is found in cache a *hit* occurs, and a copy is sent to the processors, without a request being made to the main memory (thus taking less time). If the item is not found in cache, a *miss* (or *cache fault)* is generated, and the request must then be passed on to the main memory system. When the item is returned to the processor, a copy is stored in the cache, where room must be found for it. Obviously, cache misses can be very costly in terms of speed. On the other hand, if we are to consider speed as the main criterion, then

using cache memory will be an additional advantage over normal memory [22].

The other advantage of a cache memory is that it may reduce the time the processor must spend waiting for data to arrive from the slower common memory. Memory references are generated by the central processor unit for either instruction or data access. These accesses tend to be clustered in certain regions in time, space, and ordering. The efficiency of a program using cache memory depends, in part, on the locality of reference in the program being run. Given a reasonable amount of locality of reference, for the majority of the time the processor can fetch instructions and operands from cache memory, rather than common memory. Only when the instruction or operand is not in the cache memory must the processor be idle [75].

A parallel matrix multiplication algorithm (*Example 1.*) for the shared memory model is given in section 3.5. The outer loop is done in parallel in this algorithm. The time complexity of this algorithm is as follows. Each processor calculates n/p rows of matrix C; the time needed to calculate a single row is $O(n^2)$ where n is the matrix size and p is number of processors. Hence the complexity of this algorithm is $\frac{n^3}{p} + kp$. Note that since there are only n rows, at most n processes can be used to execute this algorithm.

Developing an efficient matrix multiplication algorithm for the distributed memory model is complicated by the nonhomogeneous memory structure [75].

3.3 Basic Concepts of Parallel Computing

In this section we will define precisely some of the basic terminologies of parallel programming.

- *Process :* A sequence of operations defined by the result it produces or by its purpose. A process is an asynchronous activity such as the execution of a program by the central processing unit [12].

- *Processor :* A piece of hardware, or a combination of hardware, whose function is to interpret and execute instructions. It may be the principal operating part of a computer, in which case it is also known as the central processor. The processor or set of processors in a computer is often called the processing unit [12].

- *Grain Size (or Granularity) :* Grain Size is simply a measure of the amount of computation involved in a software process. The simplest measure is to count the number of instructions in a grain (program segment). Grain sizes are commonly described as *fine, medium,* or *coarse*, depending on the processing levels involved.

- *Latency :* Latency is a time measure of the communication overhead incurred between machine subsystems. For example, the *memory latency* is the time required by a processor to access the memory. The time required for two processes to synchronise with each other is called the *synchronisation latency.* Computational granularity and communication latency are closely related [47].

- *Dataflow Diagram :* An illuminating way to describe an algorithm is to use a dataflow diagram. Such an algorithm consists of nodes that represent data items and directed edges that represent execution dependencies. The diagram is constructed so that there are no data dependencies between leaf data items. That is, there is no contention for write access to a common location between two leaf data items. The dataflow representation of programs differs from a control flow representation in the sense that the edges of the dataflow diagram do not represent processes but the other is a diagrammatic representation of the structure of an algorithm, showing the executions performed by the program and the flow of control.

  The ability to execute program segments in parallel requires each segment to be independent of the other segments. In order to visualise the segments of the programs, we use a type of dataflow diagram for some of algorithms in this thesis. This helps to discover the inherent parallelism and show that parallel execution may be coordinated through the aggregation of this information into a dataflow diagram [22].

- *Parallelism and Load Balancing :* Load balancing is perhaps the central issue of parallel computing. The aim is always that after the initial allocation all processors have nearly the same amount of work and this should be achieved with the smallest overhead possible. If the work is not evenly allocated across the available processors, some processors will be idle. This type of parallel program may not achieve good parallelism

because load imbalance can cause poor efficiency. For example, suppose a parallel computer has 10 processors and is to perform a large matrix multiplication problem, with the matrices divided into 81 partitions. Assuming the partitions are of the same size then, the processors can work in perfect parallelism on 80 partitions but only 1 processor will be active during the remaining 1 partition and 9 processors will do nothing.

Although, in such examples, it is easy to partition the task into many parts these parts may be of widely different sizes. Hence, after an initial allocation of tasks among processors, some processors may finish their tasks much sooner than others.

An alternative way of allocating work is to consider the possibility of a *dynamic balancing of load.* Dynamic load balancing is possibly useful when the sizes of segments are not known initially. This should provide flexibility for the algorithm particularly in a multi-user environment, for, if one processor is held up others will carry out the work instead. On the other hand, suppose a parallel computer has 10 processors and a large matrix multiplication problem, with the matrices divided into 80 partitions. Assuming the partitions are of the same size then, the processors can work in perfect parallelism. In other words, if the number of partitions is divisible by the number of processors, then a *static load balancing* will be efficient. In static load balancing tasks are allocated to processors at the beginning of a computation.

Dongarra et al [22] acknowledge that the importance of load balancing

can be overstated. For example, in a system that is multiprogrammed, the fact that one or more processors are idle should not be of great concern since the idle time (from one user's point of view) can be taken up by another job. This point is of particular importance when the parallelism, measured in number of simultaneously executable tasks, varies during the course of a job. Thus it might be efficient at one time to use all the processors of a system while at another time to use only one or two processors.

- *Contention time :* As shared memory systems have a common memory (as illustrated in figure 3.4) which different processors may wish to use simultaneously, there will be a delay when some processors are waiting for the memory to be free. This delay is called *contention time* [39].

- *Starvation :* Due to the internal structure of an application or due to the difficulty in dividing tasks evenly across the available processors or waiting for data to become available, there may be points in an application's execution where some available processors may not be kept busy and this will result in poor performance. This is called *Starvation* [81].

3.4 Parallel Programing Environment

This section describes the parallel programming environment and introduces inter-thread communications. An environment for parallel programming consists of hardware platforms, i.e. the machine, the operating

system, the language supported (in which the program is to be written), and software tools for data management.

As stated before, the hardware platform used in this work was a bus-connected shared memory Encore Multimax computer running the UMAX operating system. Comparisons were carried out in this thesis using the C++ programming language for various matrix representations including C++ bare arrays, the *newmat* matrix package, and a simple Matrix and Vector class discussed in section 2.6 which was used to access the matrices. The algorithms for QR decomposition suffered failure for large matrices (run-time thread's stacksize) when using the *newmat* matrix package. This is because using the *newmat* matrix package for the manipulation of matrices needs much more space than using either the C++ bare arrays or the simple Matrix and Vector class. This is one reason why the simple Matrix and Vector class was used in preference to the *newmat* matrix package for most of the work described in this thesis. The code was written using the Encore Parallel Threads package (*THREADS*) [31], which provides among other things the facility for the programs to create parallel "THREADs" of execution explicitly using the THREADcreate function (see section 3.5). It also provides mechanisms for synchronisation. The mechanisms used here are "THREADjoin"s, *monitors*, *semaphores*, and *locks* provided in the extension to *THREADS*. They are used to provide mutually exclusive access to shared data.

In a multi-threading environment, a thread is the basic unit of central

processing unit utilisation. It is equivalent to a program stream with an independent program counter operating within each thread. Shared memory programs usually employ a number of medium-grain lightweight processes (called threads), whose number depends primarily on the amount of parallelism exhibited by the algorithm. A single thread executes a portion of a program, cooperating with other threads concurrently executing within the same address space. Like processes, every thread must have a separate program counter and stack of activation records, describing the state of its execution. Usually, threads are lightweight processes that run within the environment defined by a job. A job may have multiple threads with all threads having the same job-sharing capabilities and resources. Considerable overhead is required to create and maintain separate virtual address spaces to support inter-thread communication, switch between threads to effect simultaneous execution, synchronise threads, ensure mutual exclusion, and so on [47].

A large number of threads would be very expensive if implemented inside the operating system kernel [20]. It is fortunate that in the EPT package thread management is controlled out of the kernel and in a thread library so that threads can be implemented in user-space and thread management overhead is small. Although there are conceptual advantages to having more threads than processors in a shared memory program, a medium-grain decomposition also helps in avoiding load imbalance, which can sometimes severely impact performance.

3.5 Overview of the Encore Parallel Threads Package

EPT is a library of routines, which enables a programmer to employ the shared memory and parallel features of the Encore Multimax. It is an extension of the Threads package that was developed by Doeppner at Brown University [31].

The package has many facilities but only some of these have been used in this work. The library provides a programmer with routines at the user level. These can be used to manipulate threads of control and to provide a connection so that threads can share information, thus enabling a program to be parallelised. The threads are very suitable for use in a parallel environment with any number of processors. The EPT routines can be accessed by C++ by using the C linkage convention which is illustrated in the example algorithm (see below). A threads environment is initialised in EPT by calling the function

$$THREADgo(prcs, datasize, func, args, argsize, stacksize, priority).$$

This function provides the facilities for the programmer to specify the number of processors (*prcs*) for use. The parameter *prcs* defines the maximum possible level of true parallelism on the shared memory machine (and is bounded above by 14 in Newton). We can create as many threads as we like, but only *prcs* (number of physical processors) of these will be truly concurrent.

The argument *datasize* sets up a pool of memory or total amount of data

space. The function *func* is initiated as the first thread of execution. The argument *args* indicates the number of parameters needed by the function *func* and *argsize* is the size of storage for the parameters. If *argsize* is 0, *args* is passed to the thread unchanged. However, if *argsize* is nonzero, *argsize* bytes of data pointed to by *args* are placed on the new thread's stack and the thread is passed as a pointer to this location instead of the original value of *args* [31]. The arguments *args* and *argsize* provide the programmer with the necessary parameters to be passed to the function *func*. *stacksize* is the maximum stack size provided for the use of the newly created thread. The latter is given a runtime priority of value *priority*. This function is usually called in the main program.

As we mentioned before, the *THREADgo* function provides a multi-thread environment so that when any thread requires a new thread to be created then this can be achieved by calling *THREADcreate* with the parameters specified as follows

$$THREADcreate(funca, args, argsize, ATTACHED, stacksize, priority).$$

This is equivalent to a Fork operation (i.e. creates a new parallel thread of control). The arguments *func*, *stacksize*, *priority*, *args* and *argsize* have the same meaning as those for the *THREADgo* function. The additional argument in *THREADcreate* is 'attachment'. The 'attachment' determines if the new thread's termination is synchronised with that of the parent thread that created it. If the attachment is *ATTACHED*, this determines that there is a relationship between parent and the child

threads, so that the parent thread will only end when child thread has completed its work, and the parent can execute a call to wait until the child terminates ($THREADjoin$). If $ATTACHED$ is replaced by $DETACHED$, it indicates that the child thread bears no relationship with its parent: it is totally independent. In this thesis we only use $ATTACHED$. The $THREADjoin$ function is used in the parallel task to ensure that the child thread has been completed. This function has no parameters and the program needs as many $THREADjoin's$ as the number of $THREADcreate$ calls made.

The following example uses multiple threads for matrix multiplication which is based on static (scattered) allocation of rows to threads.

```
#include < iostream.h >
#include < math.h >
#include " Matrix.c "
extern " C "
{
   #include " thread.h "
}
struct mulpars
{
  mulpars(int N, int Pr, int Prcs, Matrix& A, Matrix& B, Matrix& C):
        n(N),pr(Pr),prcs(Prcs),a(A),b(B),c(C) { }
  int n;                          // matrix size
  int pr;                         // processors number
  int prcs;                       // number of processors
  Matrix& a;
  Matrix& b;
  Matrix& c;
};
void mult (struct mulpars *params)
```

```
{
    int i, j, k;                        // they are subscripts
    int prcs = params -> prcs;
    int pr = params -> pr;
    int n = params -> n;                // matrix size
    Matrix& a = params -> a;
    Matrix& b = params -> b;
    Matrix& c = params -> c;

     for (i = pr; i <= n; i+ = prcs)
     {
       for (j = 1; j <= n; j++)
       {
         c(i, j) = 0;
         for (k = 1; k <= n; k++)
         c(i, j) = c(i, j) + a(i, k) * b(k, j);
       }
    }
}

void multp (struct mulpars *params)
{
    int pr;
    int prcs = params -> prcs;

     for (pr = 1; pr <= prcs; pr++ )
     {
       params -> pr = pr;
       THREADcreate(mult, params, sizeof(*params), ATTACHED,
                    1024 * 1, 2);
     }
     for (pr = 1; pr <= prcs; pr++ )
     {
       THREADjoin();
     }
}
void startroutine()
{
     int i, j, pr = 1;
     int n = 100;                   // matrix size
```

```
    int prcs = 2;                    // number of processors
    Matrix a(n,n);                   // a, b, and c are all zero
    Matrix b(n,n);
    Matrix c(n,n);
    // set the matrices a and b
    struct mulpars params(n,pr,prcs,a,b,c);
    multp(&params);
}
main ()
{
    THREADgo(prcs, 2 * 1024 * 1024, startroutine, 0, 0, 2 * 1024, 1);
}
```

The *THREADcreate* function offers the user the opportunity for parameters to be passed to their respective functions using the parameters *params* and *sizeof(*params)*. As we explained earlier if in the *THREADgo* function *argsize* is 0, *args* is passed to the thread unchanged. However, if *argsize* is nonzero, *argsize* bytes of data pointed to by *args* are placed on the new thread's stack and the thread is passed a pointer to this location instead of the original value of *args*. This implies that each *THREAD* uses its own copy of the parameters.

A memory problem occured when the matrices or vectors are declared in the parallel part of the programs. This requires allocation of storage space on the heap. In order to avoid this problem matrices or vectors were declared before the parallel section and passed as reference parameters which are defined in *struct*. When necessary they are referenced from within the *struct*.

3.6 Inter-Thread Communication

A common problem occurs when two or more concurrent threads share data which is modifiable. When a thread is updating variables, it is generally unreasonable to allow any other thread to access the same variables. If

a thread is allowed to access a set of variables which are being updated by another thread concurrently, erroneous results are likely to occur in the computation. Therefore, controlled access to the shared variables is required to guarantee that a process has mutually exclusive access to the section of program in which the data is modified. Such segments of programs are called *critical sections* [20]. When one thread is already in a critical section, all other threads wanting to access the critical section must wait. When the thread in the critical section has finished performing the task it can set the critical section free and another thread takes over.

The EPT package also provides a number of mechanisms for synchronisation which entails a thread suspending its own execution, usually waiting for some other thread to cause its execution to resume [20].

Synchronisation has two uses: to constrain the ordering of events and to control interference. The most primitive synchronisation mechanisms of a shared-memory computational model are semaphores and locks. The mechanisms used in our study are semaphores, monitors, and locks. Locks are not provided by the EPT package itself, but are available as an extension code. The simplest forms of these synchronisation mechanisms are used to provide mutually exclusive access to a particular object or data structure (shared data) i.e. the execution can only be performed by a single thread at any one time.

3.6.1 Locks

In practice, a lock is most often used because it is simple. The locks synchronisation implementation uses busy waiting. This is called a *spin-lock* (that is, a thread repeatedly tests the value of a shared variable). However, locks waste processor cycles during delays. Moreover, a wait may generate extra communication traffic and consume extra time in the

network, slowing other processors doing useful work, including one working in a critical section. Reducing the amount of busy waiting is a major issue for synchronisation efficiency. Busy-waiting is a technique implemented such that a waiting thread continues to use processor cycles to test the value of the synchronisation variable until it assumes a desired value.

A lock prevents a thread from entering a critical section while another thread is accessing that section, so that the newly arrived thread waits. The critical section provides programs with a means of ensuring that shared variables are accessed by only one thread at a time. When a thread gets the lock, it sets the lock busy immediately. A thread requesting access to the section must wait until the current thread releases that section. When a thread leaves a critical section then it sets the lock to "unlock", and a waiting thread is allowed to enter. The program uses the *lock* and *unlock* operations to provide mutually exclusive access to a particular object or shared data.

The lock is used to control entry to and exit from a critical section with busy waiting. A critical section must be executed by only one thread at a time. In general locks are only suitable for short waits.

The simple *lock* and *unlock* synchronisation structure is presented using the following example which uses multiple threads for an alternative version of matrix multiplication. The algorithm illustrates the use of dynamic allocation of rows to THREADs on the shared memory machine.

```
#include < iostream.h >
#include < math.h >
#include " Locks.h "
#include " Matrix.h "
extern " C "
{
```

```
    #include " thread.h "
}
#include " matrix.c "
int prcs;                          // number of processors
struct mulpars
{
    mulpars(int N, Matrix& A, Matrix& B, Matrix& C,
            int * Nxrw, LOCK& Slock) :
            n(N), a(A), b(B), c(C), nxrw(Nxrw), slock(Slock) { }
  int n;                                        // matrix size
  int * nxrw;                                    // next row
  Matrix& a;
  Matrix& b;
  Matrix& c;
  LOCK& slock;
};
void nrow (int row,struct mulpars *pars)
{
    int j, k;                                   // they are subscripts
    int n = pars -> n;                          // matrix size
    Matrix& a = pars -> a;
    Matrix& b = pars -> b;
    Matrix& c = pars -> c;

     for (j = 1; j <= n; j++)
     {
        c(row, j) = 0;
        for (k = 1; k <= n; k++)
        c(row, j) = c(row, j) + a(row, k) * b(k, j);
     }
}
void modrow (struct mulpars *pars)
{
    int row;
    int n = pars -> n;
    do
    {
```

```
        pars- > slock.lock();
        row = *(pars- > nxrw);
        *(pars- > nxrw) = row + 1;
        pars- > slock.unlock();

        if (row <= n)
        {
          nrow(row, pars);
        }
    }
    while (row < n);
}

void matmulp (struct mulpars *params)
{
    int prcount;                          // processors count
    for (prcount = 1; prcount <= prcs; prcount + +)
    {
      THREADcreate(modrow,params,sizeof(*params),ATTACHED,
                                              1 * 1024, 2);
    }
    for (prcount = 1; prcount <= prcs; prcount + +)
    {
      THREADjoin();
    }
}

void startroutine()
{
    int i, j;
    int n = 100;                          // matrix size
    Matrix a(n, n);                       // a, b, and c are all zero
    Matrix b(n, n);
    Matrix c(n, n);
    int nxrw = 1
    // set the matrices a and b
    int nxrw = 1;
    LOCK slock;

    struct mulpars params(n, a, b, c, &nxrw, slock);
    matmulp(&params);
}

main ()
```

```
{
    prcs = 2;                          // number of processors
    THREADgo(prcs,2 * 1024 * 1024,startroutine,0, 0, 2 * 1024, 1);
}
```

3.6.2 Semaphores

A semaphore is a synchronisation mechanism which provides an alternative way of obtaining mutual exclusion. As originally proposed, these operations were first developed by E. W. Dijkstra in the mid-1960s [19]. The only logical operations on semaphores are *wait* and *signal* (P and V respectively). The operations P and V come from abbreviations of the Dutch words for waiting and signaling.

A semaphore is a shared integer variable that may only be accessed using one of three possible operations. These are $THREADseminit$, $THREADpsem$, and $THREADvsem$. The last two functions perform the corresponding P and V primary operations on semaphores. The first function is written in the form

$$sem = THREADseminit(initialvalue)$$

and creates a new semaphore, initialises the value of *initialvalue*, and returns a reference to the created semaphore. The $THREADpsem$ operation waits until the semaphore has a positive value and then decrements that value. If the value is zero or negative then the semaphore suspends the calling thread, thus producing a waiting operation, and places it in a waiting queue. Otherwise the thread continues and decrements the *initialvalue*. The function is written in the form

$$THREADpsem(sem).$$

A *THREADvsem* operation increments the *initialvalue*, and a *THREADpsem* operation waits until the *initialvalue* is greater than zero and then decrements the *initialvalue*. *THREADpsem* operations are typically used to wait (synchronise) until some condition is true (such as a shared buffer becoming non-empty), and *THREADvsem* operations typically signal that some condition is now true. The system has to ensure that each of these operations execute atomically. This means that if a wait and signal operation occur simultaneously they are executed one at a time. The programmer has no control over this and does not know in what order they are executed.

The semaphores implement synchronisation at a higher level than the busy waiting that is used to control entry to and exit from a critical section without busy waiting. This includes the appropriate queues and permits groups of threads to enter a critical section at any one time.

The following example uses multiple threads for addition of the element of a vector (using static allocation of work to threads).

```
#include < iostream.h >
#include < math.h >
#include " Matrix.h "
extern " C "
{
    #include " thread.h "
}
SEMAPHORE sem;
struct addpars
{
  addpars(int N, int Pr, int Prcs, ColumnVector& X, double& s):
        n(N), pr(Pr), prcs(Prcs), x(X), s(S) { }
  int n;                          // vector size
  int pr;                         // processors number
```

```
    int prcs;                          // number of processors
    double& s;
    ColumnVector& x;
};
void add (struct addpars *pars)
{
    double sa = 0;
    int i;                             // this is a subscript
    int prcs = pars -> prcs;
    int pr = pars -> pr;
    int n = pars -> n;                 // vector size
    double& s = pars -> s;
    ColumnVector& x = pars -> x;

    for (i = pr; i <= n; i+ = prcs)
    sa+ = x(i);
    THREADpsem(sem);
    s+ = sa;
    THREADvsem(sem);
}

void addp (struct addpars *params)
{
    int pr;
    sem = THREADseminit(0);
    for (pr = 1; pr <= prcs; pr++ )
    {
      params -> pr = pr;
      THREADcreate(add, params, sizeof(*params), ATTACHED,
                   1024 * 1, 2);
    }
    for (pr = 1; pr <= prcs; pr++ )
    {
      THREADjoin();
    }
}
void startroutine()
{
    int i, j, prcs, pr = 1;
    double s = 0;
```

```
    int n = 100;                          // vector size
    ColumnVector x(n);                    // x is zero
    for (i = 1; i <= n; i++)
    x(i) = i;                             // for example
    struct mulpars params(n, pr, prcs, x, s);
    addp(&params);
}
main ()
{
    int prcs = 2;
    THREADgo(prcs, 2 * 1024 * 1024, startroutine, 0, 0, 2 * 1024, 1);
}
```

3.6.3 Monitors

A monitor is a synchronisation mechanism that attempts to encapsulate mutual exclusion and provides convenient facilities for signaling and waking up threads. In this thesis we have not looked at the general structure of a monitor and examined the facilities it offers. Monitors are just used as an alternative mechanism to semaphores in the EPT package. This mechanism was originally proposed by C. A. R. Hoare in the early 1970s [45], and was implemented in the Concurrent Pascal programming language.

A monitor consists of a set of variables representing the state of some resource and a set of functions. When a thread requires to use a monitor, it must create one before using it. This is accomplished by a thread call

$$mon = THREADmonitorinit(conditions, resetfunc).$$

A monitor may provide condition variables (*conditions*) each of which have associated suspend and continue operations. The parameter *resetfunc* (reset function) is used for orderly reorganisation of the monitor should the thread be terminated. In its simplest form, the monitor provides exclusive

access to shared data. The required control of access can be accomplished by using the following functions:

$$THREADmonitorentry(mon, manager)$$

and

$$THREADmonitorexit(mon).$$

Only one thread can execute the code between the $THREADmonitorentry$ and $THREADmonitorexit$ functions at one time. Thus, if thread T_1 has invoked one of the monitor entry functions and thread T_2 attempts to invoke a function in the same monitor, T_2 will be blocked until T_1 relinquishes the monitor. Only one thread may be inside the monitor (i.e. executing a monitor function) at any point in time. The first parameter *mon* is the handler of the monitor that has been created to protect the data structure. The second parameter, *manager* gives the caller the option of managing the monitor control block space and *NULL* gives default action. The parameter *manager* is a pointer to an area of opaque type $THREAD\_MANAGER\_BLOCK$. This storage needs a lifetime long enough to still exist when $THREADmonitorexit(mon)$ is called.

Monitors also provide *wait* and *signal* operations in the following way. If a thread enters a monitor and finds that a required condition is not true, it can suspend itself by executing a wait statement of the form

$$THREADmonitorwait(mon, condition).$$

The function has two parameters which are *mon* and *condition*. *mon* is the name given to the monitor. The *condition* argument gives the number of condition queues (inside the monitor) to be created. The $THREADmonitorwait$ function removes the thread from the monitor and places it on a queue waiting for the condition to be signaled. When this

new thread enters the monitor and changes the condition to true then it can execute a signal statement of the form

$$THREADmonitorsignalandexit(mon, condition).$$

This function withdraws a waiting thread from the condition's queue and wakes it up. If no threads are waiting on the condition queue, the thread simply continues and the caller exits the monitor.

The constraint imposed by the monitor is that only one thread shall be inside a monitor procedure at any one time. An important feature of monitors and semaphores is that they do not use busy waiting. In general semaphores and monitors are more appropriate for longer waits.

We have considered a number of parallel algorithms for comparing the three different synchronisation mechanisms mentioned above when applied to a particular problem. When we attempted to simulate the *event* (see chapter 4 for details) synchronisation mechanism with monitors, we observe that this mechanism slowed down the performance of the implementation. This is one reason why the monitors and semaphores were not used for most of the work in this thesis. The results are discussed in chapter 5.

CHAPTER 4

Direct Solution of Linear Equations

4.1 Introduction

In this Chapter, we consider a number of different parallel algorithms for the QR and LU decomposition of a square matrix A. The first problem tackled was the decomposition of a square matrix into a product of an orthogonal matrix Q and upper triangular matrix R (i.e. QR decomposition) and its use for the solution of sets of linear algebraic equations. Wright [97] has described algorithms for this problem implemented using PASCAL with the Encore multi-tasking library and Encore Parallel Fortran (EPF) which is an extension to FORTRAN77 including parallel constructs. The work here is to consider implementation of these algorithms in C++ with the Encore Parallel Threads package [31]. Following the work in [97] a further version was also investigated using the *event* synchronisation facility in EPF where if a pivotal column is not yet available the process just waits until it is. This algorithm was also implemented in C++ with a simulation of *event* synchronisation as THREADS does not provide this facility.

The second problem tackled was the decomposition of a square matrix into a product of a lower triangular matrix L and an upper triangular matrix U with matrix L or U unit diagonal. (i.e. LU decomposition). Precisely the same strategy in QR (Householder transformations) decomposition can be used for LU decomposition if no row interchanges are carried out, and the single processor version is then equivalent to the GAXPY Gaussian elimination described in [40]. For LU decomposition with interchanges the GAXPY version described [40] has updates to columns which are not independent because the whole row is interchanged together, so that the interchanges are applied to the pivotal information stored in the lower triangle of the matrix. However, the interchange strategy may be modified so that first, the multipliers in L are not interchanged and secondly, interchanges in U are delayed until just before a column update. This strategy recovers the column independence, so that the same parallel strategies used for the QR decomposition can be applied. This has the minor disadvantage that element updates cannot be carried out using scalar products (as in the Crout algorithm) as the modifications to an element are interleaved with interchanges. This possibility is mentioned by Gallivan et al [34], but does not seem to have been investigated further.

Algorithms based on both Givens and Householder transformations are considered for QR decomposition in the section 4.2. For the LU decomposition we consider methods using both a unit lower triangular matrix L and a general upper triangular matrix U, and a unit upper triangular matrix and a general lower triangular matrix. These algorithms

are considered in section 4.4. The former is similar to Doolittle reduction, the *jki* forms of LU decomposition [39,23], and GAXPY Gaussian elimination [40] and the later is similar to Crout reduction.

4.2 Sequential Algorithms for QR Decomposition

We considered the solution of the system of linear equations

$$Ax = b, \tag{4.2.1}$$

for the n-vector x, with A an $n \times n$ matrix and b an n-vector. The aim of the QR decomposition is to transform the system (4.2.1) into one which is easy to solve. Specifically, the aim is to determine a matrix Q and an upper triangular matrix R such that

$$A = QR,$$

with Q orthogonal, that is, $Q^TQ = I_n$. If we replace A in (4.2.1) by QR and premultiply by Q^T, then, using the orthogonality of Q, it follows that

$$Rx = Q^Tb. \tag{4.2.2}$$

Having determined the QR decomposition of A, the solution of (4.2.1) therefore requires multiplication of the right-hand side vector b by Q^T and, backward substitution to solve the upper triangular system (4.2.2). Note that the Q is not usually obtained explicitly (in this work we did not accumulate the orthogonal transformations Q).

Widely used methods for the QR decomposition of a matrix are Householder transformations, Givens transformations, or Gram-Schmidt orthogonalisation. Although there are some similarities between the Gram-Schmidt orthogonalisation and Householder orthogonalisation, there are also some important differences. An attractive feature of the Gram-Schmidt process is its speed, if Q is required explicitly as it is twice as fast as Householder's algorithm since the columns of Q are computed directly. In Householder's method, Q is the product of Householder matrices. As we mentioned that we did not need to accumulate the orthogonal transformations Q then the Gram-Schmidt method is not preferable for this problem so that the Gram-Schmidt method is not discussed any further here. The QR algorithms based on Givens transformations require about twice the number of arithmetic operations as the algorithms using Householder transformations [34].

4.2.1 Sequential Algorithm: Givens

The Givens process works by taking linear combination of rows of the matrix, chosen to make the new elements below the diagonal zero. The basic computation involves the premultiplication of A by an (i, j) transformation matrix G_{ij} which just changes the coefficient in row i and row j. A Givens transformation is defined by a 2×2 orthogonal matrix of the form

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

where $c^2 + s^2 = 1$. If a $2 \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \end{pmatrix}$$

is premultiplied by G, a zero can be introduced into the a_{21} position. Each time a Givens transformation is applied, there is a requirement for data exchange between the corresponding two rows of the array. The Givens process works by taking linear combinations of rows of the matrix, chosen to make the new elements below the diagonal zero [97].

The Givens algorithm may be written:

for $i = 1, \dots, n$
  for $j = i + 1, \dots, n$
    $A(i, i:n)^{(new)} = c * A(i, i:n) - s * A(j, i:n)$
    $A(j, i:n)^{(new)} = s * A(i, i:n) + c * A(j, i:n)$
  endforj
endfori

where $c = A_{ii}/d$, $s = -A_{ji}/d$, and $d^2 = A_{ii}^2 + A_{ji}^2$. A similar transformation is applied to the right-hand side b.

4.2.2 Sequential Algorithm: Householder

The QR decomposition of a matrix A is computed using a sequence of Householder matrices to reduce A to upper triangular form. The reduction requires $n - 1$ Householder transformations. A Householder transformation uses a matrix of the form $H = I - 2ww^T$ where $w^T w = 1$, that is w is a vector with Euclidean length one. The resulting upper triangular matrix is R and the product of the Householder matrices is Q.

In the first step we premultiply A by the Householder matrix H_1 to annihilate elements 2 to n in the first column of A, giving $A_1 = H_1 A$. In the second step we premultiply A_1 by the Householder matrix H_2 to annihilate elements 3 to n in the second column of A_1. In the k^{th} step, we premultiply A_{k-1} by the Householder matrix H_k to annihilate elements $k+1$ to n in the k^th column of A_{k-1}. The Householder matrix H_k has the form

$$H_k = \begin{pmatrix} I & \\ & H \end{pmatrix}$$

where I is a $(k-1) \times (k-1)$ unit matrix and H is an appropriately sized Householder matrix. Step k generates $A_k = H_k A_{k-1}$. After $n-1$ steps, we arrive at R:

$$R = A_{n-1} = H_{n-1}A_{n-2} = H_{n-1}H_{n-2}A_{n-3} = ... = H_{n-1}H_{n-2}...H_1 A$$

finally, $A = QR$, where

$$Q = H_1 H_2 ... H_{n-1}$$

since $H_i^{-1} = H_i^T = H_i$, $i = 1, 2, \ldots, n-1$.

Let $a = (a_1, a_2, \ldots, a_n)^T$ be a vector such that not all of the entries $a_2, a_3, \ldots, a_n$ are zero, and suppose we want to transform a to a vector $h = (h_1, h_2, \ldots, h_n)^T$ where $h = Ha$ with H orthogonal and such that $h_2 = h_3 = \ldots = h_n = 0$. Define $h = (\sigma\ 0\ \ldots\ 0)^T$ then $|\sigma| = \|h\|_2 = \|a\|_2$ because H is chosen an orthogonal matrix.

H can be chosen to be a Householder transformation matrix given in the form $H = I - \gamma u u^T$, where $u = a - h = (a_1 + \sigma, a_2, a_3, \ldots, a_n)^T$ and $\gamma = 2/\|u\|_2^2$. We specified that $\sigma = \pm\|a\|_2$, but we did not specify the sign. In theory either choice works, but in practice σ should be chosen so that its sign is the same as that of a_1. This ensures that cancellation cannot occur in the calculation of $a_1 + \sigma$.

The following algorithm finds the QR factors for an $n \times n$ matrix A, overwriting the coefficient matrix with both R and the vectors characterising each Householder matrix. Since all the nonzero elements of w do not fit within the space created by the annihilated coefficients, we store the diagonal of R in a separate array d to make room for the first nonzero component of w. This process modifies each pivotal column, headed by a_{kk} for $k = 1, \ldots, n$ followed by updates to the later columns. The algorithm takes the form:

for $k = 1 : n-1$
  $\sigma = \sqrt{a_{kk}^2 + a_{k+1k}^2 + \ldots + a_{nk}^2}$
  if $\sigma = 0$
    $d(k) = 0$
  else
    $t = A(k,k)$
    $\gamma(k) = 1/(\sigma * (\sigma + |t|))^2$
    if $t < 0$
      $\sigma = -\sigma$,
    $d(k) = -\sigma$
    $A(k,k) = \gamma(k) * (t + \sigma)$
    $A(k:n,k) = \gamma(k) * A(k:n,k)$
    $v(k+1:n)^T = v(k+1:n)^T + A(k:n,k)^T * A(k:n,k+1:n)$
    $A(k:n,k+1:n) = A(k:n,k+1:n) - A(k:n,k) * v(k+1:n)^T$
endfor

This may be considered as modifying sub-columns with heads in the upper triangle of the matrix in the order $(1,1),(1,2),...,(1,n)$, then $(2,2),(2,3),\ldots,(n-1,n)$. The right-hand side can just be treated as an additional column of the matrix.

Before considering possible parallel implementations, the diagram in figure 4.1 illustrates the data dependencies of the Householder QR and LU (column version of LU) decomposition algorithms which are essentially the

same. Both types of algorithm have the same natural parallelism and may be used in a number of different ways. This illustrates that there are a lot of choices in the ordering of updates for these columns.

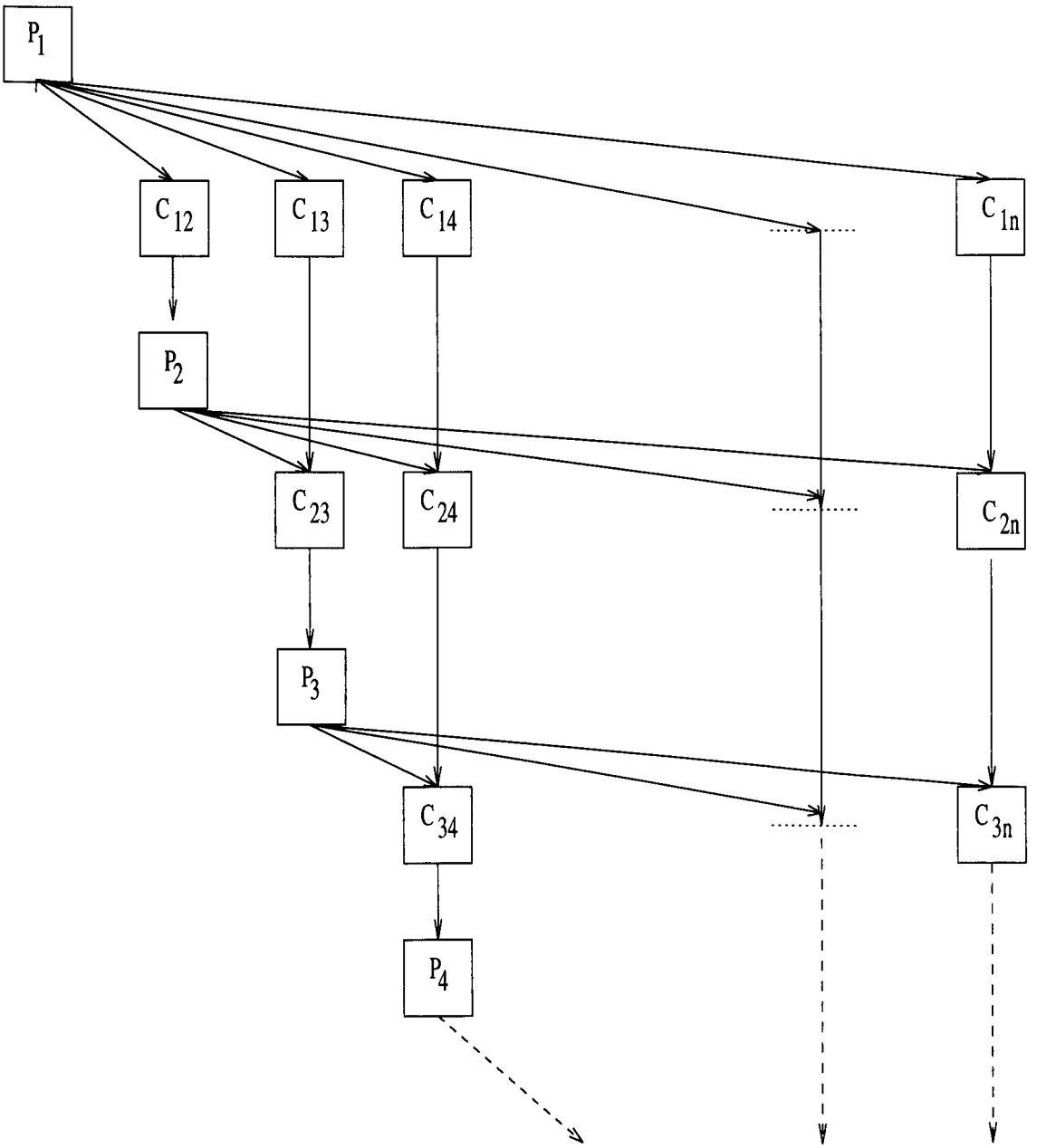


Fig. 4.1 Dataflow Diagram for QR and LU Decomposition

In the Householder algorithm once a pivotal column has been completed

no further changes are made to the column. In the usual Crout and Doolittle algorithms modification may be made because of interchanges taking place corresponding to later pivotal columns. Here the algorithm is modified so that interchanges of the multipliers do not take place, with the algorithms organised in a similar way to the Householder implementations.

4.3 Parallelisation of QR and LU decomposition

In the diagram in figure 4.1 the k^{th} pivotal update is denoted by P_k and with standard column update is denoted by C_{jk} for $(j = k + 1 : n)$. This forms the basis of a simple parallel implementation of the Householder algorithm and a simple parallel implementation of the LU decomposition. In these algorithms modification to the pivotal columns are carried out sequentially and each parallel task is fairly small. This algorithm will be described below.

In [97] a number of parallel implementations for QR decomposition were compared using PASCAL with the Encore multi-tasking library. It was found that one using Householder transformations with multiple updates to columns was very effective. The Householder algorithm considered for QR decomposition and most of the algorithms considered for LU decomposition use a similar idea though one simple implementation is also used for comparison. The diagram in figure 4.1 shows that once a pivotal column and the consecutive column have been completed then the next pivotal column can be dealt with before the remaining columns with heads in the same row as the first pivotal column are modified. These updates are independent

and can hence be carried out in parallel. This observation will be made use of in some algorithms described below for QR decomposition (Householder algorithm) as well as LU decomposition.

In the parallelisation of the QR decomposition and LU decomposition the critical part of the process is the pivotal update which suggests treating this as soon as possible, in particular before all the updates in the current pivotal row. This can be done by ordering the updates working down columns rather than along the rows. This leads to a parallel implementation assigning columns to different parallel tasks. This clearly requires some mechanism to ensure that updates are not carried out before the corresponding pivotal columns are ready. Some ways of achieving this are discussed below in sections 4.3.1 and 4.4.2. Two types of implementation of the Householder algorithm will be discussed. The first is a simple fixed allocation implementation, the other uses dynamic allocation. Both algorithms do precisely the same arithmetic.

4.3.1 Parallel Algorithm: Givens

Several parallel implementations of the Givens method have been introduced in the literature. One implementation of this method has been suggested by Modi [66] and it is adapted to a multiprocessor machine by Wright [97]. Here a simpler but related version of the algorithm is described and this simplified version has been implemented. The ordering of eliminations used in the usual sequential algorithm is not suitable for parallelisation so an alternative ordering is used. Note that

the transformations may be ordered so that any pair of rows can be combined which also produce an upper triangular matrix. The alternative computation considered here is divided up into a number of stages [97]. In the first stage roughly half of the first column is made zero by rotations. For the first column, the row j is taken with row $n - j + 1$ for $j = 1, \ldots, [(n-i+1)/2]$. In the second stage a similar transformation is applied to reduce half of the remaining non-zero rows in the first column and start on the second column. Implementation of the Givens method is done by keeping two indices for each column indicating the first and last rows in this column which are available for processing at each stage. At the end of the stage these indices are updated. Note that at each stage all these transformations are independent of each other, so that they can be allocated to threads in either a pre-determined fashion or dynamically. At the end of each stage synchronisation is required as the new stage cannot start until the previous one is complete. Each stage is terminated by "THREADjoin"s.

4.3.2 Parallel Algorithm: Householder

The first parallel implementation of the Householder algorithm treats the pivotal column sequentially and then updates the later columns with heads in the same row in parallel. All these updates are carried out before the next pivotal column is dealt with. These updates are all independent and so can be performed in parallel. The allocation of column updates to threads was done in a pre-determined (or static) way. There are two obvious ways of doing this i.e. the *blocked* and *scattered* forms. In this implementation

only the scattered form was used, that is at the k^{th} pivotal stage thread j dealt with columns $k+j$, $k+pr+j$, $k+2pr+j$ where pr is the number of threads.

The second method allocated columns dynamically with waiting when necessary, and the waiting was carried out using a simulation of *event* synchronisation using a loop and with a test protected by a *monitor*. This version orders the updates working down columns rather than along the rows and uses *event* synchronisation to ensure that the updates to the columns are carried out in the correct order. For any column, updates are carried out so long as the corresponding pivotal columns have been completed. If they have not then the process waits until the pivotal columns are ready. In this algorithm the columns are allocated in ascending order of column number and the columns are allocated dynamically by keeping a record of the position of the next untreated column. Any processor which finishes processing a column starts on the next one indicated. The variables used to control the allocation of work to threads are only modified in critical regions controlled by *monitors*. If there are no available columns for modification the thread terminates with "THREADjoins".

4.3.3 Experimental Results for QR Decomposition

We have tested Givens and Householder transformation algorithms for QR decomposition using *C++ Data Arrays*, data represented by the matrix class type defined in the *Newmat Matrix Package*, and the *Matrix Class* defined in chapter 2. The results are obtained with only row representation

for all the matrix representation types. For simplicity we only measured the time for the computation of the triangular factor R. The interval timer was used in order to compare the efficiency of the algorithms. The results are also obtained with and without use the *inline function* included for the array subscripting of the matrix class (see section 2.6 in chapter 2 for details).

An initial empirical comparison was carried out using four different versions based on the algorithms described above. The two Householder transformation algorithms using C++ Data Arrays were *HouA* and *HoudA* versions using the Matrix Class were *HouC* and *HoudC*, and versions using the Newmat Matrix Package were *HouM* and *HoudM*. In the first version the tasks are allocated to threads in a pre-determined (scattered) ordering and the other version the tasks are allocated to threads dynamically. We also obtained the results from Householder versions using the Matrix Class with array bound checking *(cHouC* and *cHoudC)* in order to provide a more appropriate comparison with the results using the Newmat Matrix Package. The other algorithm was based on Givens transformations, using C++ Data Arrays *(GivA)*, Matrix Class *(GivC)* without array bound checking, and Newmat Matrix Package *(GivM)*, with pre-determined allocation.

Times were obtained for the two different algorithms and the four matrix environments using one to eight processors with matrices of sizes 40(40)200. In addition to the raw times, values were obtained for the efficiency by comparing parallel versions with a sequential version of the Householder algorithm ($HouA$) for Householder versions and with a sequential version

of the Givens algorithm ($GivA$) for Givens versions using C++ data arrays. These sequential times were better than the other implementations. These comparisons were made in order to compare the algorithms under different conditions, matrix representations, and to make use of the C++ programming language as far as possible.

As pointed out in chapter 2, C++ provides the *inline* feature to avoid function call overhead. To investigate the effect of using *inline* function calls for array subscripting, we used the two Householder transformation algorithms using the Matrix Class $HouC$ and $HoudC$. The results are illustrated in table 4.1 which gives the measured elapsed times. The subscript indicates whether the *inline* function was used or not.

Sequential Time				
n	$HouC_{inline}$	$HouC_{noinline}$	$HoudC_{inline}$	$HoudC_{noinline}$
40	0.523	0.593	0.565	0.700
80	3.693	4.667	4.352	5.121
120	12.094	16.253	14.463	16.849
160	35.153	39.366	35.633	37.465
200	61.573	77.347	65.394	76.302

Table 4.1 Matrix Class with and without Inline Function.

Tables 4.1 and 4.2 show the sequential and parallel times for the two versions. It is obvious from the table that using *inline* functions with the Matrix Class is significantly better than not doing so using both Householder transformation versions. The remaining results all use the *inline* function with the Matrix Class.

Parallel Time for 2 processors				
n	$HouC_{inline}$	$HouC_{noinline}$	$HoudC_{inline}$	$HoudC_{noinline}$
40	0.568	0.613	0.434	0.503
80	2.727	3.221	2.600	3.108
120	8.378	9.184	7.834	9.055
160	19.131	21.876	17.882	21.945
200	37.641	42.341	37.864	44.374

Table 4.2 Matrix Class with and without Inline Function.

Figures 4.2 and 4.3 display plots of mean efficiency against number of processors for different implementations using C++ Data Arrays, Matrix Class, Newmat Matrix Package. There is no column matrix representation for C++ Data Arrays and Newmat Matrix Package so that the time is not obtained for the Matrix Class using the column representation. In this section the times are obtained for the efficiency with a row matrix representation. Figure 4.4 shows actual efficiencies using 2 processors and figure 4.5 shows similar plots for 6 processors for different sized matrices.

The results generally confirm expectations, particularly that the

Householder implementations with dynamic allocation give better performance than the simple implementation.

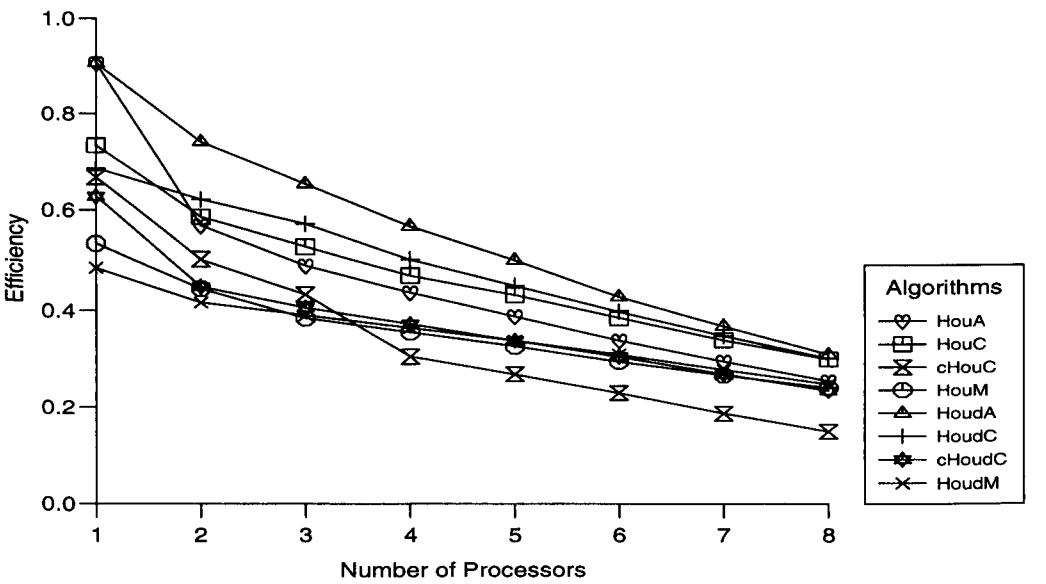


Fig. 4.2 Mean Efficiency Graph: Householder Transformations

In [97] a number of algorithms for QR decomposition were compared and it was observed that the efficiency of the algorithms is affected by the organisation of data. Also [97] has illustrated the difference between the use of the normal representation of a matrix (i.e. row matrix representation) and its transpose (i.e. column matrix representation) with the Householder implementations. Using the Matrix Class was particularly convenient for such comparisons as the Matrix Class could be altered internally to give storage of the matrix by rows or by columns (we have not used column matrix representation for QR decomposition) and to include versions with

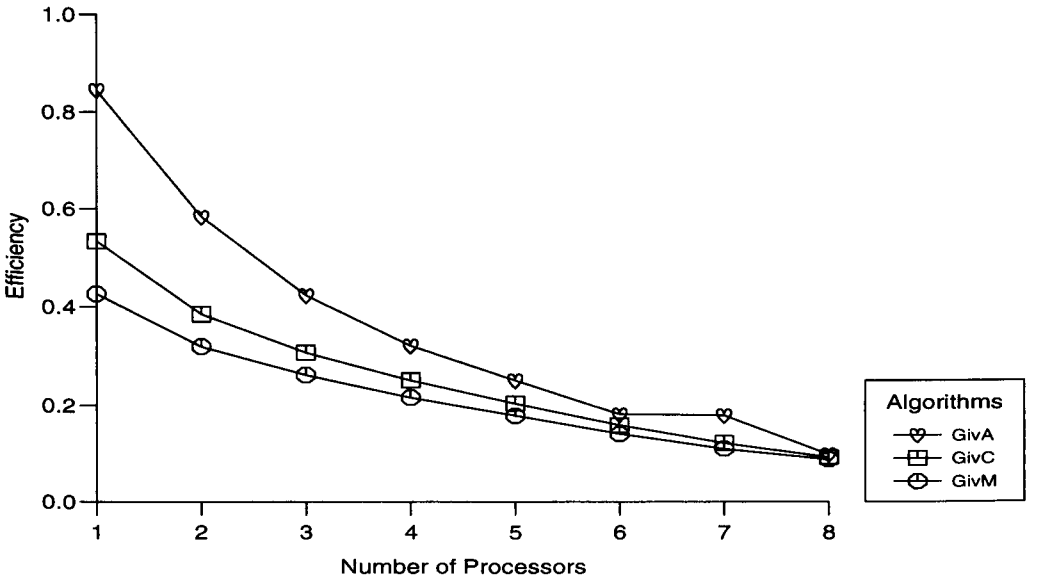


Fig. 4.3 Mean Efficiency Graph: Givens Transformations

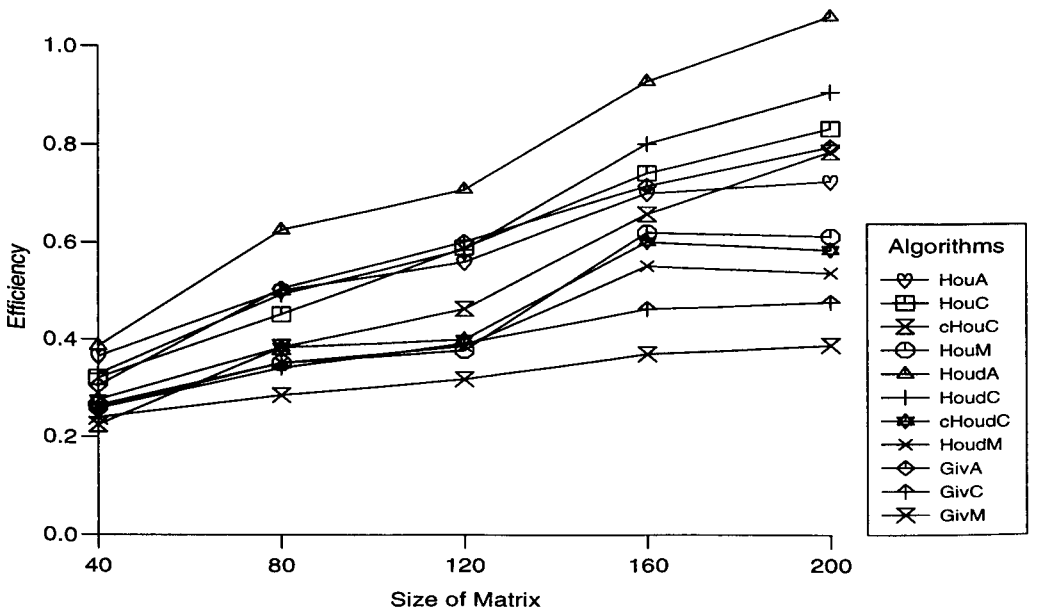


Fig. 4.4 Efficiency Graph for 2 Processors: QR Decomposition

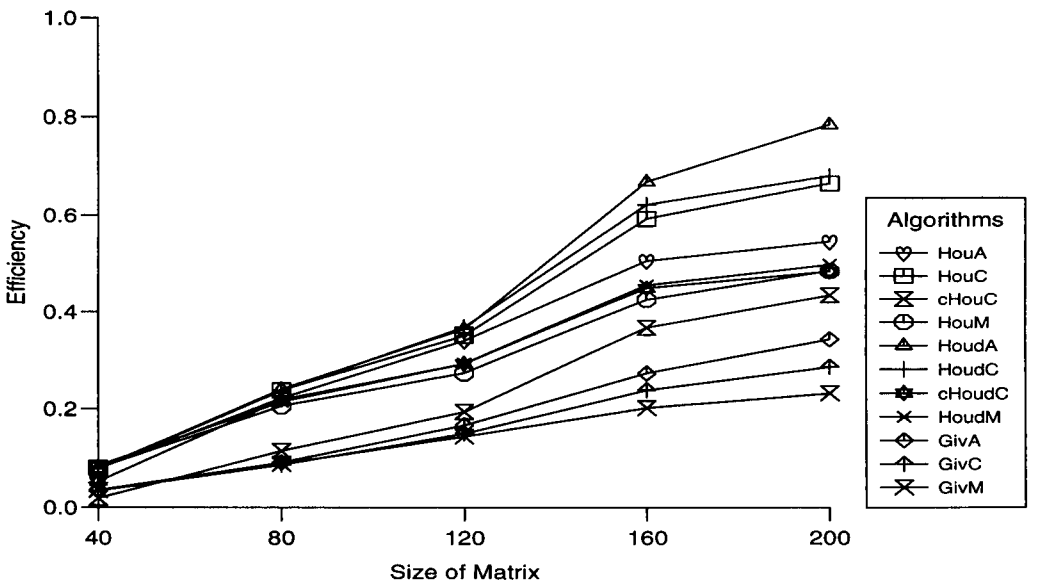


Fig. 4.5 Efficiency Graph for 6 Processors: QR Decomposition

or without array bound checking.

4.3.4 Conclusion for QR Decomposition

We have compared the *C++ Data Arrays, Matrix Class,* and the *Newmat Matrix Package* representation in the testing of QR decomposition using the Householder transformations and the Givens transformations. A very significant factor affecting the times seems to be the representation of the matrix (i.e. C++ Data Arrays, Newmat Matrix Package and Matrix Class). These versions using the *Newmat* Matrix Package were significantly slower than those using C++ Data Arrays or Matrix Class without array bound checking. This is also expected since in the *Newmat* Matrix Package arrays involve array bound checking as well as access via functions. On the

other hand, these versions using the *Newmat* Matrix Package were slightly better than those using Matrix Class with array bound checking particularly for two processors.

The Householder method was generally faster than the Givens method using C++ Data Arrays, *Newmat* Matrix Package, and Matrix Class. This is expected from the operations count. There is a clear conclusion that the efficiency curves for the dynamic allocation Householder transformations using C++ Data Arrays ($HoudA$) and Matrix Class ($HoudC$) are better than those for the other representations. The efficiency curve for the pre-determined allocation Householder transformations using Matrix Class ($HouC$) is also marginally better than other implementations and in most cases it is quite close to curves of the $HoudA$ and $HoudC$ versions. As the size of matrix increases, the graphs for all the versions rise rapidly with matrix size for 2 processors as well as for 6 processors. These results are shown in figures 4.4 and 4.5.

The only differences between the implementations $HouA$ and $HouC$ is in the representations (i.e. the C++ Data Arrays or Matrix Class) since in both versions the way of allocation of the columns to threads is the same. In spite of this $HouC$ is more efficient than $HouA$ which is surprising because the $HouC$ version involves more work than the $HouA$ version as $HouC$ accesses the matrix elements via a function. C++ Data Arrays subscripting should save time for reading and writing in data but additional waiting during the process may be caused by contention for access to the shared memory. As

we mentioned in chapter 1, the lack of direct control over the location of data causes some difficulty in investigating this because the transfer of data between shared and cache memory is controlled by the hardware.

Surprisingly, the dynamic allocation Householder version ($HoudM$) using the *Newmat* Matrix Package has poorer efficiency than the pre-determined allocation version ($HouM$) for 2 processors. When the number of processors and the size of the matrices is large, the efficiency for the $HoudM$ version is marginally better than that for $HouM$ for 6 processors. This is shown in figure 4.4.

Other important remarks follow from the comparison between the results using the Newmat Matrix Package and the Matrix Class with array bound checking. The results show that the $HouM$ version gave significantly better result than the $cHouC$ version. On the other hand, the $HoudM$ and $cHoudC$ versions have only small differences in these efficiency curves for more than 2 processors. Nevertheless the efficiency of the $cHoudC$ version is slightly better than $HoudM$ for 1 and 2 processors. These can be seen in figure 4.2.

For all versions, the larger the matrix size the better the efficiency, particularly for larger numbers of processors. Comparing the results for these versions shows that the $HoudA$ and $HoudC$ versions come out better than the others, in most cases.

4.4 Sequential Algorithms for LU Decomposition

We shall now consider LU decomposition. The sequential algorithm producing a unit upper triangular Matrix is outlined below. To facilitate later parallel implementation a number of modifications to the standard Crout algorithm are introduced. The pivotal column is treated as in the Crout algorithm but, following this, updates are applied only to the immediately succeeding column, all the updates to this column being delayed so that they can be performed together. Once a pivotal column has been updated no further changes are made to this column. In particular no interchanges are carried out on the pivotal column at this stage. The index $piv(j)$ indicates the row chosen as pivot at the j^{th} pivotal stage.

The summary of the algorithm may be written:

for $j = 1 : n$
  for $i = 1 : j - 1$
    $A(i, j) \leftrightarrow A(piv(i), j)$
    $A(i, j) = A(i, j)/A(i, i)$
    $A(i + 1 : n, j) = A(i + 1 : n, j) - A(i + 1 : n, i)A(i, j)$
  endfor
  Find p with $j \leq p \leq n$ so $\mid A(p, j) \mid = \parallel A(j : n, j) \parallel_\infty$
  $A(j, j) \leftrightarrow A(p, j)$
  $piv(j) = p$
endfor

Note that the forward substitution applied to the right-hand side is carried out after the decomposition phase but the order is just the same as that for a column of the matrix with the interchanges interleaved with the elimination steps. As these steps are interleaved it follows that the L and

U matrices produced by this method do not in general satisfy

$$LU = PA$$

for any permutation matrix P. We then carry out forward and backward substitutions to find the solution x.

The algorithm giving a unit lower triangular matrix is equivalent to the GAXPY Gaussian elimination described in [40] apart from the interchange implementation. The structure of the algorithm is similar to that for the unit upper triangular version except that the pivotal column is divided by the pivot instead of the pivotal row.

4.4.1 Parallel implementations for LU Decomposition

We consider four types of implementation, applicable to both upper and lower unit triangle versions. The first two are based on *event* synchronisation to ensure that the updates to the columns are carried out in the correct order. For any column, updates are carried out so long as the corresponding pivotal columns have been completed. If they have not, then the process waits until the pivotal columns are ready. In both algorithms the columns are allocated in ascending order of column number. In the first algorithm the columns are allocated dynamically by keeping the position of the next untreated column. Any processor which finishes processing a column then starts on the next one indicated. In the second method the columns are allocated in a predetermined (scattered) fashion.

In the third algorithm processes avoid waiting for synchronisation by

starting treatment of a new column when further progress in the column currently being updated is not possible. The columns are allocated dynamically to the processors by keeping an index for each column indicating how many updates have been carried out, along with an indicator of whether the column is currently being processed. When a thread finds that no more updates to a column can be performed, a search is carried out for these indices to find the first column which is not being processed and where the updates can be performed. As with the other implementations there is an index giving the position of the next pivotal column to be processed.

In [97] a number of algorithms for QR decomposition were compared and ones using Householder transformations with multiple updates to columns were found to be very effective. Most of the algorithms considered here for LU decomposition use a similar idea though one simple implementation is also used for comparison.

The fourth algorithm is a simple implementation carrying out all updates corresponding to a pivotal column together in parallel, with the pivotal columns treated sequentially. This is to provide a basis for comparison for the other algorithms.

All four algorithms do precisely the same arithmetic, so that even the rounding errors will be the same.

In addition to the algorithms described above which access the elements of the matrix directly, variants of the algorithms were used in which the columns of the matrix were copied to a local vector before processing and

then copied back when all the updates currently possible had been carried out. The tests were carried out using Matrix class with both row and column representations of the matrices.

4.4.2 Experimental Results for LU Decomposition

In this section we present the numerical results (using C++) obtained from seven different parallel versions of the LU decomposition algorithms. These versions are outlined in section 4.4.1. Four of the algorithms use Crout-like reduction and three of the algorithms use Doolittle like reduction.

The results use the notation *cr* for Crout-like and *do* for Doolittle-like algorithms. Implementation one is indicated by *evd* (dynamic event), implementation two by *evs* (static event), implementation three by m (column splitting), and the simple implementation four by s. The use of copying a column to a vector is indicated by v, and the representation of the matrix by columns is indicated by t. Results are only given for the column representation as these were always better than those for the row representation, and so as not to confuse the other comparisons.

We tested the algorithms using from one up to ten processors with matrices of sizes 100(100)500. To illustrate the relative performance of the algorithms graphs of efficiencies are displayed in the figures. For the results without array bound checking a sequential version of the Crout-like reduction algorithm (*crevst*) was used for the sequential time as this produced the best times (better than the standard Crout version). Similarly,

the results with array bound checking used a sequential version of the Doolittle-like reduction algorithm (*doevst*) for the sequential times as this gave the best overall performance in this case. The parallel and sequential times were obtained for the same sized matrices and the same environment.

Figures 4.6, 4.7, 4.8 and 4.9 display plots of mean efficiencies for the over the matrix sizes against number of processors for different algorithms. Figures 4.10 and 4.11 show actual efficiencies using 2 processors, and figures 4.12 and 4.13 present similar plots for 6 processors. Figures 4.14 and 4.15 display results for different sized matrices and different numbers of processors for the Crout-like column splitting algorithms *crmvt* and *crmt*, the first of which turned out to be the best algorithm overall.

The results of the experiments are generally as expected, in particular that the simple parallel implementation is significantly slower than all other versions. More surprising is the finding that the column copying versions of all the algorithms are significantly better than the versions without copying. It is perhaps also surprising that with processor number increase the efficiencies decrease more rapidly when checking is used than when it is not. The other differences between the implementations are relatively small, though method three, the column copying Crout-like implementation (*crmvt*), comes out best. However, without column copying, this algorithm is not so good as the other versions.

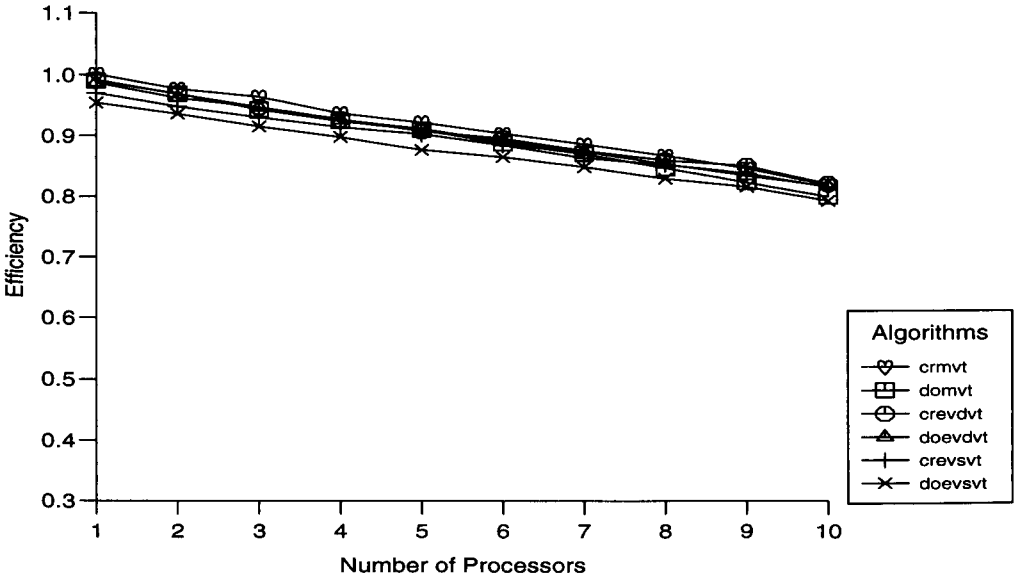


Fig. 4.6 Mean Efficiency Graph: LU Dec. No Check and Copying Vector

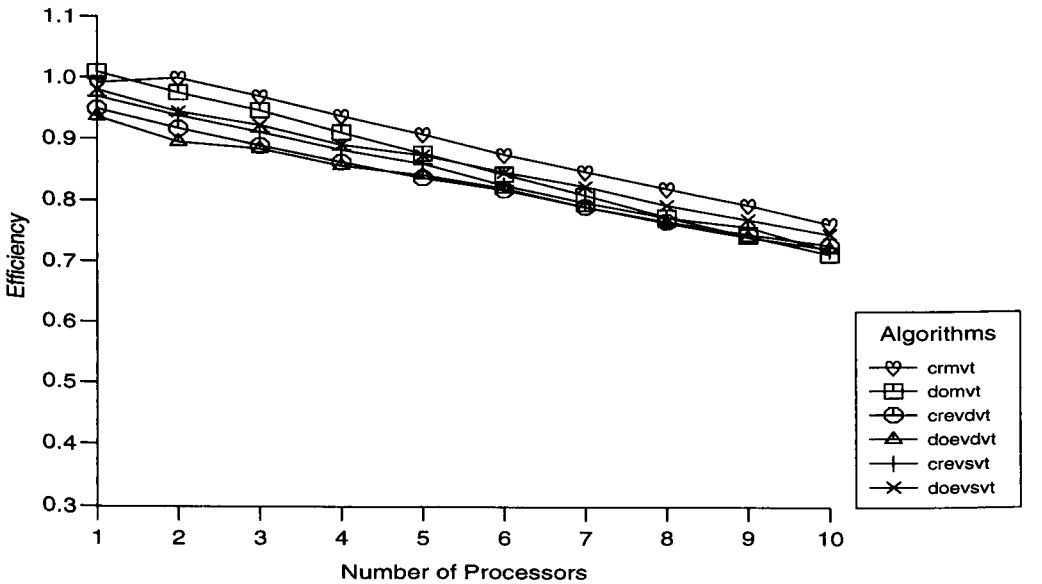


Fig. 4.7 Mean Efficiency Graph: LU Dec. Check and Copying Vector

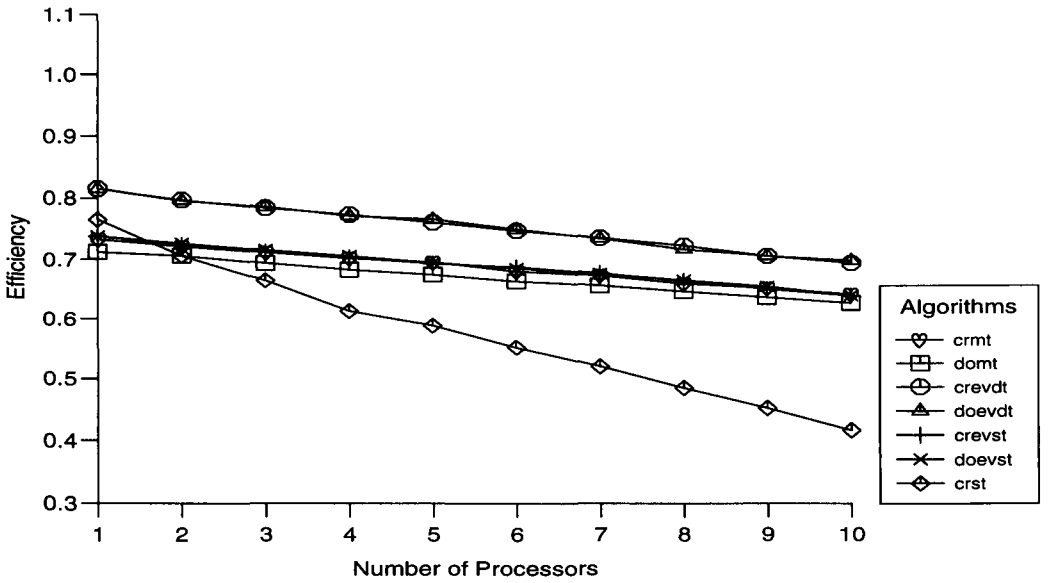


Fig. 4.8 Mean Efficiency Graph: LU Dec. No Check and No Copying Vector

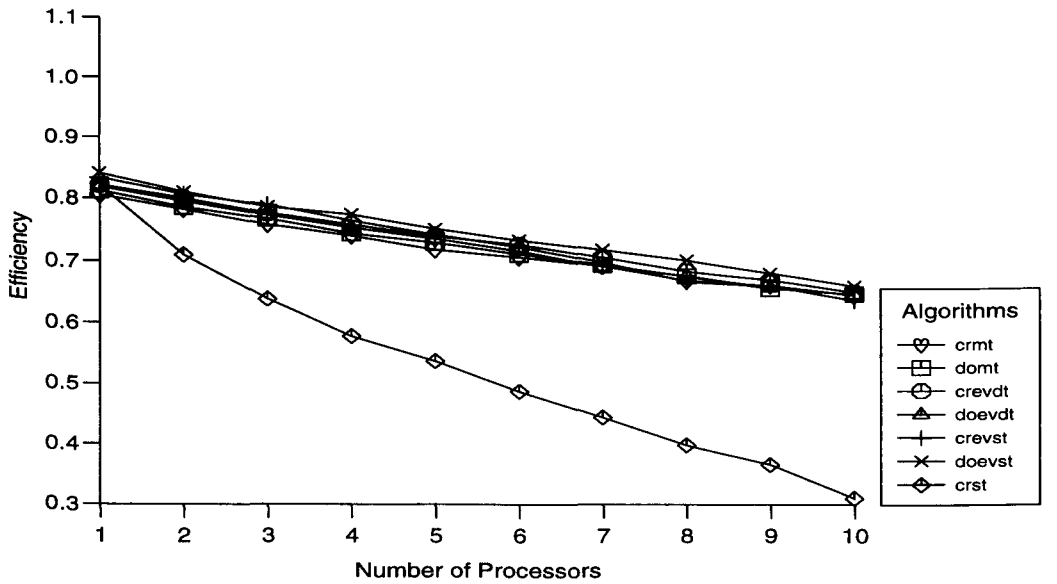


Fig. 4.9 Mean Efficiency Graph: LU Dec. Check and No Copying Vector

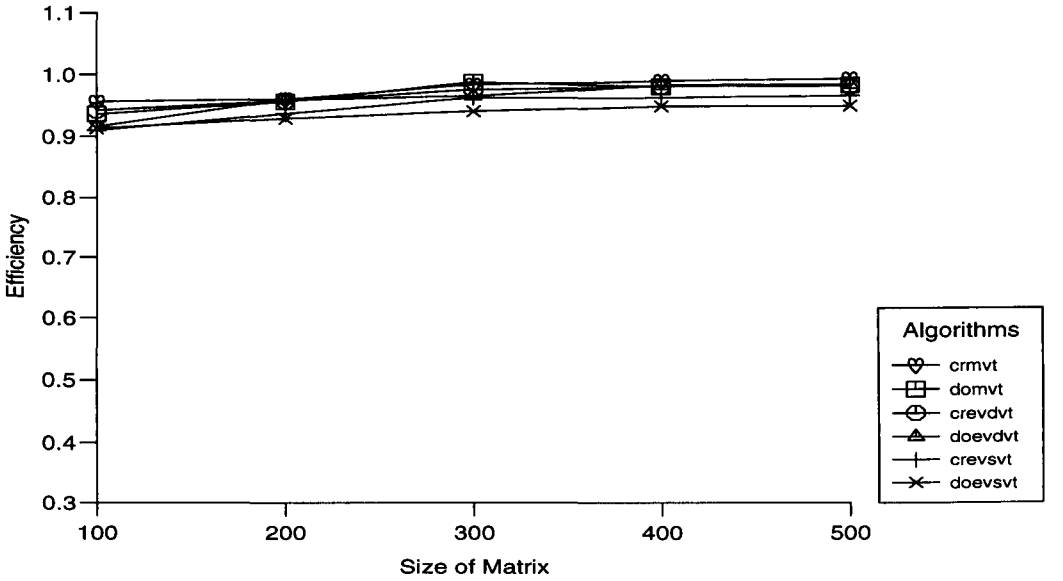


Fig. 4.10 No Check and Copying Vector for 2 Processors

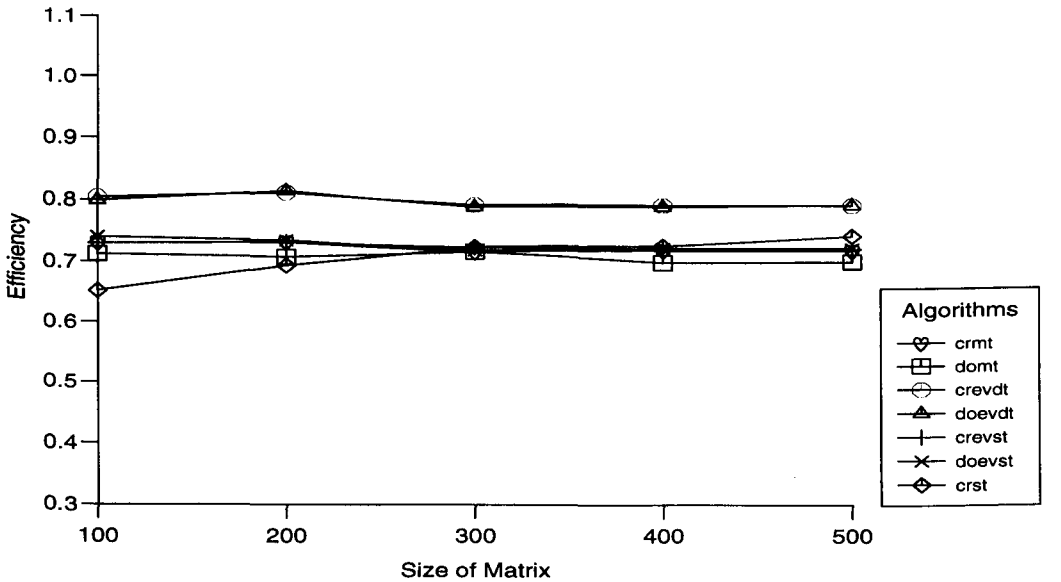


Fig. 4.11 No Check and No Copying Vector for 2 Processors

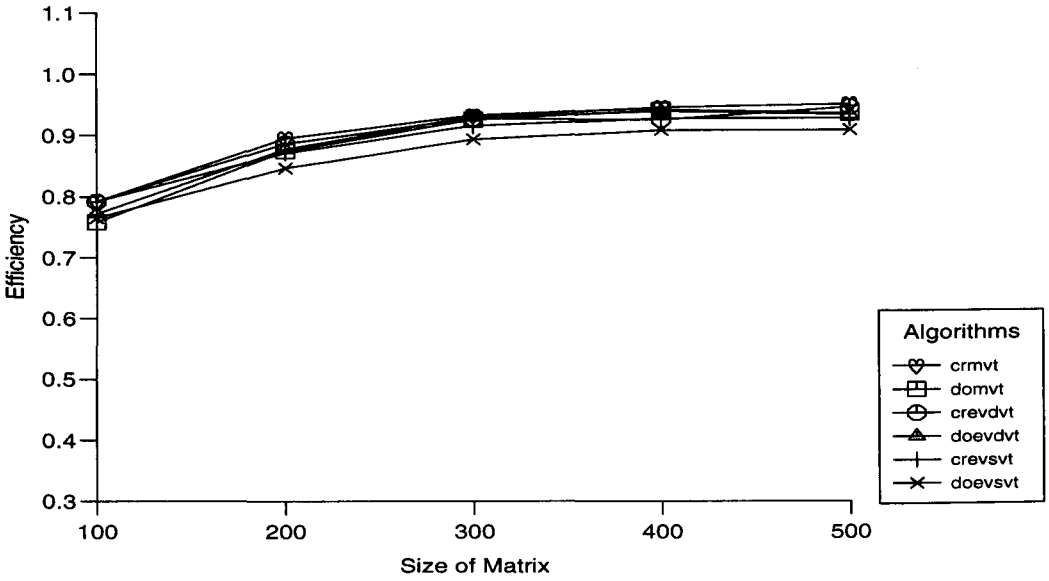


Fig. 4.12 No Check and Copying Vector for 6 Processors

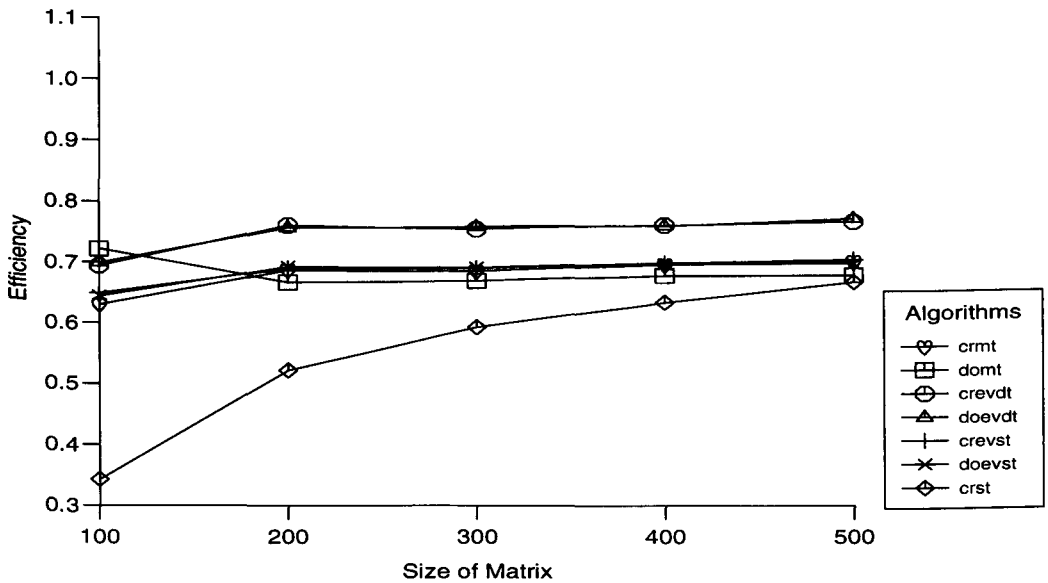


Fig. 4.13 No Check and No Copy Vector for 6 Processors

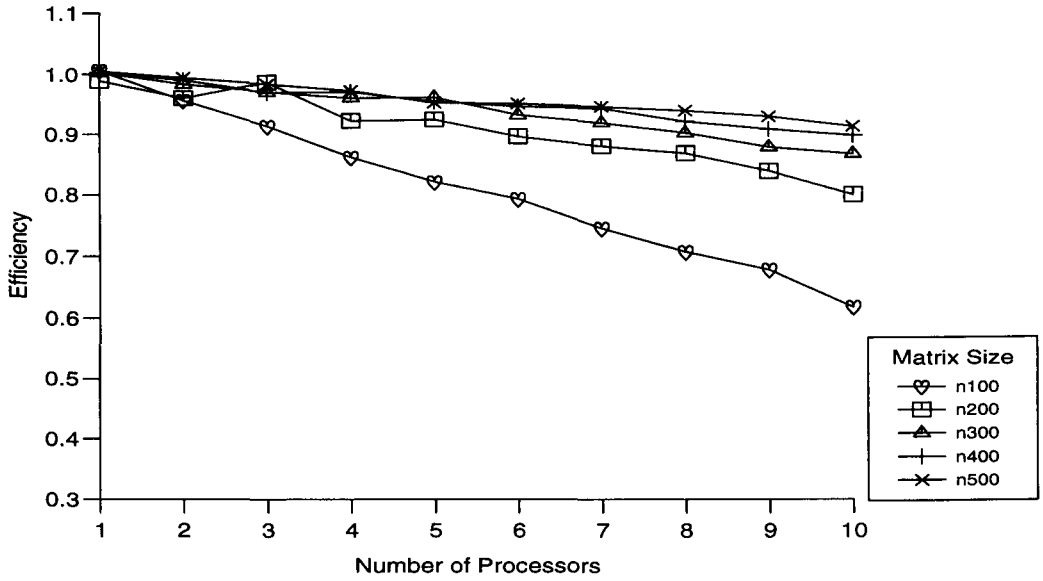


Fig. 4.14 No Check and Copying Vector "crmvt "

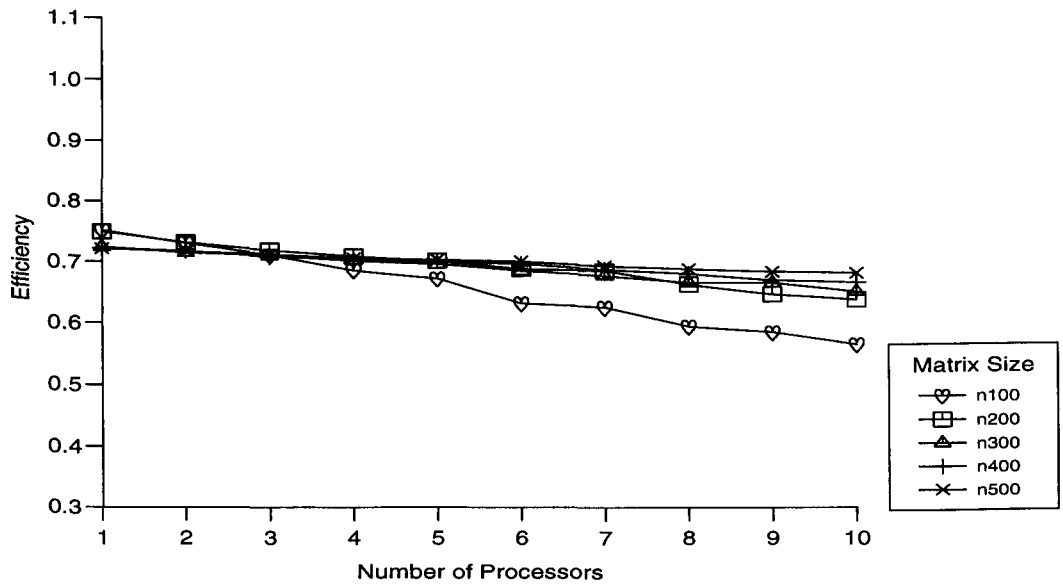


Fig. 4.15 No Check and No Copying Vector "crmt "

In the *checking* and *no copying* case, method two, the static allocation Doolittle-like (*doevst*) algorithm is best. In the 'no checking' and 'no copying' case the dynamic allocation *crevdt* and *doevdt* algorithms give almost identical results, significantly better than the other versions.

In most cases the Crout-like versions are more efficient than the corresponding Doolittle-like versions and this may be because the former only requires finding a pivot and interchanging two values at the pivotal stage while the latter also requires divisions of the whole of the remainder of the column by the pivot.

4.4.3 Conclusion for LU Decomposition

The good results for column copying indicate that the time needed for this copying is possibly more than offset by the reduction in array subscript calculations and better locality of data. The comparison of the three basic implementations is less clear as there are a number of factors involved. For the column splitting versions with column copying it appears that more time is saved on avoiding waits than is lost in re-allocation of the columns, but that this is not true when reference is made directly to the matrix. Perhaps it is worth noting that separate experiments indicated that waiting or column changing due to pivot columns not being available are relatively rare events, so the overhead involved is not as important as might be thought at first.

Perhaps most important is the comparison of the new methods with the simple implementation of the Crout algorithm. They gain firstly by

avoiding THREAD joins at the end of each group of columns and secondly by modifying the pivoting implementation and so avoiding different processors writing to the matrix elements. In the 'event' implementations any matrix element is only written to by one thread and no other thread reads this element until its final value has been set, which means that good use of the write-deferred cache is likely.

For all versions, the larger the matrix size the better the efficiency, particularly for larger numbers of processors. In addition, the efficiencies for all methods get closer as the matrix size increases, which is not surprising as the arithmetic becomes relatively more important compared to overhead as the matrix size increases. However, the methods found best here work well even for medium sized matrices.

Overall the study shows that the column based approach to LU decomposition using the modified pivoting implementation described here does have significant advantages on a shared memory multiprocessor. It is also clear that copying of columns which are to be updated to vectors is worthwhile, and that using this the Crout-like column splitting version is marginally the best.

CHAPTER 5

Reduction of a General Matrix to Hessenberg Form

5.1 Introduction

In this chapter we examine the reduction of a general matrix to upper Hessenberg form. Such algorithms do not of themselves solve the eigenvalue problem, but this approach does reduce the problem to a form that can be manipulated inexpensively Watkins [93]. We describe an evaluation of five parallel implementations using Householder transformations. We also compare three different synchronisation mechanisms (these mechanisms are introduced in chapter 3) when applied to this particular problem.

The chapter is organised as follows. A sequential algorithm for reduction to Hessenberg form is described in section 5.2 and the data dependencies in this algorithm are considered. In section 5.3 five parallel implementations are described. Experimental result are presented in section 5.4. Conclusions are given in section 5.5.

5.2 Sequential Algorithm

The following algorithm reduces a general $n \times n$ matrix A to Hessenberg form. H is a Householder matrix with form $H = I - 2ww^T$ where w is a unit vector. An example of this type of algorithm is described in [43]. In this algorithm a sequence of Householder matrices $H_1 H_2 ... H_{n-2}$ are chosen such that

$$A_{n-1} = H_{n-2} H_{n-3} ... H_1 A H_1 ... H_{n-3} H_{n-2}$$

is upper Hessenberg matrix. Each step in the reduction of A to an upper Hessenberg matrix can be given as

$$A^* = HAH$$

where H is a Householder matrix and A^* indicates the new matrix. The multiplication takes the form $(HA)H$, unlike the algorithm described by Dongarra et al [27] which combines the two multiplications. This combination has benefits in reducing data transfer but involves more arithmetic. As good speed up with the simple algorithm is obtained and the alternative algorithm will not be considered further.

The algorithm considered here chooses an H using the pivotal column, headed by $a_{k+1,k}$ for $k = 1, 2, \ldots, n-2$, so that the elements below the head of the new pivotal column are zero. The Householder transformation is used to update the later columns $k+1, k+2, \ldots, n$ while the columns $k = 1, 2, \ldots, k-1$ are not altered. When the column updates corresponding to the first multiplication are completed, the row updates corresponding to

the second multiplication are carried out for all the rows. Following this the next pivotal column can be treated and the whole process repeated for the next stage.

The algorithm below performs real Householder reduction to upper Hessenberg form:

for $k = 1 : n-2$
$s = \sqrt{A_{k+1k}^2 + A_{k+2k}^2 + ... + A_{nk}^2}$
$t = A(k+1, k)$
$r = 1/(s(s + |t|))^{1/2}$
if $(t < 0)$
$s = -s$
$u(k) = -s$
$A(k+1, k) = r * (t + s)$
$A(k+2 : n, k) = r * A(k+2 : n, k)$
(Set up the Householder transformation H)

$t(k+1 : n)^T = A(k+1 : n, k)^T * A(k+1 : n, k+1 : n)$
$A(k+1 : n, k+1 : n) = A(k+1 : n, k+1 : n) - t(k+1 : n)$
$*A(k+1 : n, k)^T$
(Multiply on the left by H)

$t(1 : n) = A(1 : n, k+1 : n) * A(k+1 : n, k)$
$A(1 : n, k+1 : n) = A(1 : n, k+1 : n) - t(1 : n) * A(k+1 : n, k)^T$
(Multiply on the right by H)
endfor

Before considering possible parallel implementations, it is useful to consider the data dependencies of this algorithm. We consider updates to pivotal columns, standard columns and rows rather than updates to individual elements. This is a simplification of the situation but is sufficient for the present discussion.

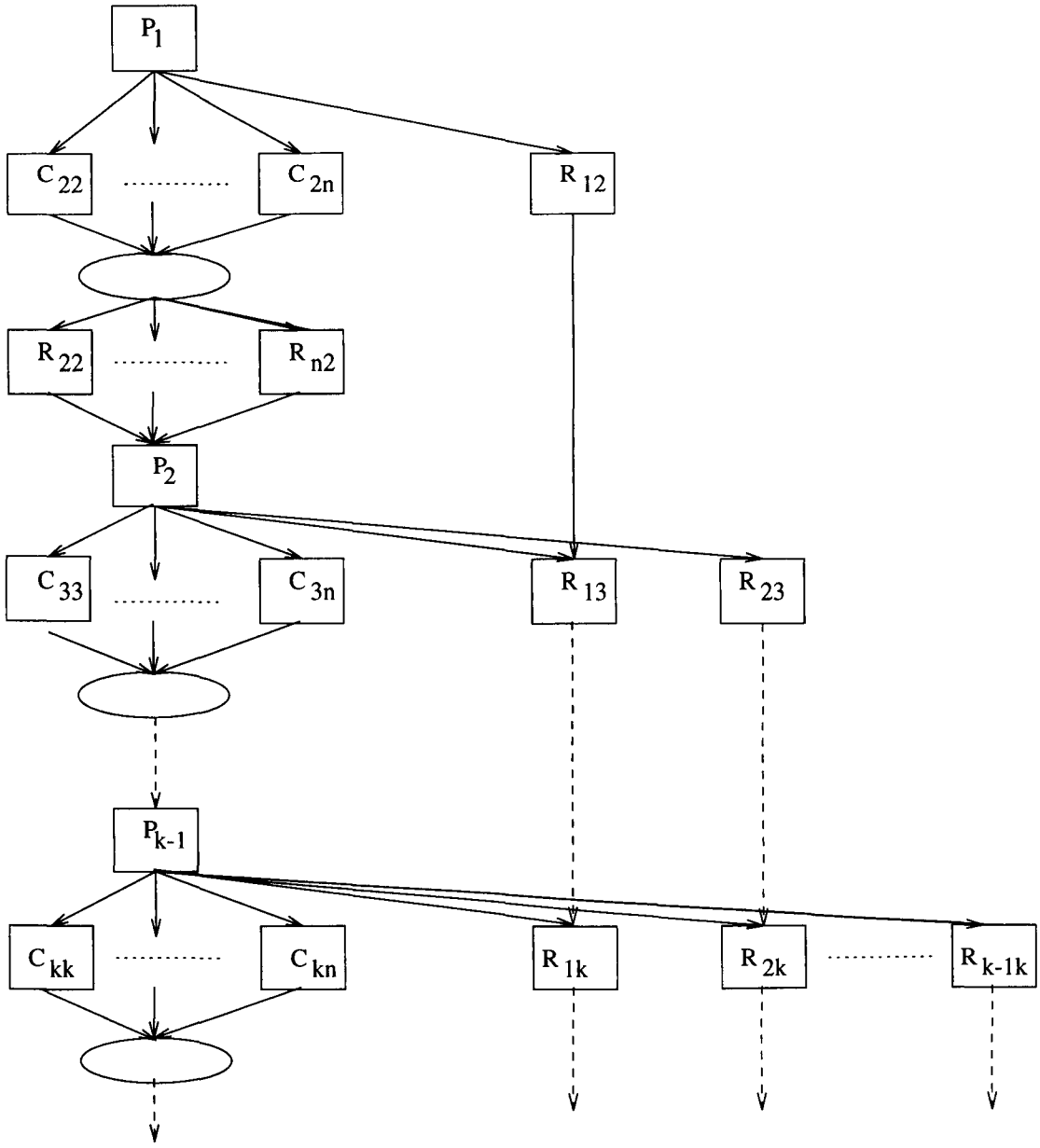


Fig. 5.1 Dataflow Diagram for Hessenberg Form

In figure 5.1 the k^{th} pivotal update is denoted by P_k, the update to the column headed by a_{jk} using P_{j-1} is denoted by C_{jk} and the update to the row starting with element a_{jk} using P_{k-1} is denoted by R_{jk}. We can see from the above diagram that the row updates at any particular stage can be divided into two groups. The first group which will be denoted by *rowa* consist of the rows with row numbers less than or equal to the latest pivotal

stage number. The other rows are denoted by *rowb*. The *rowb* rows must be completed before the next pivotal update can be made, while this is not true for the *rowa* rows. This observation will be made use of in algorithms 4 and 5 described below.

5.3 Parallel Implementations

We consider five essentially different parallel implementations. The first of these is a simple implementation carrying out the pivotal column updates sequentially, followed by parallel column updates and then parallel row updates. This is possible because all the column updates C_{kk} to C_{kn} are independent of each other, and can be carried out once the pivotal update P_{k-1} has been completed. Similarly, rows R_{1k} to R_{nk} are independent and can be updated when the column updates C_{kk} to C_{kn} have been completed. Both these parallel updates are terminated with "THREADjoins", which act as barriers. This simple implementation provides a basis for comparison with the other algorithms.

The next two algorithms are based on the observation that at any stage the updates to *rowa* rows can be carried out at the same time as the column updates while the *rowb* row updates can not. The pivotal updates are again carried out sequentially. The *thread* chooses columns for updating until there are no more columns to be allocated. If all columns are allocated, then a row is chosen from rows 1 to n taken in order. Updates are carried out immediately for *rowa* rows, but for *rowb* rows the thread waits until all the columns have been completed, as these row updates may only be carried out if this has happened. The waiting is carried out by using a loop checking a count of the number of columns (*done*) which have been completed. These two algorithms avoid one of the "THREADjoins" needed at each stage of the first algorithm.

In the second algorithm the columns and rows are allocated to threads dynamically. This should provide flexibility for the algorithm in a multi-user environment, for if one thread is held up others will carry out the work instead. The variables used to control the allocation of work to threads are only modified in critical regions. For this second algorithm the critical region was initially controlled by *locks*, but in addition, we also use the *semaphore* and *monitor* synchronisation mechanisms. This was done so that the different mechanisms could be compared.

In the third algorithm the columns and rows are allocated in a predetermined way (scattered ordering). This allows the pre-allocation of tasks to threads, which should reduce the amount of interprocessor communication as no lock is needed for the row and column allocation. However, this gives less flexibility than algorithm 2. The parallel version of algorithm 2 is outlined below:

```
for k = 1 : n − 2
    Process pivotal column k sequentially
    nxcol = k + 1          // next column
    nxrow = 1              // next row
    done = k               // number of columns completed

    do in parallel
    do
        col = nxcol
        nxcol = col + 1
        Allocate column (col) to this thread and increment (col)
        If (col <= n)
            modify column (col)
            then increment count (done)
            If (done == n)
            allow rows k to n to be processed
            as all columns are completed
    while (col < n)
    end parallel
```

```
    do in parallel
    do
        row = nxrow
        nxrow = row + 1
        Allocate row (row) to this thread
        If ((row > k) and (done < n)
            wait until count done = n
        modify row (row)
    while (row < n)
    end parallel
end
```

The fourth algorithm avoids the sequential updating of the pivotal column by allowing a start on the next pivotal column while *rowa* rows are still being processed. The calculations are still carried out in stages corresponding to the pivotal updates. A stage starts after a pivotal update is made and the previous *rowa* rows have been completed. The next set of columns are allocated for update in parallel, with *rowa* rows allocated when no more columns remain to be allocated. However, as soon as all the set of column updates is completed, *rowb* rows are allocated for update even though all *rowa* rows may not have been allocated. When the updates of the *rowb* rows are completed, the next pivotal column is started, regardless of whether all *rowa* rows are completed or not. The stage terminates by completing any *rowa* row updates which have not been treated previously.

This implementation is expected to reduce the waiting time as *rowa* row processing can take place at the same time as the pivotal column update. Five counters and three Boolean flags are needed, one for completion of columns (*done*), one for completion of *rowb* rows (*doner*), and the others are used in the dynamic allocation of columns (*nxcol*), rows *rowa* (*nxrowa*) and rows *rowb* (*nxrowb*). The flag *rd* indicates whether the *rowb* rows can be updated, the second flag *dopiv* indicates whether the pivotal column can be updated, the third one (*test*) indicates whether the pivotal column as

well as *rowa* rows are completed. A "THREADjoin" is used before the next set of columns is updated.

Note that this algorithm does not allow *rowa* rows to be allocated after all *rowb* rows are allocated until all *rowb* rows are completed. This could cause some extra waiting. This parallel version is outlined as algorithm 4 below:

```
Process first pivotal column sequentially and
set next pivot number (piv = 2)
for k = 1 : n − 2
    nxcol = k + 1              // next column
    nxrowa = 1                 // next rowa row
    nxrowb = k + 1             // next rowb row
    done = k                   // Number of columns completed
    doner = k                  // Number of the rowb rows completed
    do in parallel
    char rd, dopiv, test // to arrange the rowa rows update
    rd and dopiv = FALSE
    do
        col = nxcol
        nxcol = col + 1
        Allocate column (col) to this thread and increment (col)
        If (col <= n)
            modify column (col)
            then increment count (done)
            If (done == n)
            allow (rowb) rows to be updated
            as all column are completed
    while (col < n)
    do
        If (done == n)
            row = nxrowb
            nxrowb = row + 1
            Allocate rowb (row) to this thread
            rd = TRUE
        else
            row = nxrowa
```

```
            nxrowa = row + 1
            Allocate rowa (row) to this thread
        If ((rd) and (row <= n))
            update row
            increment count (doner)
        If ((!rd) and (row <= k))
            update row
    while (doner! = n)
    do
        ra = 1                  // ra is rowa row
        If (piv == k + 1)
            then increment pivot count (piv = k + 2)
            dopiv = TRUE
        If ((dopiv) and (k < n - 2))
            process pivotal column (k + 1)
            dopiv = FALSE
        else
            ra = nxrowa
            row = ra
            nxrowa = ra + 1
            Allocate rowa (row) to this thread
            If (row <= k)
                update row
        test = (piv = k + 2) and (ra > k)
    while (!test)
    end parallel
end
```

The fifth algorithm was designed to avoid the waiting problem in algorithm four and also to avoid using any "THREADjoins" except on completion of the whole reduction. The other important aim in the design of this algorithm was to avoid splitting the algorithm into stages by making use of the observation made clear in the dataflow diagram that the pivotal column, column and *rowb* row updates are not dependent on the *rowa* row updates being completed. In fact when this algorithm is run on a single processor all the *rowa* row updates are left until all the other updates have been completed.

Work is allocated to threads in the order indicated in the left hand column of the dataflow diagram, that is, a pivotal update is followed by a set of column updates which is in turn followed by a set of *rowb* updates, leading to the next pivotal update. However, when any thread finds none of this work available for allocation, any *rowa* rows which are ready for updating are allocated instead. When *rowa* rows are processed all updates currently possible are carried out, that is all updates which depend on pivotal columns which have been already treated excluding any updates that have been previously carried out. This should allow transfer between cache and shared memory to be reduced. To achieve this two integer vectors are kept. The vector element $ct(ra)$ is used to indicate the position of the last update to row ra and the element of vector $nactv(ra)$ is used to indicate whether some thread is currently working on row ra. The corresponding element of the vector *nactv* is set to $FALSE$ (i.e. thread is working on row *rowa*) whenever a *rowa* rows being updated. When the corresponding *rowa* rows updates are completed, then vector element $ct(ra)$ is incremented and the element of vector $nactv(ra)$ is set to $TRUE$.

The columns and rows are allocated dynamically to the threads, and counts for the columns and the *rowb* rows are used as before. As we indicated above, the *rowa* rows are updated independently. This should ensure that the pivotal columns are given high priority in the early part of the calculation when they are more critical, but gradually have less priority as the calculation proceeds, as in the later stages when there will be plenty of *rowa* rows ready for updating. Two implementations of this algorithm which only differ in the way the reduction is terminated were developed and tested. These implementations are expected to be more efficient than the previous versions.

In the version *Hella* a thread terminates when all the *rowa* rows are

completed. In the *Helle* version the thread terminates when all the *rowa* rows are allocated and the current thread has no further work. In the *Hella* version a delay loop was included which is activated when the thread finds no work to carry out. This was with the aim of reducing contention for the lock which is repeatedly accessed in the allocation phase. In practice this delay made little difference to the results. Algorithm 5 (*Hella* version) is outlined below:

```
Process first pivotal column sequentially and
set next pivot number (pv = 2)
rct = 0
for k = 1 : n − 2
    ct(k) = k
    nactv(k) = TRUE
end
    nxcol = 2
    nxrowa = 1
    nxrowb = n + 1
    done = 1                    // Number of columns completed
    doner = 1                 // Number of the rowb rows completed
    dopiv = FALSE
    do in parallel
    do
        piv = pv
        If ((dopiv) and (piv <= n − 2))
            dopiv = FALSE
            process pivotal column (piv)
            then increment pivot (pv = piv + 1)
            nxcol = piv + 1
            done = piv
        else
        col = nxcol
        If (col <= n)
            nxcol = col + 1
            modify column (col)
            increment count (done)
            If (done == n)
```

```
                nxrowb = piv
                doner = piv − 1
        else
            row = nxrowb
            If (row <= n)
                nxrowb = row + 1
                modify row (rowb)
                increment count (doner)
                If (doner == n)
                dopiv = TRUE
            else
                ra = nxrowa
                If (ra == piv − 1)
                nxrowa = 1
                else
                nxrowa = ra + 1
                If ((ra < piv) and (ct(ra) <= n − 2) and nactv(ra))
                    nactv(ra) = FALSE
                    while(ct(ra) < piv)
                      modify row (rowa)
                      increment number (ct(ra))
                    nactv(ra) = TRUE
                    If (ct(ra) == n − 1)
                        then increment row count (rc)
                else
                  wait
    while (rc <  n − 2)
    end parallel
end
```

The *Helle* implementation is a little more complicated due to the need to ensure that all *rowa* row updates are completed while allowing a thread to terminate before this has happened. This is done by checking at the end of each row of *rowa* updates whether the final pivotal column has been completed and if so continuing with this row until all updates have been completed. When the final pivotal column is completed the indicator for the next *rowa* (*nxrowa*) is set to one and the threads then process the rows

in order. When row *ra* is allocated *nactv*(*ra*) is set to FALSE and is then not reset to TRUE on completion of the updates so that threads do not attempt to process a row which has already been completed.

5.4 Experimental Results

The results use the notation *He* for our implementations of parallel upper Hessenberg reduction. The numerical results were obtained for the five versions outlined in section 5.3. The first method (i.e. the simple implementation) is denoted by *ss*, the second implementation using dynamic allocation by *ld* (using locks), *smd* (using semaphores), and *md* (using monitors). The third implementation using static allocation is denoted by *ls*, the fourth implementation using pivotal processing done in parallel by *mml* and the fifth implementations using delayed *rowa* allocation by *lla* and *lle*. The representation of the matrix by columns is indicated by the final character *t*, otherwise the representation is by rows.

The comparisons were carried out by measuring the elapsed time for a general matrix to be reduced to upper Hessenberg form using each of the implementations. Comparisons were carried out between the performance of the five implementations. Comparisons were also carried out between the performance of the three different synchronisation mechanisms in the second implementation.

We tested the algorithms using from one up to ten processors (1,2,4,6,8,10) with matrices of sizes 100(100)500. The sequential times were obtained from the simple algorithm (*Heseq*) which was compiled without array bound checking using the row representation version. This was slightly better than the column representation version and as expected, significantly better than the same version with array bound checking. The parallel and sequential times were, of course, obtained for the same sized matrix.

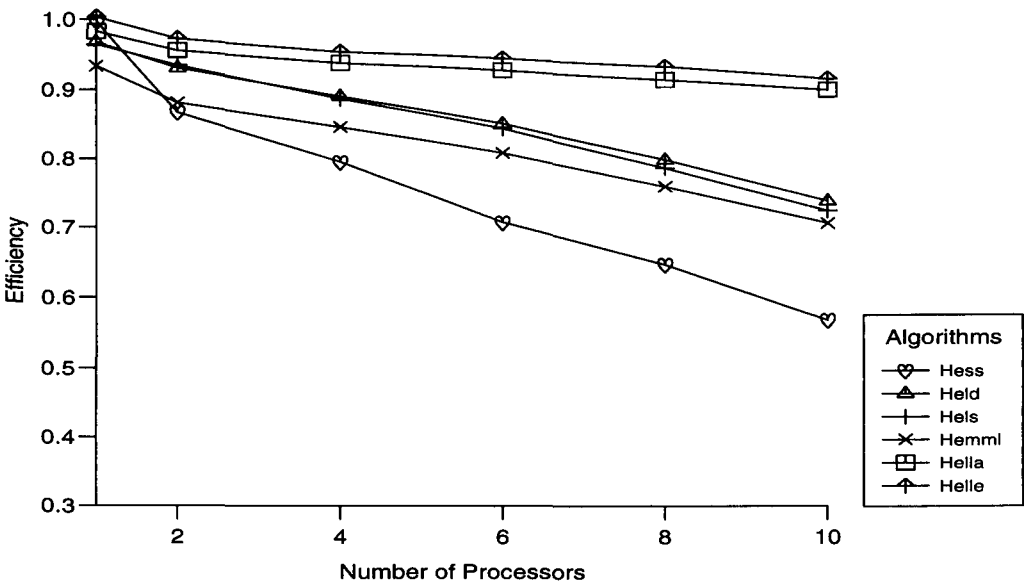


Fig. 5.2 Mean Efficiency Graph: Hessenberg Form with No Check

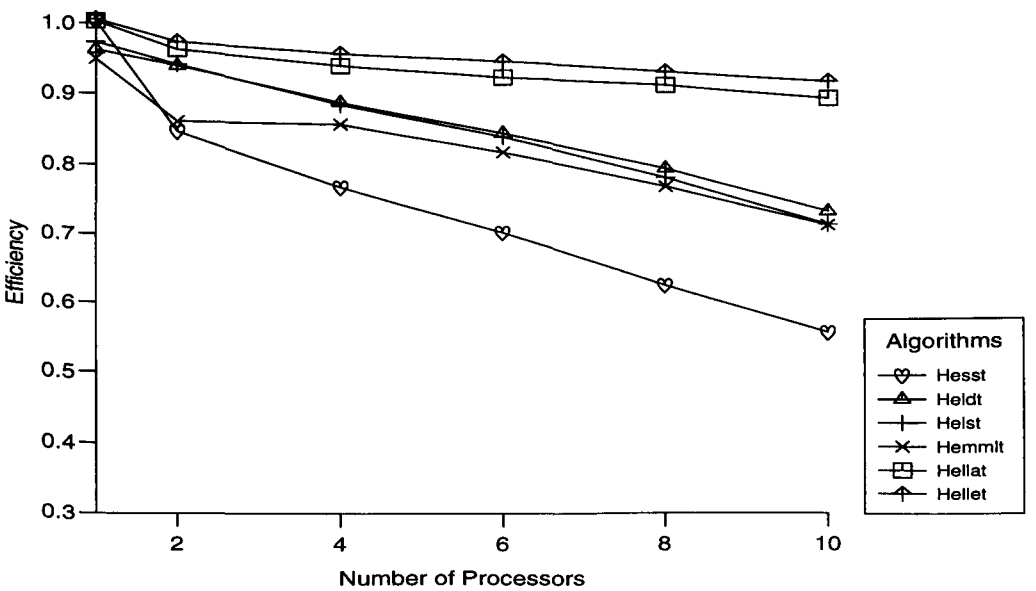


Fig. 5.3 Mean Efficiency Graph: Hessenberg Form with No Check

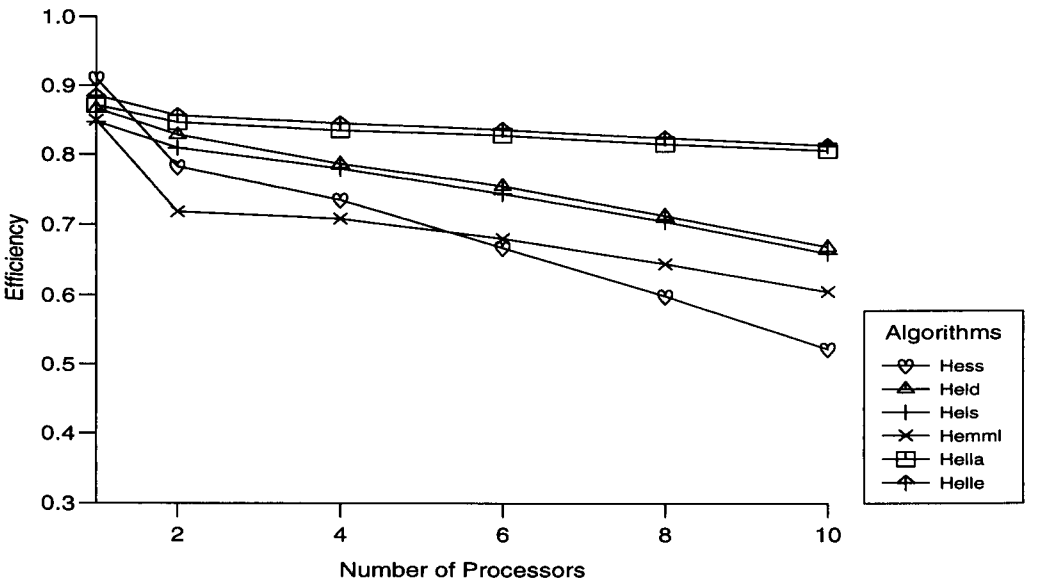


Fig. 5.4 Mean Efficiency Graph: Hessenberg Form with Check

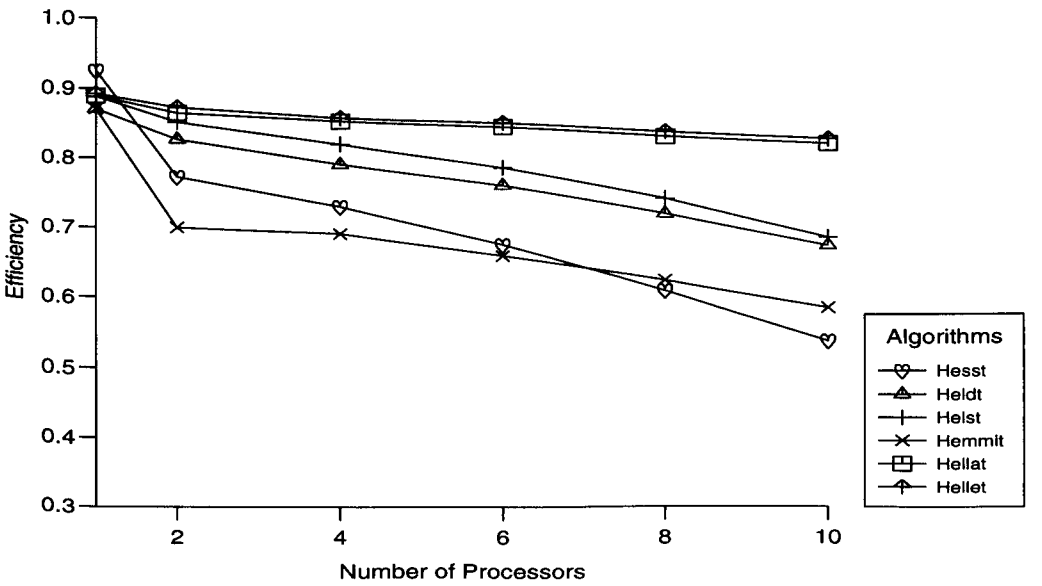


Fig. 5.5 Mean Efficiency Graph: Hessenberg Form with Check

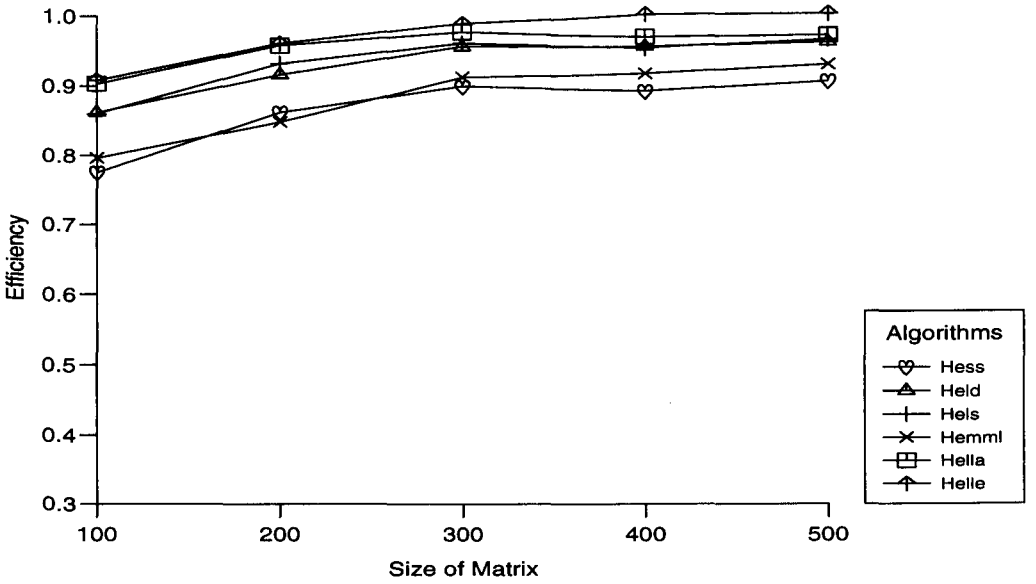


Fig. 5.6 No Check for 2 Processors

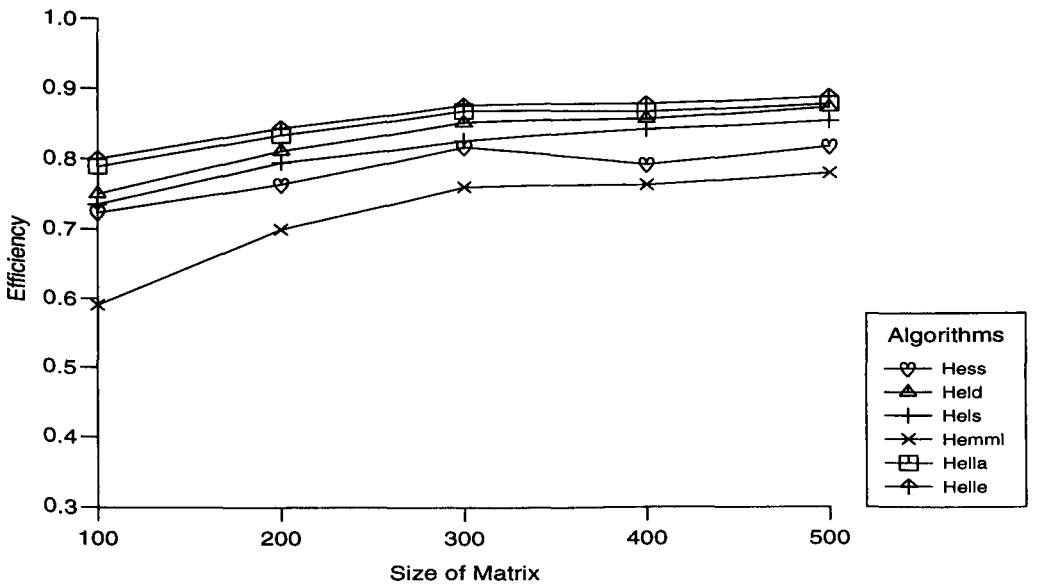


Fig. 5.7 Check for 2 Processors

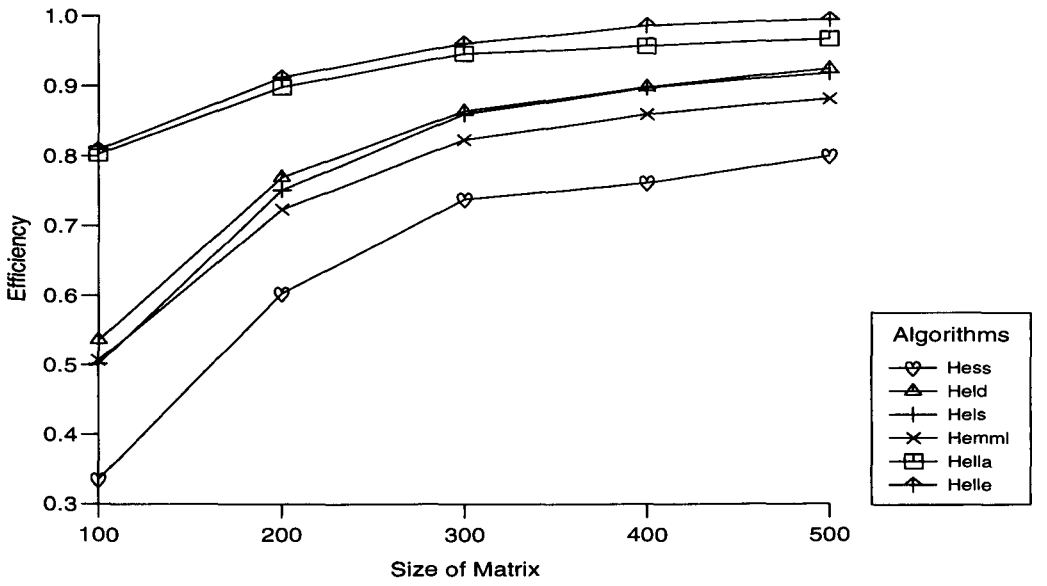


Fig. 5.8 No Check for 8 Processors

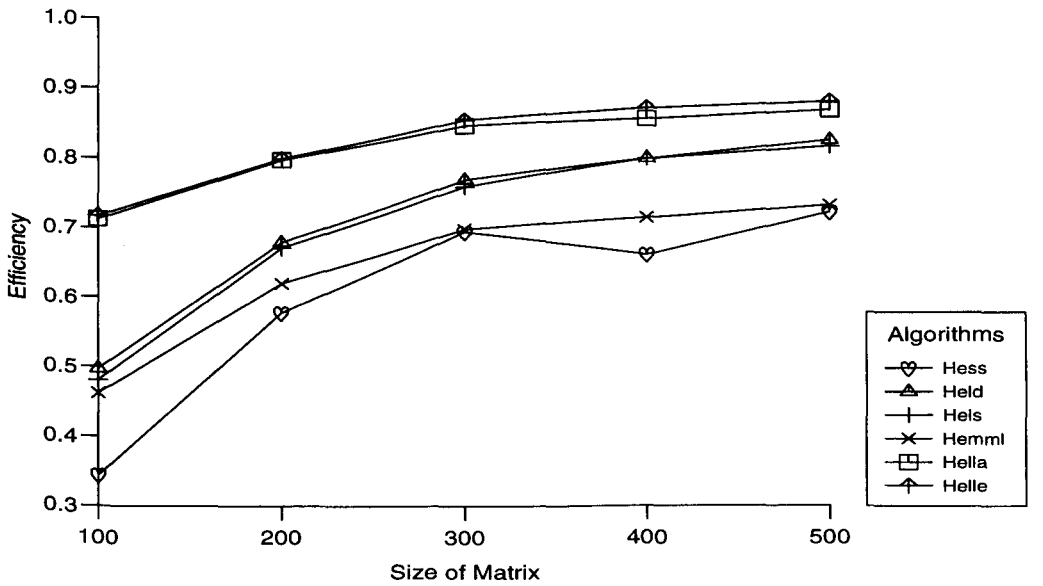


Fig. 5.9 Check for 8 Processors

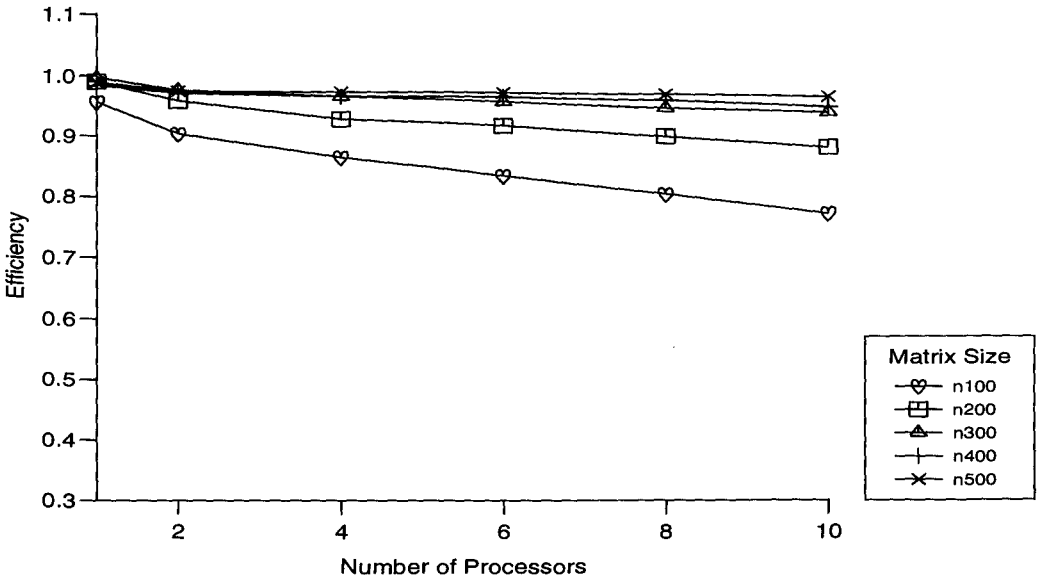


Fig. 5.10 No Check "Hella"

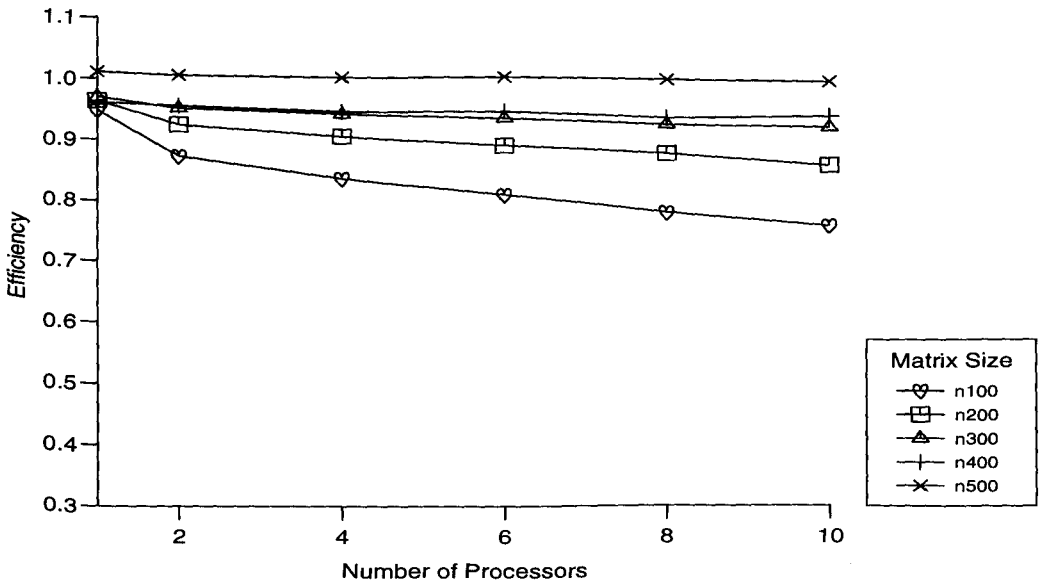


Fig. 5.11 No Check "Helle"

To show the performance of the five different parallel versions, we plot in figures 5.2, 5.3, 5.4, and 5.5 mean efficiencies against number of processors. Figures 5.6 and 5.7 show actual efficiencies using 2 processors for results with and without array bound checking and figures 5.8 and 5.9 show similar plots for 8 processors. Figures 5.10 and 5.11 display results for different sized matrices and different numbers of processors for the fifth method (version *Helle*).

Also to show the performance of the three different synchronisation mechanisms in the second implementations we plot in figures 5.12, 5.13, 5.14, and 5.15 mean efficiencies against number of processors, for results with and without array bound checking. Figures 5.16 and 5.15 show actual efficiencies using 2 processors for results with and without array bound checking and figures 5.18 and 5.19 show similar plots for 8 processors. The second method using *lock* (version *Held*) displays a good parallel efficiency for all our versions.

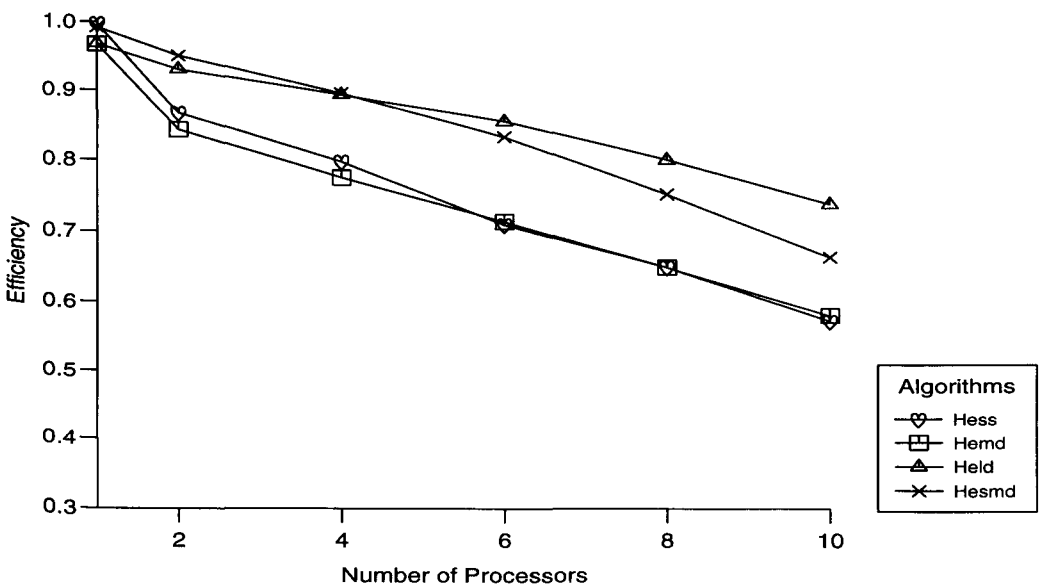


Fig. 5.12 Mean Efficiency Graph with No Check

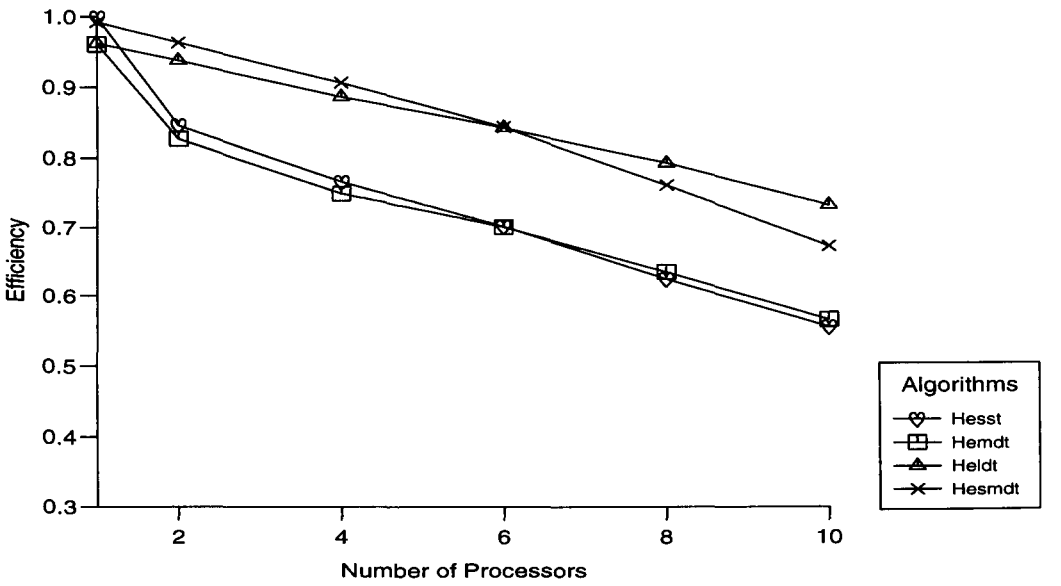


Fig. 5.13 Mean Efficiency Graph with No Check

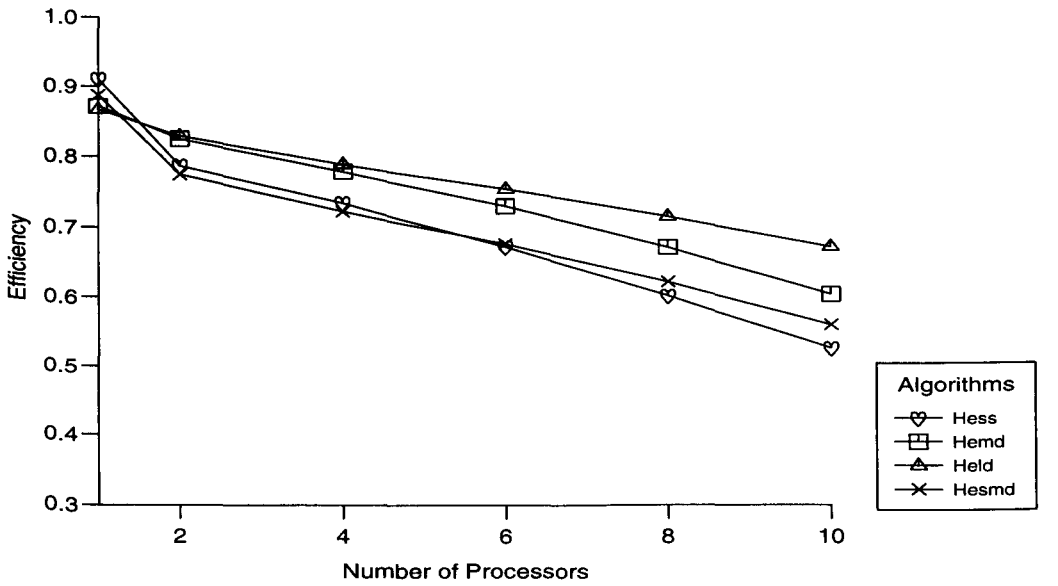


Fig. 5.14 Mean Efficiency Graph with Check

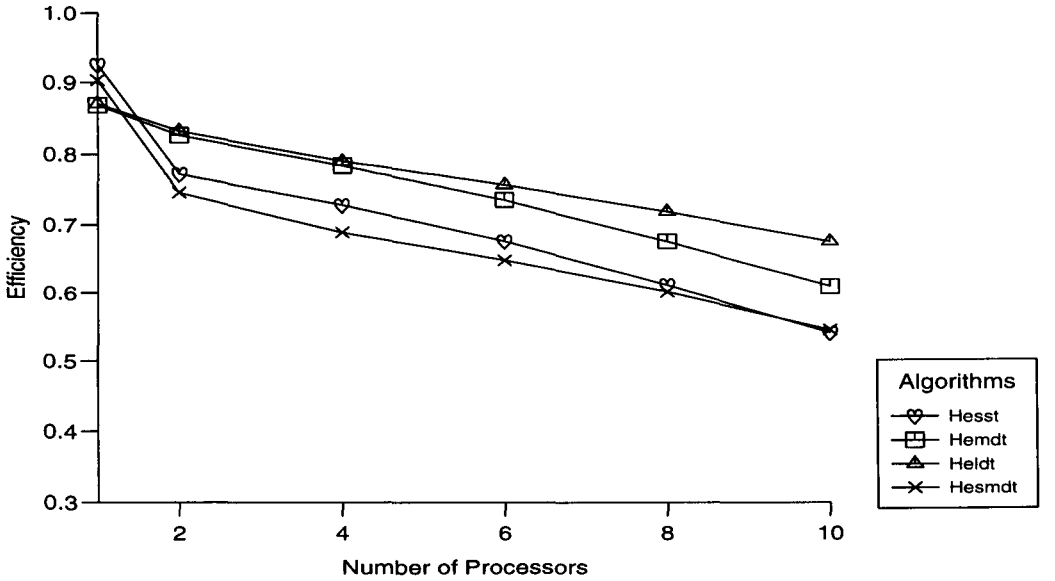


Fig. 5.15 Mean Efficiency Graph with Check

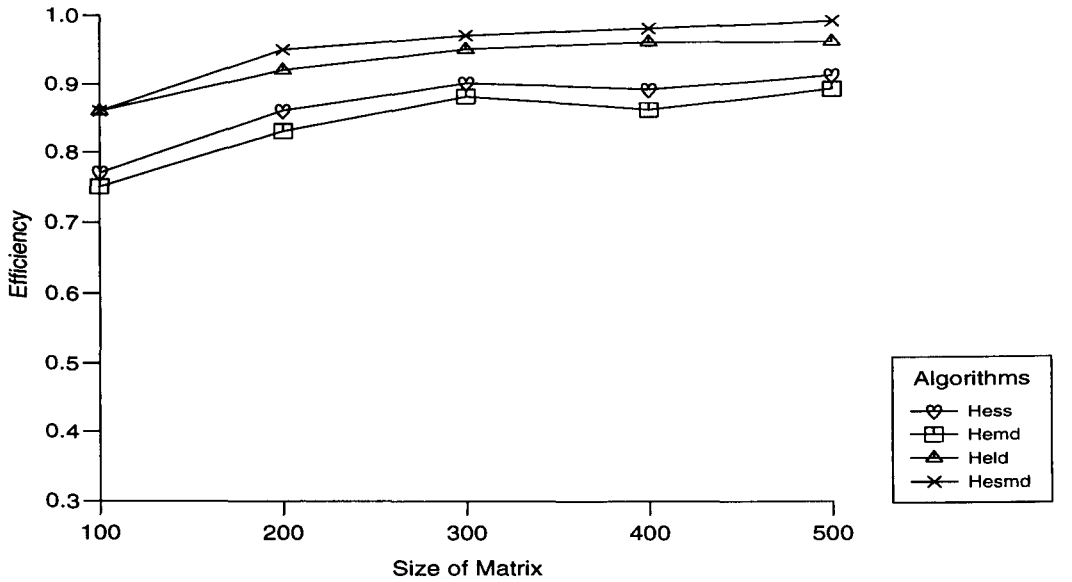


Fig. 5.16 No Check for 2 Processors

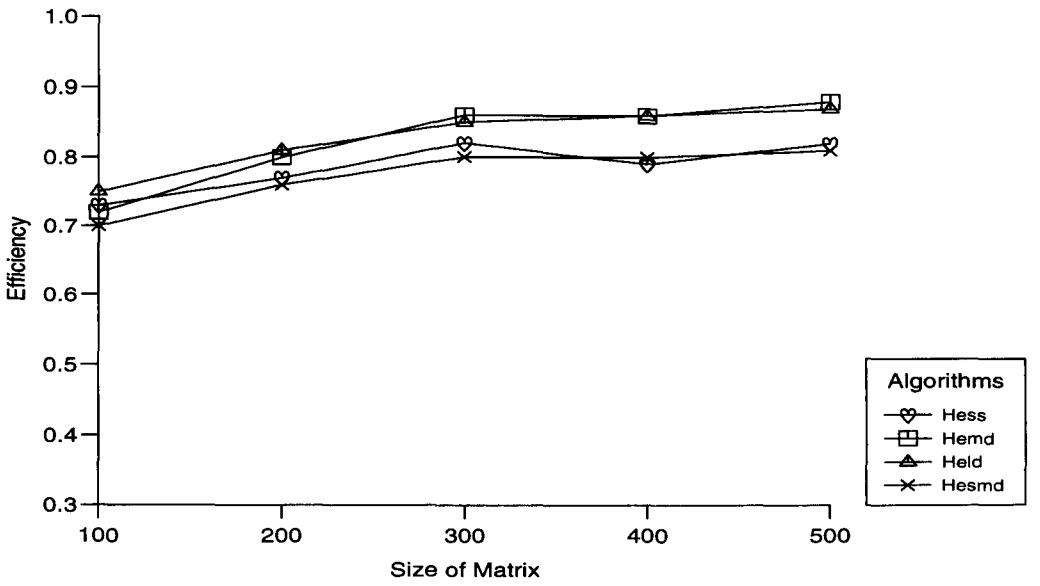


Fig. 5.17 Check for 2 Processors

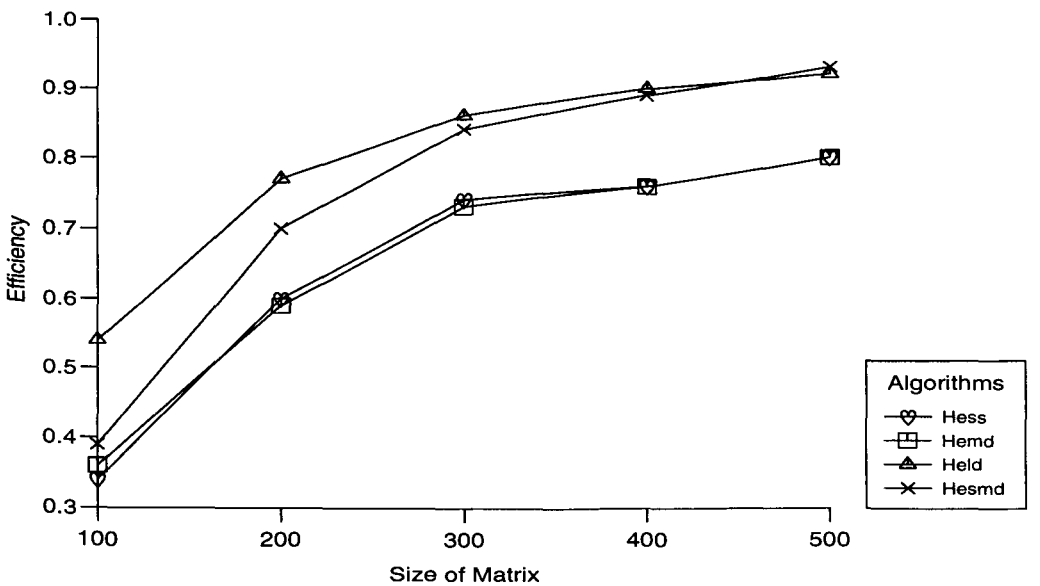


Fig. 5.18 No Check for 8 Processors

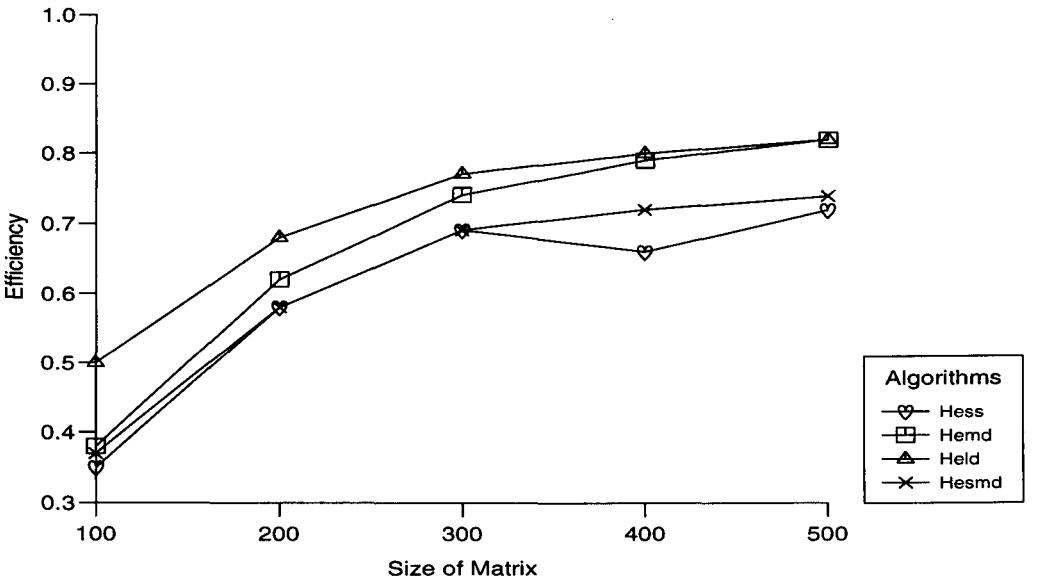


Fig. 5.19 Check for 8 Processors

5.5 Conclusions

In this chapter we have investigated some parallel algorithms for the reduction of a general matrix to upper Hessenberg form.

As expected, the first method (the simple parallel implementation *Hess*) is slower than all the other versions. This is because the *THREAD* joins at the end of each stage of column and row updates lead to significant waiting.

There is a clear conclusion that the fifth implementations (*Hella* and *Helle*) are better than the other implementations, with little difference between row and column versions. The efficiency graphs show that the difference between the (*Hella* and *Helle*) implementations are very small and very close to possible errors in measurement, though *Helle* seems to be marginally better. It is notable that for larger numbers of processors with these implementations even small sizes of matrix give significantly better results than other versions. As the number of processors increases the graphs

for this algorithm are very nearly horizontal indicating little loss. The high efficiency indicates that losses due to critical areas controlled by locks are very small.

Surprisingly, the fourth algorithm (*Hemml*) including the pivotal column done in parallel has poorer efficiency than the other implementations except the simple parallel implementation (*Hess*), possibly due to more overhead than *Held* and static *Hels*, and more waiting than version *Hella* and *Helle*, as *rowa* rows are not processed after all columns are processed until all *rowb* rows are completed. In spite of this the *Hemml* version is still significantly better than *Hess* in most cases. The differences between the implementations *Held* and *Hels* which only vary in their method of allocation to the threads are relatively small. Comparing the results for these versions shows that the *Held* version comes out better in most cases, except with array bound checking and using column representation of the matrix, where version three *Helst* is better than *Heldt*.

For all versions, the larger the matrix size the better the efficiency, particularly for larger numbers of processors. In addition, the efficiencies for the *Held*, *Hels* and *Hemml* methods get closer as the matrix size increases. This is not surprising as the arithmetic becomes relatively more important compared to overhead as the matrix size increases.

When we compare the "locks" and "semaphores" synchronisation mechanisms for the second algorithm, we find that with no array bound checking and both row and column representation the versions *Held* and *Hesmd* are very close particularly for 4 and 6 processors.

With array bound checking the *lock* version is significantly better than the *semaphore* versions. This is shown in figures 5.12 and 5.13.

The *monitor* version is worse than the *semaphore* version with no check

but it is better than the *semaphore* version with array bound checking case. The *monitor* version in the checking case is still significantly poorer than the *lock* version, and the difference increases with the number of processors.

Also with array bound checking the *semaphore* versions are some times even poorer than the simple version *Hess*. With no array bound checking the *monitor* versions *Hemd* and *Hemdt* give similar efficiencies to the simple implementation *Hess*.

There is a clear conclusion about using the synchronisation mechanisms. All implementations using "locks" are more efficient than those using "monitors" or "semaphores". The "locks" version gives the best efficiency in most cases. In conclusion, we have observed that the parallel implementation of delayed *rowa* row allocation (*Hella* and *Helle* versions) attain remarkably high efficiencies, with both row and column representations of the matrix.

CHAPTER 6

Reduction of a Symmetric Matrix to Tridiagonal Form

6.1 Introduction

In this chapter we examine the reduction of a real symmetric matrix to tridiagonal form. Such algorithms do not of themselves solve the eigenvalue problem, but this approach does reduce the problem to a form that can be manipulated inexpensively [93]. Reduction to tridiagonal form is a major step in eigenvalue computations for symmetric matrices. Finding the eigenvalues and eigenvectors of a tridiagonal matrix is significantly simpler than computing those for a general symmetric matrix [95]. In recent years a number of parallel implementations of algorithms for eigenvalue problems have been widely investigated. Algorithms for shared memory architectures ([26] and [27]) and distributed memory architectures ([28],[55] and [13]) have been implemented for reducing a symmetric matrix to tridiagonal form. Dongarra and Sorensen [26] present a method intended to be used in conjunction with the initial reduction to tridiagonal form to compute the complete eigensystem of the original matrix. Dongarra et al [27] consider

block algorithms for the reduction of a real symmetric matrix to tridiagonal form using Householder transformations.

Let B be an $n \times n$ symmetric matrix T. Our aim is to reduce a symmetric matrix to tridiagonal form, that is a matrix described as follows:

$$T_{ij} = 0 \quad unless \quad j = i, \;\; i+1, \; or \;\; i-1.$$

The sequential Householder tridiagonalisation approach is described in [40],[93] and [95].

The algorithms for reduction of a general matrix to Hessenberg form can also be used for the reduction of a symmetric matrix to tridiagonal form, and there are some rather obvious savings which can be made by taking the symmetry into account. However, if only half of the matrix is stored or used, such algorithms have much greater data dependencies which limit the possibilities for parallelisation.

The chapter begins with the description of a sequential algorithm for reduction to tridiagonal form in section 6.2. The data dependencies in this algorithm are also considered. In section 6.3 three parallel implementations are described. Experimental results are presented in section 6.4 and conclusions are given in section 6.5.

6.2 Sequential Algorithms

We consider the tridiagonalisation of an $n \times n$ real symmetric matrix B, using Householder transformations. The method uses $(n-2)$ successive

transformations to reduce a matrix to tridiagonal form. An example of such an algorithm is described in [93]. We present below the Householder transformation process and the tridiagonalisation of the symmetric matrix B.

Let $b = (b_1, b_2, \ldots, b_n)^T$ be a vector such that not all of the entries $b_2, b_3, \ldots, b_n$ are zero, and suppose we want to transform b to a vector $q = (q_1, q_2, \ldots q_n)^T$ where $q = Qb$ with Q orthogonal and such that $q_2 = q_3 = \ldots = q_n = 0$. Define $q = (\sigma,\ 0\ , \ldots\ 0)^T$ then $|\sigma| = \|q\|_2 = \|b\|_2$ because Q is chosen an orthogonal matrix.

Before we summarise the algorithm for the reduction of a symmetric matrix to tridiagonal form, let us point out one other potential danger. In order to calculate σ, we must calculate $\|b\|_2 = (b_2^2 + b_3^2 + \ldots + b_{n-2}^2)^{1/2}$ for such a computation. Since squaring doubles the exponent of a number, an overflow can occur if some of the entries of b are very large and an underflow can occur if some of the entries are very small. An overflow will usually stop the computation. An underflow may or may not stop the computation, depending on which compiler options are in effect. If the computation is not stopped, the underflow will be set to zero which can be dangerous if this makes σ equal to zero.

The problem with overflows and underflows in the calculation of $\|b\|_2$ can be alleviated by the following simple method: Let $m = max_{1 \leq k \leq n} |b_k|$. If $m = 0$, then $\|b\|_2 = 0$. Otherwise let $\hat{b} = (1/m)b$. Then $\|b\|_2 = m\|\hat{b}\|_2$. This scaling statement eliminates the possibility of overflow because $|\hat{b}_k| \leq 1$ for

all k. Underflows can not cause problems as clearly σ will be greater than or equal to 1. Thus these underflows are harmless and can safely be set to zero [93]. In this thesis the scaling statement is only used in the algorithm for the reduction of a symmetric matrix to tridiagonal form, though it could be applied in the other algorithms using Householder matrices.

To describe the algorithm for the reduction of a symmetric matrix to tridiagonal form, we adopt the following notation introduced in [93]. We begin with the matrix

$$B = \begin{pmatrix} b_{11} & a^T \\ a & \hat{B} \end{pmatrix}.$$

In the first step of the reduction, we transform B to $B_1 = Q_1 B Q_1$, where

$$Q_1 = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & & & \\ \vdots & & \hat{Q}_1 & \\ 0 & & & \end{pmatrix}$$

and $\hat{Q}_1$ is a Householder transformation matrix chosen so that $\hat{Q}_1 a = [\sigma_1\ 0\ \dots\ 0]^T$. Thus

$$B_1 = \begin{pmatrix} b_{11} & a^T\hat{Q}_1 \\ \hat{Q}_1 a & \hat{Q}_1 \hat{B} \hat{Q}_1 \end{pmatrix} = \begin{pmatrix} b_{11} & \sigma_1 & 0 & \dots & 0 \\ \sigma_1 & & & & \\ 0 & & & & \\ \vdots & & \hat{B}_1 & & \\ 0 & & & & \end{pmatrix}$$

We save a little bit here by not performing the computation $a^T\hat{Q}_1$, which duplicates the computation $\hat{Q}_1 a$. The main saving is made in the calculation

of $\hat{B}_1$ as described below. Finally, because the matrix B is symmetric, we need only store the lower triangles of the matrices.

$\hat{Q}_1$ is a Householder transformation matrix given in the form $\hat{Q}_1 = I - \gamma uu^T$, where $u = a - h = (a_1 + \sigma, a_2, a_3, \ldots, a_n)^T$ and $\gamma = 2/\|u\|_2^2$. Thus

$$\hat{B}_1 = (I - \gamma uu^T)\hat{B}(I - \gamma uu^T)$$

$$= \hat{B} - \gamma \hat{B}uu^T - \gamma uu^T\hat{B} + \gamma^2 uu^T\hat{B}uu^T \qquad 6.2.1$$

The terms in this expression admit considerable simplification [93] if we introduce the auxiliary vector

$$v = -\gamma \hat{B}u. \qquad 6.2.2$$

Thus

$$-\gamma \hat{B}uu^T = vu^T, \; -\gamma uu^T\hat{B} = uv^T \qquad 6.2.3$$

and

$$\gamma^2 uu^T\hat{B}uu^T = -\gamma uu^T vu^T. \qquad 6.2.4$$

Introducing the scalar

$$\delta = -\frac{1}{2}\gamma u^T v, \qquad 6.2.5$$

we can rewrite this last term as $2\delta uu^T$. Thus

$$\hat{B}_1 = \hat{B} + vu^T + uv^T + 2\delta uu^T \qquad 6.2.6$$

The final manipulation is to split the last term into two pieces in order to combine one piece with the term vu^T and the other piece with the term uv^T. Specifically, let

$$w = v + \delta u \tag{6.2.7}$$

Then

$$\hat{B}_1 = \hat{B} + wu^T + uw^T \tag{6.2.8}$$

In the algorithm below $B(k, j:n)$ denotes the vector consisting of the elements of $B(k,i)$, $i = j, \ldots, n$, and the k^{th} Householder vector u is stored in $B(k+1:n, k)$.

We assume that the *Input* for the algorithm is the lower triangular portion of a real symmetric matrix B, which is updated

$$B = \begin{pmatrix} b_{11} & & & & \\ b_{21} & b_{22} & & & \\ b_{31} & b_{32} & b_{33} & & \\ \vdots & \vdots & \ddots & \ddots & \\ b_{n1} & b_{n2} & \ldots & b_{nn-1} & b_{nn} \end{pmatrix}.$$

The *Output* is the lower diagonal of a symmetric tridiagonal matrix T represented by two vectors.

$$T = \begin{pmatrix} \alpha_1 & & & & \\ \beta_1 & \alpha_2 & & & \\ & \beta_2 & \alpha_3 & & \\ & & \ddots & \ddots & \\ & & & \beta_{n-1} & \alpha_n \end{pmatrix}$$

In displaying input and output matrices, our convention is that empty portions of the matrices are always zero. The above algorithm code segment may be written as

Algorithm
for $k = 1 : n - 2$
  $\alpha\ (k) = B(k, k)$
  $m = max(|B(j, k)|,\ j = k + 1 : n)$
  if $(m \neq 0)$
    $B(k + 1 : n, k) = B(k + 1 : n, k)/m$
    $\beta\ (k) = \sqrt{\sum_{i=k+1}^{n} B(i, k)^2}$
    if $(B(k + 1, k) < 0)$
      $\beta(k) = -\beta(k)$
    $B(k + 1, k) = B(k + 1, k) + \beta(k)$
    $\gamma(k) = 1/(\beta(k) * B(k + 1, k))$
    $\beta(k) = \beta(k) * m$
    (Set up the Householder transformations)
    $wa = 0$
    for $j = k + 1 : n$
      $wa = a(j, j) * B(j, k)$
      $v(j + 1 : n) = v(j + 1 : n) + B(j + 1 : n, j) * B(j, k)$
      $wa = wa + B(j + 1 : n, j)^T * B(j + 1 : n, k)$
      $v(j) = v(j) + wa$
    endfor
    (Calculate $\hat{B}u$, where u is stored in $B(k + 1 : n, k)$)
    $\delta = 0$
    $v(k + 1 : n) = -\gamma(k) * v(k + 1 : n)$
    $\delta = \delta + v(k + 1 : n)^T * B(k + 1 : n, k)$
    $\delta = \gamma(k) * \delta/2$
    $w(k + 1 : n) = v(k + 1 : n) + \delta * B(k + 1 : n, k)$
    (Calculate w (this could overwrite v instead))
    for $j = k + 1 : n$
      $B(j : n, j) = B(j : n, j) + w(j : n) * B(j, k)$
        $+B(j : n, k) * w(j)$
    endfor
    (Update B)
  endif
endfor

$\alpha\ (n-1) = -B(n-1, n-1)$
$\alpha\ (n) = B(n,n)$ // The main-diagonal of the output matrix
$\beta\ (n-1) = -B(n, n-1)$ // The off-diagonal of the output matrix

The implementations store the main-diagonal entries of the tridiagonal matrix T in a separate one-dimensional array (α) and the off-diagonal entries in a one-dimensional array (β).

The second step of the reduction has no effect on the first row and column of B_1. This was not so in the non-symmetric case as there the first row needs to be altered; it is this difference that makes the symmetric reduction *less than* half as expensive as the non-symmetric reduction [93]. The second step is identical to the first step, except that it acts on the submatrix $\hat{B}_1$.

Figure 6.1 represents the dataflow for this algorithm. This dataflow diagram shows that in the first stage, the update to the components of the vectors by u_j and v_i $(i, j = k, k+1, \ldots, n)$ are carried out. In the second stage, the computation of the scalar δ is carried out. This is a crucial point as it is needed for the computation of the vector w_i $(i = k, k+1, \ldots, n)$, in the third stage. The final stage, the update to $\hat{B}$ using $u_k, u_{k+1}, \ldots, u_n$ and $w_k, w_{k+1}, \ldots, w_n$ is denoted by $\hat{B}_k$. After the k^{th} reduction step the $(k+1)^{th}$ step starts to update the reduction in the same way as the k^{th} step. These observations will be made use of in Algorithm I described below.

We consider three slightly different versions for the reduction of a symmetric matrix to tridiagonal form. As only the lower triangle of B is stored the scalar products in forming Bu split into two parts. The first part uses the part of the row in the lower triangle of B. The other part corresponding to the upper triangle uses its reflection, that is part of a column of B. This mean that every element of the lower triangle B

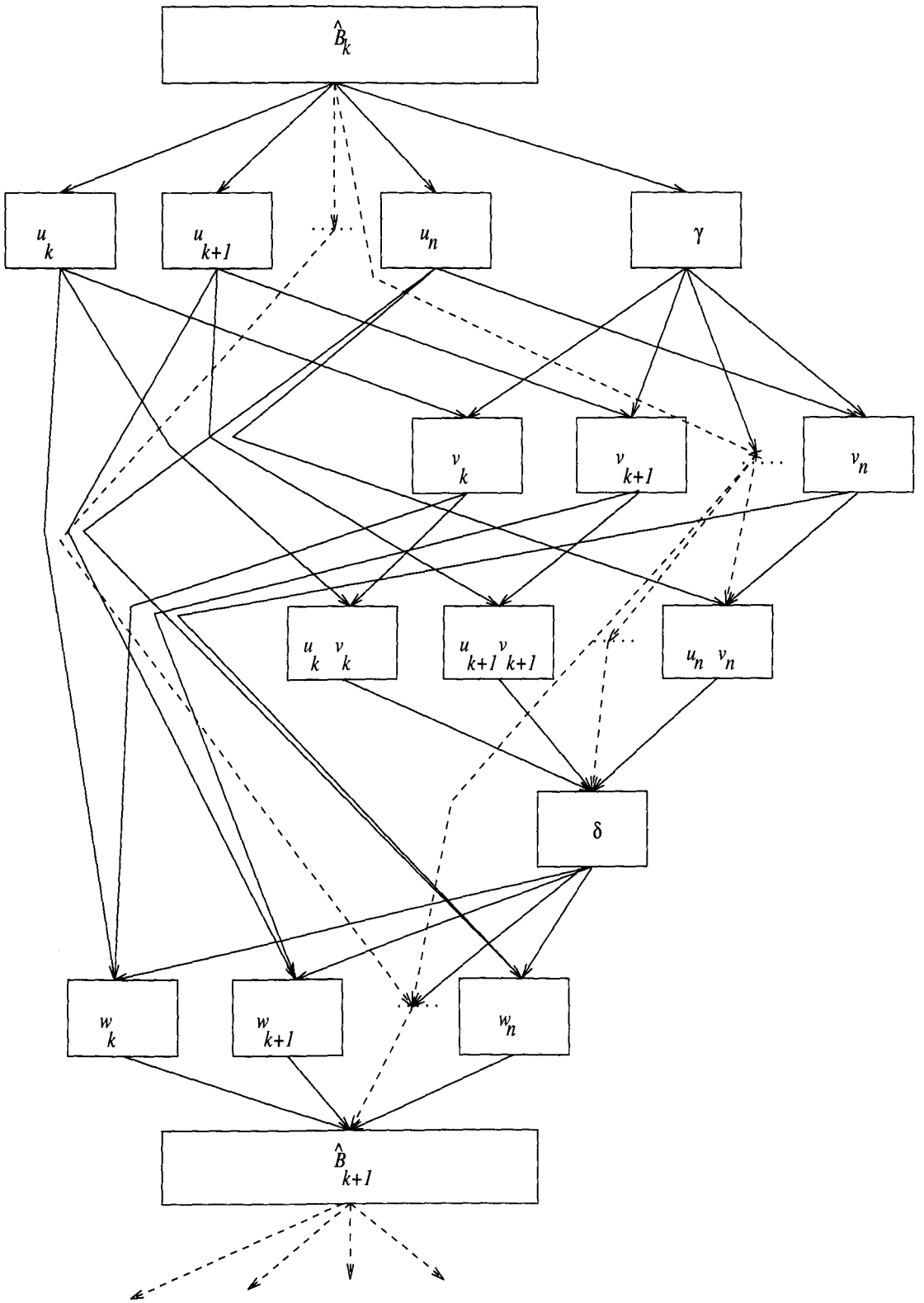


Fig. 6.1 Dataflow Diagram for Tridiagonal Form

contributes to two different scalar products. The first version uses a modification of the Watkins algorithm [93] so that each element of B is used only once in forming the scalar products Bu. The second version used a further modification where all contributions to each scalar product in Bu were calculated together. In the third version the formation of the scalar products Bu, the multiplication of them by a constant scalar $-\gamma$ (i.e. to find the vector v) and the calculation of δ are included in the same loop.

6.3 Parallel Implementations

The sequential algorithms have been described above. Here we will consider three types of parallel implementation. In all these implementations, the tasks are allocated in a predetermined (scattered) ordering. This allows the pre-allocation of tasks to threads.

In summary, in all three implementations the update of the pivotal column is carried out sequentially as shown in figure 6.1. As soon as the pivotal column has been updated, the computation of the matrix vector products $\hat{B}_k u_j$ can proceed. These need to be synchronised in the first implementation because different threads contribute to the same scalar product but synchronisation is not necessary for the other implementations. The process is continued until all the scalar products required for $\hat{B}_k u_j$ are completed. After that the resulting vector is multiplied by the constant $-\gamma$ to get the vector v using 6.2.2. As the scalar product $u^T v$ is formed in parallel, we store the partial result during the calculation of the scalar product and then add the partial sums to the final total using a lock. Then the scalar product $u^T v$ is multiplied by a constant $-\frac{1}{2}\gamma$ to get the scalar δ using 6.2.5. This is done sequentially in all parallel implementations.

It is clear from figure 6.1 that the calculation of δ is a bottle-neck as the vector w_i can not be updated using 6.2.7 before its completion. After

this computation of δ, the components of the vector w are updated. Notice that the computation of the vector w components are independent of one another and that they can be performed in parallel without using a lock.

Finally, from 6.2.8 we can see that $\hat{B}_{k+1}$ can be updated either after each element of the vector w is computed or after all components of vector w have been evaluated. Following this, the next pivotal column can be treated in the same way and the whole process is repeated for the next stage.

The first implementation starts by carrying out the pivotal column updates sequentially. This is followed by the matrix vector product updates 6.2.2 done in parallel (i.e. forming v). Updating the matrix vector product needs a lock because different threads contribute to the same scalar product. When this is all completed then the calculation of the scalar δ using 6.2.5 is carried out in parallel and so we store the partial results which contribute to the scalar δ. We used a lock in order to prevent simultaneous contribution of the partial result to δ by the different threads. The multiplication of the scalar δ by $-\gamma/2$ is carried out sequentially. This is followed by forming the vector w using 6.2.7. B_1 is updated in parallel after all components of the vector w have been evaluated using 6.2.8

For the calculation of $\hat{B}u$ the parallel section is terminated with "THREADjoin"s which ensures that all the components of v are computed before commencing the calculation of the scalar δ using 6.1.1.5 which is done in parallel. The algorithm is as follows:

Algorithm I

prcount is the thread number
prcs is number of processor (threads)

For $k = 1 : n - 2$
(i) (pivotal column updates sequentially, as in sequential algorithm)

for $(i = k + 1;\ i <= n;\ i++)$
$v(i) = 0.0$
$\delta\ = 0.0$

START PARALLEL
double $wa = 0.0$
(ii) for $(j = k + prcount;\ j <= n;\ j = j + prcs)$
$wa = a(j, j) * b(j, k)$
for $(i = j + 1;\ i <= n;\ i++)$
lock()
$v(i) = v(i) + b(i, j) * b(j, k)$
unlock()
$wa = wa + b(i, j) * b(i, k)$
endfor
lock()
$v(j) = v(j) + wa$
unlock()
endfor
FINISH PARALLEL
(Matrix vector product using 6.2.2 updates in parallel)

START PARALLEL
double $sa = 0.0$
(iii) for $(i = k + prcount;\ i <= n;\ i = i + prcs)$
$v(i) = -\gamma(k) * v(i)$
$sa = sa + v(i) * b(i, k)$
endfor
lock()
$\delta\ = \delta\ + sa$
unlock()
FINISH PARALLEL
(Compute the scalar δ using 6.2.5 in parallel)

$\delta\ = -\gamma\ * \delta\ /2$

START PARALLEL
(iv) for $(i = k + prcount;\ i <= n;\ i = i + prcs)$
$w(i) = w(i) + \delta\ * b(i, k)$
endfor
FINISH PARALLEL
(Compute the vector w using 6.2.7 in parallel)

START PARALLEL

(v) for $(j = k + prcount;\ j <= n;\ j = j + prcs)$
for $(i = j;\ i <= n;\ i++)$
$b(i,j) = b(i,j) + w(i) * b(j,k) + b(i,k) * w(j)$
endfor
FINISH PARALLEL
(Update B using 6.2.8 in parallel)
Endfor

Synchronisation is required when the scalar δ is updated using 6.2.5, because different threads contribute to the sums. This is done by first computing partial results, followed by global summation of the partial results. This synchronisation is necessary to avoid two processors attempting to add into the sum at the same time and is carried out using a lock. When all contributions to δ have been added the parallel section is terminated. We then update w using 6.2.7 in parallel and terminate this parallel section. In the last stage we update B_1 using 6.2.8 in parallel. This parallel implementation uses four parallel sections and one lock which is used in three places. The first implementation provides a basis for comparison with the other implementations.

In the second implementation the algorithm is modified so that some data dependency is avoided. The important point is that in the second version in the calculation of Bu each processor only writes to one element of v but each B value is read twice. These updates are independent and can hence be carried out in parallel without the use of a lock. When a thread finds that no more updates to a matrix-vector product Bu are completed, then the parallel section is terminated. The calculation of the scalar δ using 6.2.5, w using 6.2.7 and updating $\hat{B}_1$ using 6.2.8 are done in parallel as in the first implementation. In this implementation we use four parallel sections and one *lock* used in only one place.

Algorithm II

prcount is the thread number
prcs is number of processor (threads)

For $k = 1 : n - 2$
(i) (pivotal column updates sequentially, as in sequential algorithm)
START PARALLEL
(ii) (Matrix vector product using 6.2.2 updates in parallel)
for $(j = k + prcount;\ j <= n;\ j = j + prcs)$
$v(j) = 0$
for $(i = k + 1;\ i <= j;\ i + +)$
$v(j) = v(j) + b(j, i) * b(i, k)$
for $(i = j + 1;\ i <= n;\ i + +)$
$v(j) = v(j) + b(i, j) * b(i, k)$
endfor
FINISH PARALLEL

(iii) (Multiply v by γ and compute the scalar δ
using 6.2.5 in parallel, as in *Algorithm I* part (iii)).

(iv) (Compute the vector w using 6.2.7 in parallel,
as in *Algorithm I* part (iv)).

(v) (Update B using 6.2.8 in parallel, as in *Algorithm I* part (v)).

Endfor

In the third implementation we combined the calculation of the products $\hat{B}u$ and the scalar δ. We use a *lock* when updating δ because two processors might add into the sum of δ at the same time. The rest of the algorithm which finds w using 6.2.7 and $\hat{B}_1$ using 6.2.8 are parallelised and terminated in the same way as in the first implementation. We expected that this implementation will be more efficient than the others because combining the matrix-vector product and the evaluation of the scalar δ decreases the number of parallel sections. We use three parallel sections and one *lock* for this implementation.

Algorithm III

prcount is the thread number
prcs is number of processor (threads)

For $k = 1 : n - 2$
(i) (pivotal column updates sequentially, as in sequential algorithm)

$\delta = 0.0$

(ii) (Matrix vector product using 6.2.2 and the scalar δ using 6.2.5 are combined (i.e. (ii) and (iii)) and update in parallel)

START PARALLEL
double $sa = 0.0$
for $(j = k + prcount;\ j <= n;\ j = j + prcs)$
$v(j) = 0$
for $(i = k + 1;\ i <= j;\ i++)$
$v(j) = v(j) + b(j,i) * b(i,k)$
for $(i = j + 1;\ i <= n;\ i++)$
$v(j) = v(j) + b(i,j) * b(i,k)$
$v(j) = -\gamma(k) * v(j)$
$sa = sa + v(j) * b(j,k)$
endfor
lock()
$\delta = \delta + sa$
unlock()
FINISH PARALLEL

(iii) (Compute the vector w using 6.2.7 in parallel, as in *Algorithm I* part (iv)).

(iv) (Update B using 6.2.8 in parallel, as in *Algorithm I* part (v)).
Endfor

6.4 Experimental Results

Results illustrating the performance of the algorithms presented in this chapter are reported in this section. These tests used both the simple Matrix class used for a general Matrix and a special symmetric Matrix class written

so that only half the matrix is stored. With the general Matrix class only the lower triangular part is used which wastes storage. These classes were used to access the matrices and the computations were carried out using both row and column representations of the matrix. This was made easy by the use of the C++ class facility.

The storage allocation is illustrated here using the elements of the matrix to indicate the subscript used in the one dimensional array. Both row and column storage are illustrated. In the illustrations the elements of the matrices represent the subscripts for the corresponding elements in the one dimensional array used for storage. For the $n \times n$ symmetric matrix only the lower triangular part was considered.

The $n \times n$ lower triangular matrix defined by

$$B_{ij} = \begin{cases} 0, & j > i \\ \mathrm{B}_{ij}, & j \leq i \end{cases},$$

may be represented using a one-dimensional array ordered for the representation by rows as follows:

$$B_{row} = \begin{pmatrix} 0 & & & & \\ 1 & 2 & & & \\ 3 & 4 & 5 & & \\ 6 & 7 & 8 & 9 & \\ 10 & 11 & 12 & 13 & 14 \end{pmatrix}.$$

The same matrix can be represented by columns as follows:

$$B_{column} = \begin{pmatrix} 0 & & & & \\ 1 & 5 & & & \\ 2 & 6 & 9 & & \\ 3 & 7 & 10 & 12 & \\ 4 & 8 & 11 & 13 & 14 \end{pmatrix}.$$

The C++ simple matrix class and symmetric matrix class only vary in the way the matrix is stored. In the symmetric matrix class, only the lower part of the matrix is stored but the calculation of the subscripts for the matrix elements needs more arithmetic than the simple matrix class; so it requires less storage but needs extra arithmetic. The value of the subscript needed for the simple matrix class is calculated as $position = (i-1)*n+j-1$ for the row representation and $position = (j-1)*n+i-1$ for the column representation. The calculation of the subscript needed for the symmetric matrix class is $position = (i*(i-1)/2)+j-1$ for the row representation and $position = ((j-1)*(2*n-j+2)/2)+i-j$ for the column representation, where i and j are indices of the matrix element and n is the matrix size.

In the results we used the notation Tri for our parallel implementations of reduction to tridiagonal form. The numerical results were obtained for the three versions outlined in section 6.3. The first method (the simple implementation) is indicated by $1p$, the second implementation by $2p$ and the third implementation by $3p$. The representation of the matrix by columns is indicated by the final character t, otherwise the representation is by rows. The elapsed time for the reduction of a real symmetric matrix to tridiagonal form using each of the implementations was measured. Comparisons were also carried out both with and without array bound checking.

We tested the algorithms using one to ten processors (1,2,4,6,8,10) with matrices of sizes 100(100)500. In calculating the results using array bound checking for the symmetric matrix class we used a sequential version of the algorithm ($Tri2$) for comparison. For the simple matrix class we used a sequential version of the algorithm ($Tri1t$). Similarly, for the results without array bound checking and using the symmetric matrix class we used a sequential version of the algorithm ($Tri3t$). These sequential versions gave times which were slightly better than those for other versions. Each

efficiency calculation was carried out with the same representation and the same compiler options. Normally, we expect the efficiency to be less than one.

To show the performance of the three different parallel versions and their representations of the matrix by columns and rows, we plot, in figures 6.2 and 6.3, mean efficiencies against number of processors for the simple C++ matrix class, and in figures 6.4 and 6.5 mean efficiencies against number of processors for the C++ symmetric matrix class. Figures 6.6 and 6.7 show actual efficiencies using 2 processors for the simple C++ matrix class with and without array bound checking and figures 6.10 and 6.11 show similar plots for 6 processors. Figures 6.8 and 6.9 show actual efficiencies using 2 processors for the C++ symmetric matrix class with and without array bound checking and figures 6.12 and 6.13 show similar plots for 6 processors.

It can be seen from figures 6.6, 6.7, 6.8, and 6.9 that there is only a very small increase in the efficiency values as matrix size increases. Even when there is an increase in the efficiency (for example with $Tri2p$, $Tri2pt$ and $Tri3p$, $Tri3pt$), this is followed by a fall in some cases (for example with or without array bound checking, the symmetric matrix class using 2 processors performs as in figures 6.8 and 6.9). On the other hand, the efficiency increases slightly with the size of matrix for the simple matrix class (see figures 6.6 and 6.7) with or without array bound checking.

The results generally confirm our expectations, particularly when we use the representation of the matrix by rows or by columns (i.e. the transpose of the row representation of the matrix). In most cases the different representations gave rise to very close efficiency curves when these tests used the simple matrix class. This is because similar subscript arithmetic was involved for the row and column representation and the other computations would be the same with the representations of the matrix columns as well as

rows. On the other hand, when these tests use the C++ symmetric matrix class the efficiency curve of the transposed versions is marginally better than the efficiency curve of the row versions. This improvement did not surprise us because as we can see from the formulae for the subscripts of the matrix, the calculation of the subscript requires extra arithmetic operations for the column representation of the matrix.

Mean Efficiency						
pr	Tri1p	Tri1pt	Tri2p	Tri2pt	Tri3p	Tri3pt
1	0.946	0.950	0.904	0.922	0.913	0.925
2	0.572	0.578	0.828	0.841	0.829	0.844
4	0.317	0.321	0.714	0.726	0.728	0.740
6	0.193	0.196	0.602	0.614	0.625	0.640
8	0.129	0.131	0.495	0.505	0.529	0.537
10	0.093	0.093	0.399	0.396	0.441	0.449

Table 6.1 No Check with Matrix Class.

Tables 6.1 and 6.2 show the mean efficiency of all the three versions. The numbers in the tables are efficiency values based on times used to complete the reduction to tridiagonal form of an $n \times n$ symmetric matrix using a simple matrix class and symmetric matrix class. It is obvious from tables 6.1 and 6.2 that the difference between the two classes is small. The efficiency with the symmetric matrix class is marginally better than that with the simple matrix class using 1, 2, 4, and 6 processors for *Tri2pt* and *Tri3pt* versions but not for 8 and 10 processors. Also with the symmetric matrix class the efficiency is marginally better than that with the simple matrix class using 1 processor for *Tri1p* and *Tri1pt* versions but not using the other number

of processors. In other cases, the efficiency values with simple matrix class is marginally better than that with symmetric matrix class for all the other versions (except for *Tri2p* with 4 processors where efficiencies are almost identical).

Mean Efficiency						
pr	Tri1p	Tri1pt	Tri2p	Tri2pt	Tri3p	Tri3pt
1	0.937	0.890	0.944	0.872	0.957	0.906
2	0.628	0.631	0.835	0.752	0.897	0.814
4	0.364	0.359	0.712	0.681	0.771	0.716
6	0.240	0.236	0.620	0.591	0.673	0.632
8	0.169	0.167	0.525	0.508	0.585	0.551
10	0.121	0.121	0.425	0.425	0.502	0.474

Table 6.2 No Check with Symmetric Matrix Class.

For the implementations using the simple matrix class the lower and upper triangular parts of matrix B are both stored but we need only update the main diagonal and lower triangular matrix, whereas, for the symmetric matrix class a lower triangular matrix was used to store B without storing the elements above the main diagonal. This may have affected the storage pattern but we do not have clear evidence for this. This is shown in table 6.2.

It is obvious from the efficiency graphs that the first implementation using all representations of the matrix gave very poor times even for large matrix sizes and numbers of processors.

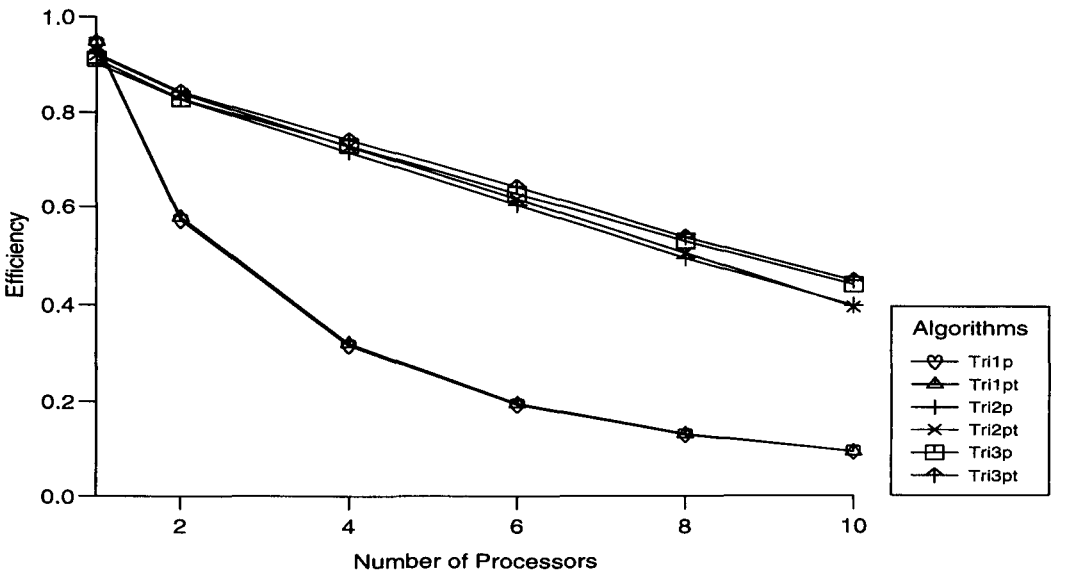


Fig. 6.2 No Check with Simple Matrix Class

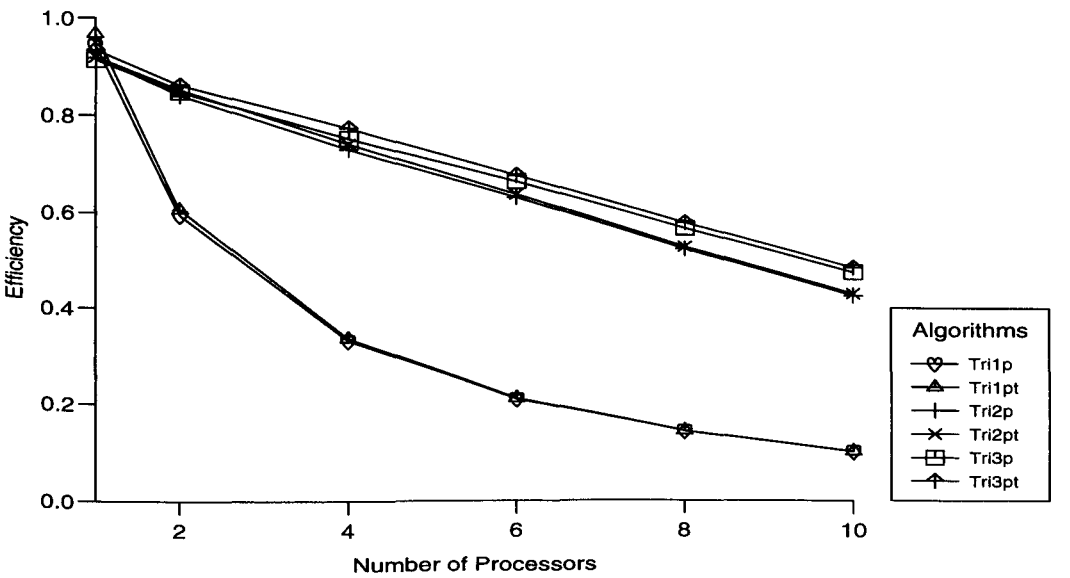


Fig. 6.3 Check with Simple Matrix Class

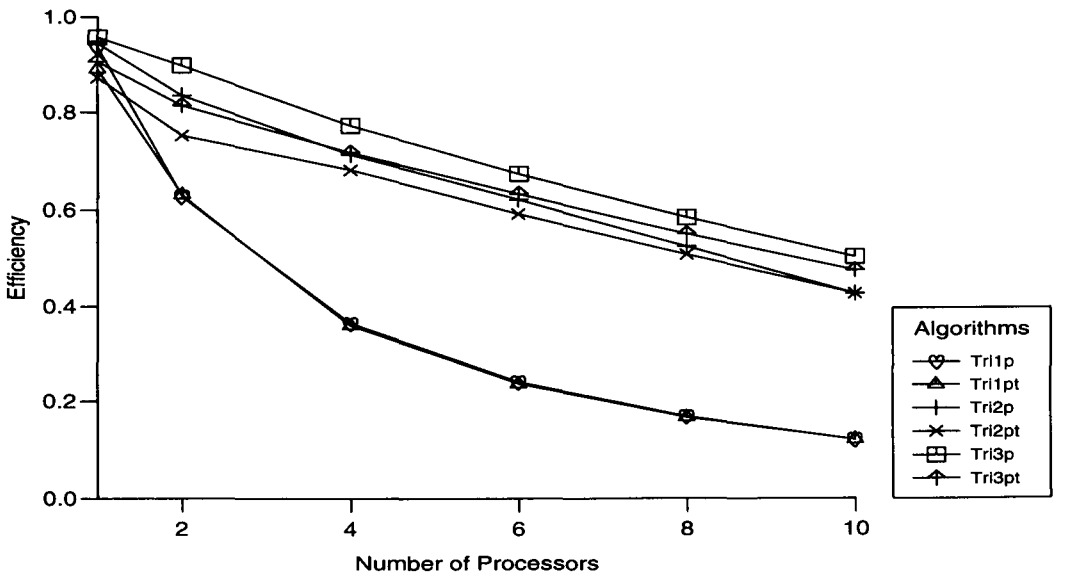


Fig. 6.4 No Check with Symmetric Matrix Class

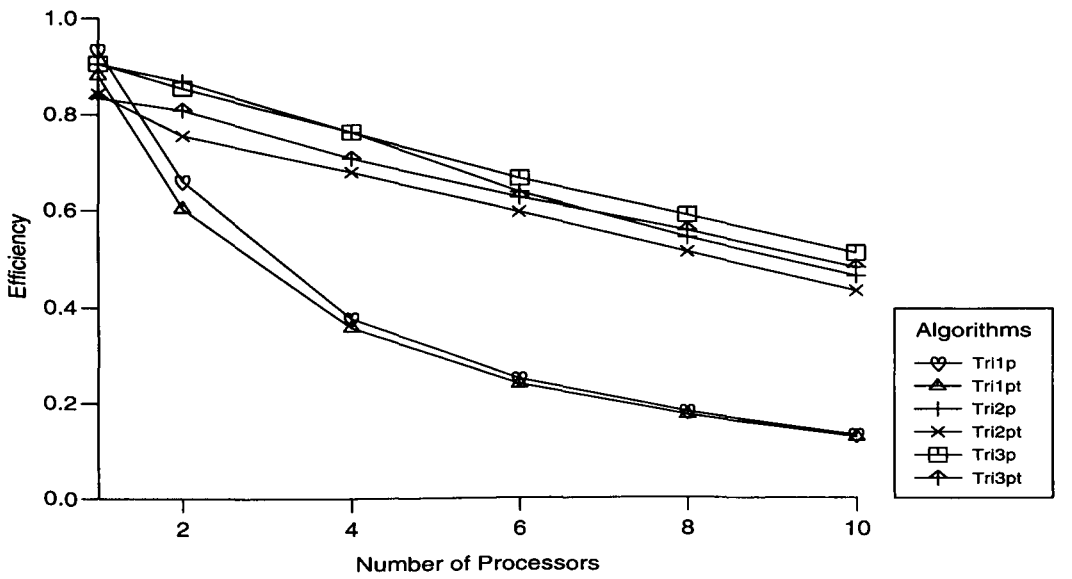


Fig. 6.5 Check with Symmetric Matrix Class

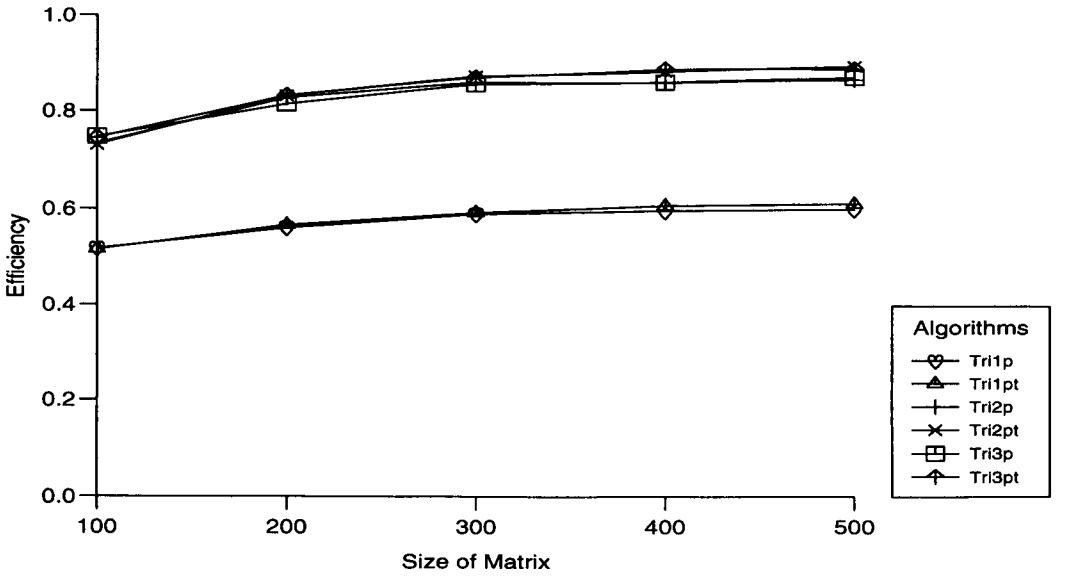


Fig. 6.6 No Check with Simple Matrix Class for 2 Processors

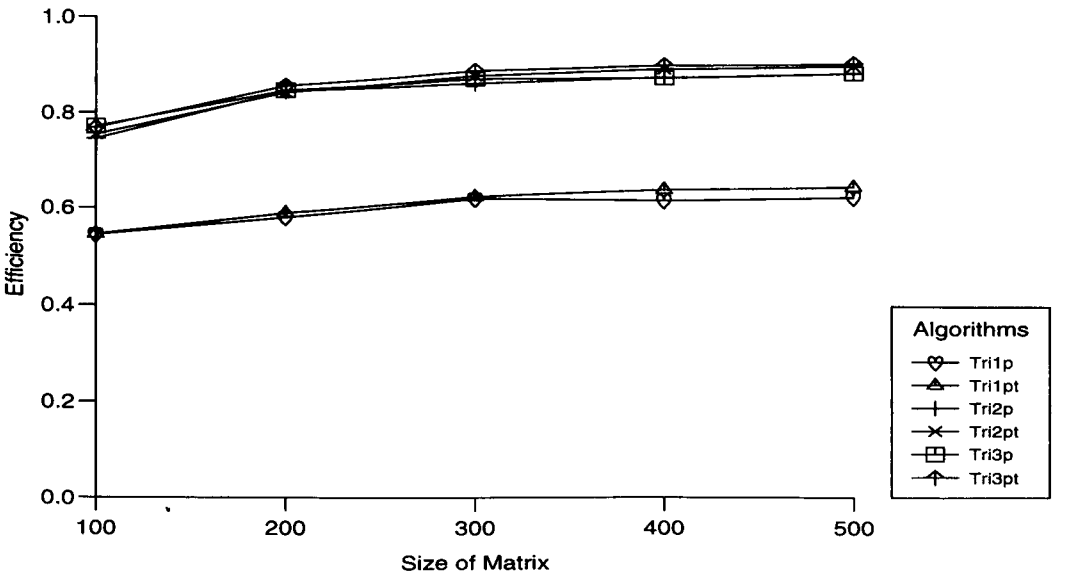


Fig. 6.7 Check with Simple Matrix Class for 2 Processors

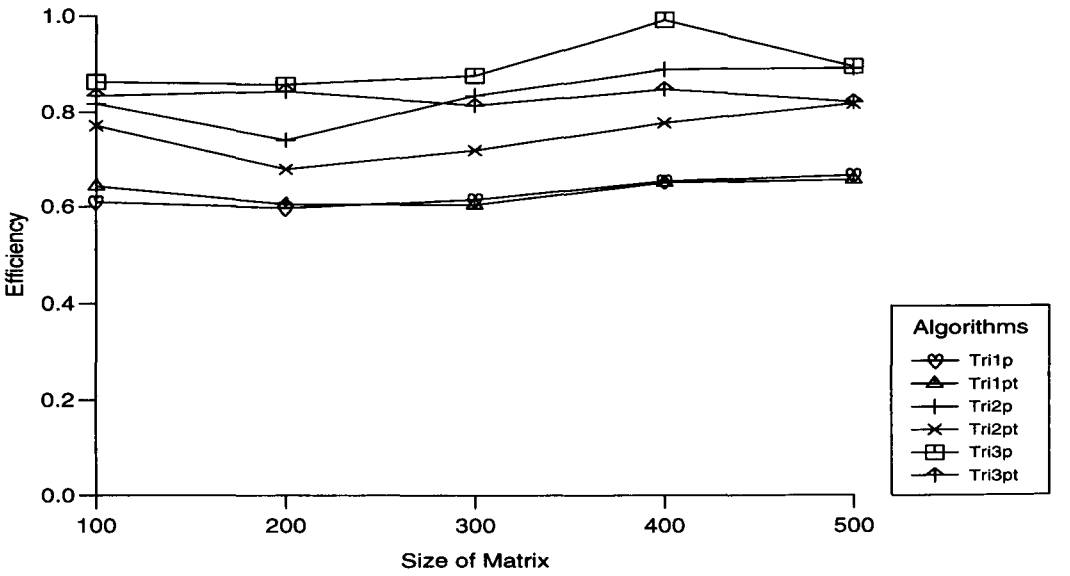


Fig. 6.8 Check with Symmetric Matrix Class for 2 Processors

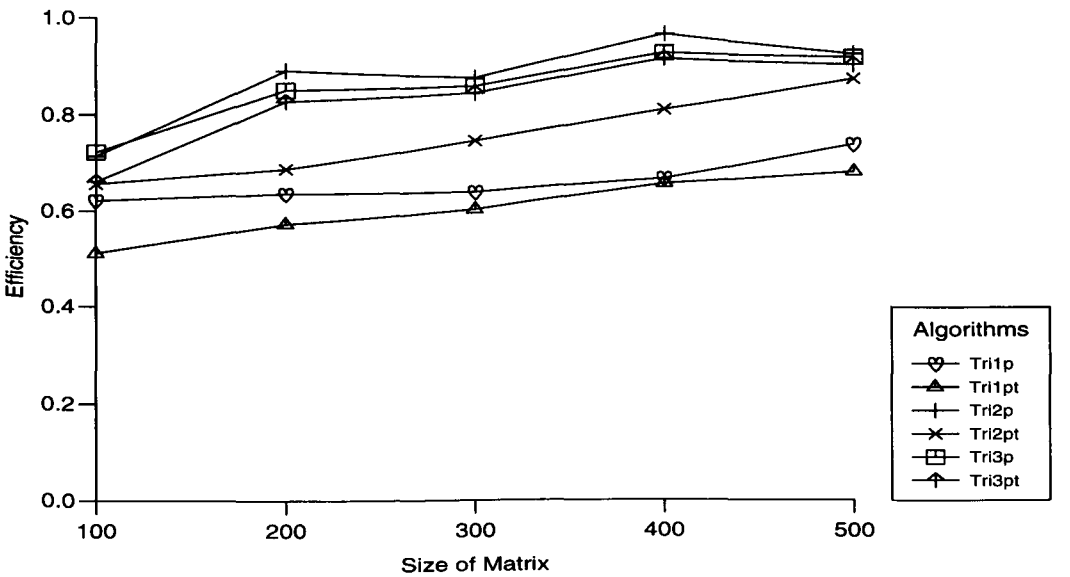


Fig. 6.9 Check with Symmetric Matrix Class for 2 Processors

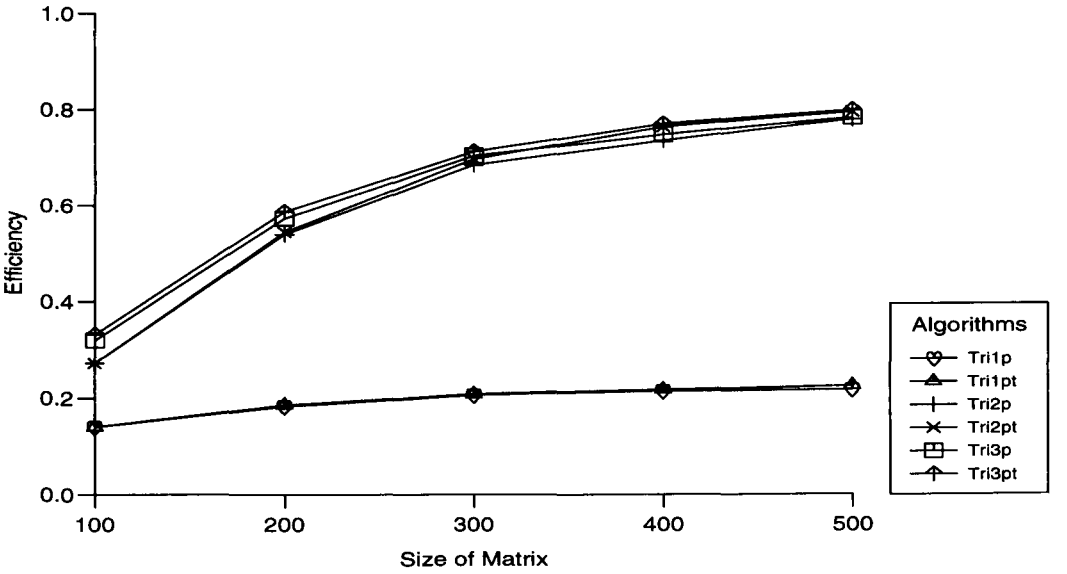


Fig. 6.10 No Check with Simple Matrix Class for 6 Processors

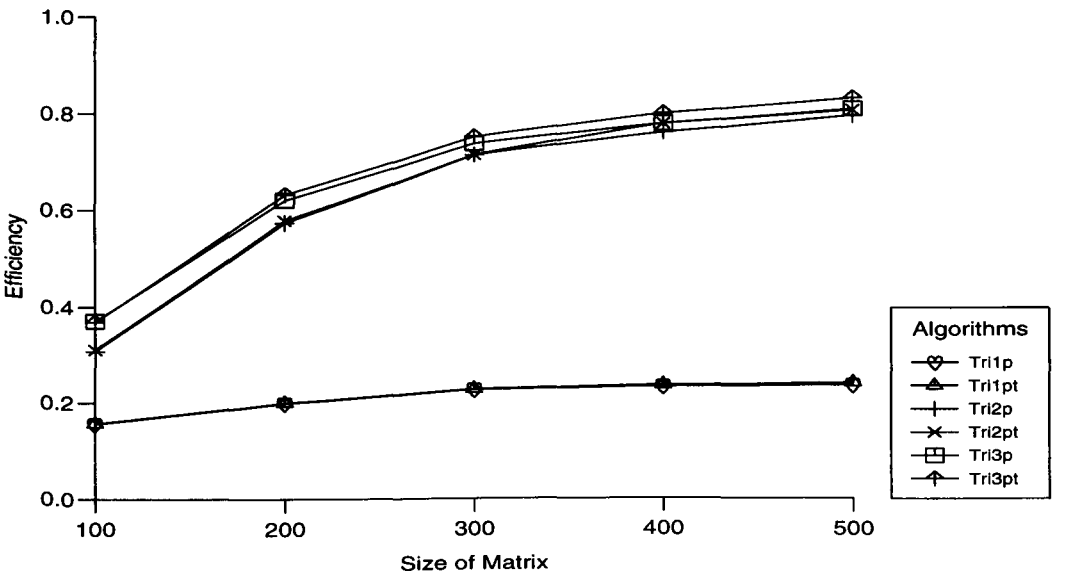


Fig. 6.11 Check with Simple Matrix Class for 6 Processors

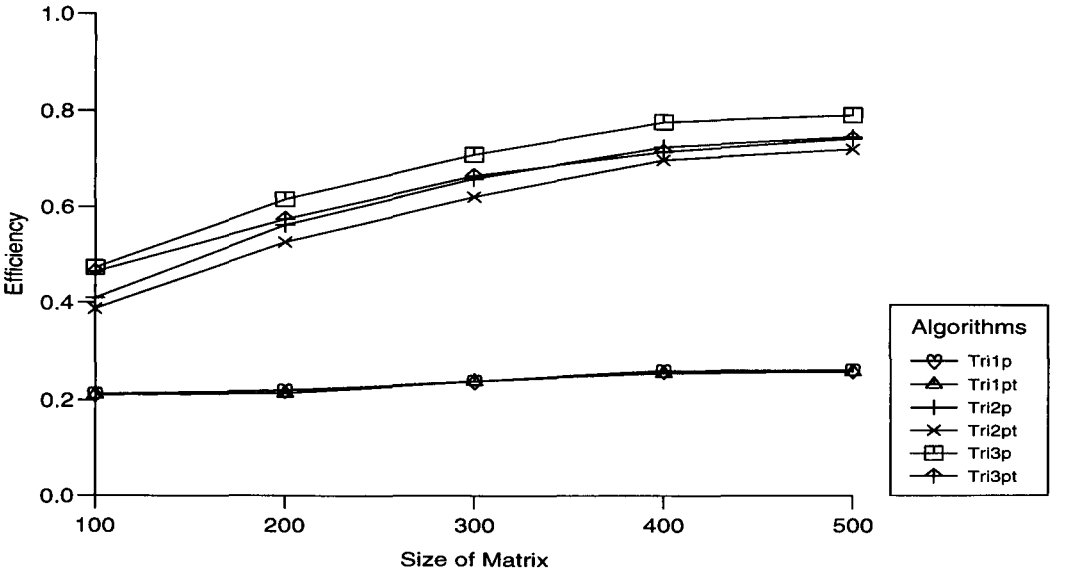


Fig. 6.12 No Check with Symmetric Matrix Class for 6 Processors

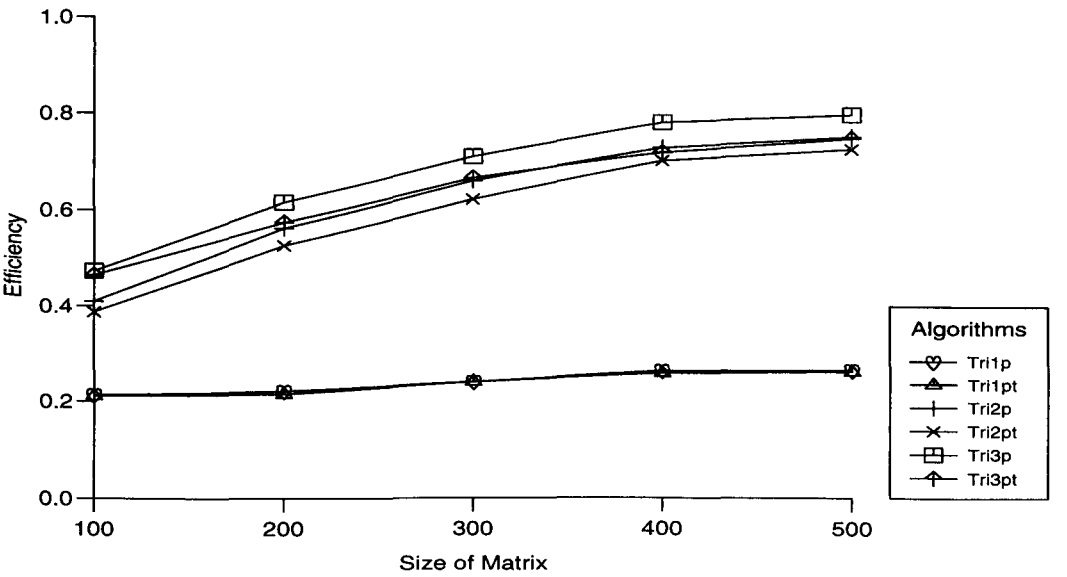


Fig. 6.13 Check with Symmetric Matrix Class for 6 Processors

In addition, in most cases this implementation ($Tri1p$ and $Tri1pt$) gave rise to almost indistinguishable efficiency curves. As we mentioned in the section 6.2, this is because this version has greater data dependencies which limit the exploitation of the parallel system. As we can see from the efficiency graphs in figures 6.10-13, the performance of $Tri1p$ and $Tri1pt$ show little improvement as the matrix size and the number of processors increase. On the other hand, for the second and third implementations ($Tri2p$, $Tri2pt$ and $Tri3p$, $Tri3pt$) the efficiency improves as matrix sizes increase.

6.5 Conclusions

In this chapter we have investigated some parallel implementations of an algorithm for the reduction of a real symmetric matrix to tridiagonal form.

As expected, the first implementation ($Tri1p$) is slower than all the other versions. This is possibly because in this algorithm more $THREADjoins$ are needed. The other reason is that the algorithm uses an extra lock which may also lead to extra waiting.

The efficiency graphs show that the difference between the $Tri2p$ and $Tri3p$ implementations is very small, may be less than possible errors in measurement, though $Tri3p$ seems to be marginally better. It is notable that the third version for larger numbers of processors and for larger sizes of matrix gives marginally better results than the second version. This may be because of using an extra set of $THREADjoins$ in the second version.

Even with the extra subscript arithmetic the results of implementations using the symmetric matrix class are marginally better than those using the simple matrix class. The reason for the marginal improvement is possibly due to a smaller storage demand. As we pointed out in the previous section, the calculation of the subscripts for the matrix elements needs more arithmetic operations in the symmetric matrix class than the simple matrix

class.

Finally, it can be concluded from the experimental results of parallelising this algorithm that some minor changes did give some improvements, but all the efficiencies obtained were much poorer than those for algorithms described in chapter 5.

CHAPTER 7

The Symmetric Tridiagonal Eigenproblem

7.1 Introduction

We implement Cuppen's method for finding all of the eigenvalues and corresponding eigenvectors of real symmetric $n \times n$ tridiagonal matrix T. The method uses a partitioning technique which reduces the original problem to smaller ones of the same type, by a rank-one modification. Cuppen [17] observed that there can frequently be deflation in the updating process as the original matrix is rebuilt from the subproblems. Dongarra and Sorensen [26] implemented a further deflation technique to make the algorithm more efficient and more stable.

The implementation in [26] always computes the eigenvalues to high accuracy, but some specific examples [6],[17] and [26] illustrate that it may not compute fully orthogonal eigenvectors. To resolve this problem, Kahan [54] suggests computing some key quantities more accurately using simulated extended precision. Sorensen and Tang [83] presented an alternative

implementation scheme which was inspired by the earlier work of Kahan [54]. They showed that this method is stable but that it requires extended precision and so is machine-dependent [83]. Gu and Eisenstat [42] suggested an alternative method using the same rank-one modification as [83] but a different approach to finding the eigenvectors after all eigenvalues are computed. This new way makes the simulated double precision unnecessary, and they showed that the new method is backward stable.

The rest of this chapter is organised as follows. In section 7.2 we review the description of the divide-and-conquer method presented in [17]. In section 7.3 we discuss computing eigenvalues and eigenvectors of a rank one modification of a diagonal matrix and exceptional cases are considered in section 7.4. In section 7.5 we examine the arithmetic complexity of the divide-and-conquer method. In section 7.6 we discuss a number of sequential algorithms based on recursive and non-recursive versions. We consider parallel implementations of these versions in section 7.7 of which four are recursive and two are non-recursive. In addition we also consider parallelisation of the matrix multiplication part of the algorithms in section 7.7.3. Section 7.8 presents results of some numerical results illustrating an experimental evaluation of the effect of deflation on accuracy, comparison of the parallel implementations and comparison of the additional parallelisation for matrix multiplication. Conclusions are given in section 7.9.

7.2 Cuppen's Divide-and-Conquer Algorithm

Let us consider a $n \times n$ symmetric tridiagonal matrix T as follows:

$$T = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \beta_{n-1} \\ & & & \beta_{n-1} & \alpha_n \end{pmatrix}.$$

Without loss of generality, we will assume $\beta_i \neq 0$. Using a rank-one modification, Cuppen's Divide-and-Conquer method divides the given symmetric tridiagonal matrix T replacing the original problem by two problems with smaller matrices. The modification of the matrices is the heart of this method.

The divide and conquer approach is to divide the original problem into two simpler subproblems, solve each of these subproblems, and then combine these two to form the solution to the original problem. The subproblems can be solved in the same way and so the whole problem is recursive. As indicated in [17], we can either carry on till we arrive at trivial 1×1 or 2×2 eigenvalue problems, or using an alternative method such as QR to calculate the eigenvalues and the eigenvector matrix of small $n_0 \times n_0$ blocks. Here we only use the first alternative.

T can be written as

$$T = \begin{pmatrix} \bar{T}_1 & 0 \\ 0 & \bar{T}_2 \end{pmatrix} + \beta \begin{pmatrix} 0 & & & & & & \\ & \ddots & & & & & \\ & & 0 & & & & \\ & & & 1 & 1 & & \\ & & & 1 & 1 & & \\ & & & & & 0 & \\ & & & & & & \ddots \\ & & & & & & & 0 \end{pmatrix}$$

$$= \begin{pmatrix} \bar{T}_1 & 0 \\ 0 & \bar{T}_2 \end{pmatrix} + 2\beta w w^T \qquad (7.2.1)$$

where $\bar{T}_1$ and $\bar{T}_2$ are of order $n_1 \geq 1$ and $n_2 \geq 1$ with $n_1 + n_2 = n$, β is the n_1^{th} off-diagonal element of T and the unit vector w is given by

$$w = \frac{1}{\sqrt{2}}(0 \ \ldots \ 0 \ 1 \mid 1 \ 0 \ \ldots \ 0)^T \qquad (7.2.2)$$

where the non-zeros are in the $i = n_1$ and $i = n_1 + 1$ position.

This may be written $T = \bar{T} + 2\beta w w^T$ where

$$\bar{T} = \begin{pmatrix} \bar{T}_1 & 0 \\ 0 & \bar{T}_2 \end{pmatrix}.$$

Let us assume that the eigenvalues and eigenvectors of $\bar{T}$ are known, that is we have an orthogonal $\bar{Q}$ and diagonal $\bar{D}$ such that $\bar{T} = \bar{Q}\bar{D}\bar{Q}^T$. The eigenvalues of $\bar{T}$ consist of eigenvalues of $\bar{T}_1$ and $\bar{T}_2$. Suppose that the solutions of the eigenproblem for the $\bar{T}_1$ and $\bar{T}_2$ matrices are given by

$$\bar{T}_1 = \bar{Q}_1 \bar{D}_1 \bar{Q}_1^T \ \ and \ \ \bar{T}_2 = \bar{Q}_2 \bar{D}_2 \bar{Q}_2^T \qquad (7.2.3)$$

which gives

$$T = \begin{pmatrix} \bar{Q}_1 \bar{D}_1 \bar{Q}_1^T & 0 \\ 0 & \bar{Q}_2 \bar{D}_2 \bar{Q}_2^T \end{pmatrix} + 2\beta w w^T.$$

For the next part of the process $\bar{D}$ and $\bar{Q}$ need to be reordered so that the new diagonal elements are in ascending order. Suppose the permutation matrix P is such that $P\bar{D}P$ has diagonal elements in order. We first permute the elements of $\bar{D}$ and $\bar{Q}$ in such a way that we will replace the $\bar{D}$ with a new $\tilde{D}$ and the $\bar{Q}$ with a new $\tilde{Q}$. If $\tilde{D} = P\bar{D}P$, then $\bar{D} = P\tilde{D}P$ and $\bar{T} = \bar{Q}P\tilde{D}P\bar{Q}^T$ so if we define $\tilde{Q} = \bar{Q}P$, then we have the following form

$$T = \tilde{Q}\,(\,\tilde{D} + \rho z z^T\,)\,\tilde{Q}^T \tag{7.2.4}$$

where ρ is the scalar 2β and z is the real vector of order n given by

$$z = \tilde{Q}^T w. \tag{7.2.5}$$

From (7.2.4) it suffices to consider finding the eigenvalues and eigenvectors of the matrix $\tilde{D} + \rho z z^T$. A scheme for doing this is outlined in [26]. We will consider this scheme as a numerical approach for calculating the eigensystem in the next section.

Recursion is a very important tool in which a function invokes itself and forms a natural means for implementing divide and conquer algorithms. Such algorithm can simplify the solution of a problem by enabling us easily to divide its solution into manageable pieces. A recursive solution to a problem has to have at least one base case in order to terminate the recursion [80].

The computation consists of two stages for both recursive and non-recursive methods. In the first stage the subdivisions are formed. This stage can also be further divided into two steps. The first step subdivides and modifies the matrix, and in the second step the quadratic equations for sizes the 2×2 eigenvalue problem are solved. In the second stage, the method looks for ways to solve the original problem in terms of the pieces, that is after the solution of the pieces, the sub-solutions must be combined together in this stage.

7.3 Computing Eigenvalues and Eigenvectors of $\tilde{D} + \rho zz^T$

In this section we consider the calculation of the eigenvalues and eigenvectors of a matrix $\tilde{D} + \rho zz^T$. The modification is based on the following theorem due to Wilkinson [95]. Let $C = \tilde{D} + \rho zz^T$, where $\tilde{D}$ is diagonal, $\|z\|_2 = 1$. Let $\tilde{d}_1 \leq \tilde{d}_2 \leq \ldots \leq \tilde{d}_n$ be the eigenvalues of $\tilde{D}$, and let $\lambda_1 \leq \lambda_1 \leq \ldots \leq \lambda_n$ be the eigenvalues of C. Then $\lambda_i = \tilde{d}_i + \rho\mu_i$, $1 \leq i \leq n$, where $\sum_{i=1}^{n} \mu_i = 1$, and $0 \leq \mu_i \leq 1$. Moreover, $\lambda_1 \leq \tilde{d}_1 \leq \lambda_2 \leq \ldots \leq \lambda_n \leq \tilde{d}_n$ if $\rho < 0$ and $\tilde{d}_1 \leq \lambda_1 \leq \tilde{d}_2 \leq \ldots \leq \tilde{d}_n \leq \lambda_n$ if $\rho > 0$. Finally, if the $\tilde{d}_i$ are distinct and all the elements of z are nonzero, then the eigenvalues of C strictly separate those of $\tilde{D}$.

In this section we will assume the $\tilde{d}_i$ are distinct and also that all z_i non-zero. Consider the matrix equation

$$(\tilde{D} + \rho zz^T)q = \lambda q \tag{7.3.1}$$

where λ is an eigenvalue and q is an eigenvector. The above equation can

be written as follows

$$(\tilde{D} - \lambda I)q + \rho(z^T q)z = 0 \tag{7.3.2}$$

where the scalar $z^T q \neq 0$ because of the assumption that $\lambda \neq \tilde{d}_i$. Multiplying (7.3.2) on the left by $(\tilde{D} - \lambda I)^{-1}$, gives

$$q + \rho(z^T q)(\tilde{D} - \lambda I)^{-1} z = 0 \tag{7.3.3}$$

multiplying (7.3.3) by $z^T / z^T q$ on the left as $z^T q \neq 0$, we obtain

$$f(\lambda) \equiv 1 + \rho z^T (\tilde{D} - \lambda I)^{-1} z = 0 \tag{7.3.4}$$

λ must satisfy (7.3.4) and this equation may be written as

$$f(\lambda) \equiv 1 + \rho \sum_{i=1}^{n} \frac{z_i^2}{\tilde{d}_i - \lambda} = 0 \tag{7.3.5}$$

since $(\tilde{D} - \lambda I)^{-1} z = diag(\frac{z_i}{\tilde{d}_i - \lambda})$. This equation which is referred to as the *secular equation* in [38], gives the eigenvalues of $\tilde{D} + \rho z z^T$ as the roots of $f(\lambda) = 0$. There are number of ways to solve the secular equation. For simplicity, we found the numerical solution of equation (7.3.5) by the bisection method. Newton's method is another possibility but since $f(\lambda)$ is a rational function of λ, Bunch et al [6] suggest a method based on rational interpolation. Li [63] suggested a new improved method based on this idea.

Let us look at the behaviour of the function f. It is a rational function with the n distinct poles $\tilde{d}_1, \tilde{d}_2, \ldots, \tilde{d}_n$. The derivative of f is given by

$$f'(\lambda) = \rho \sum_{i=1}^{n} \frac{z_i^2}{(\tilde{d}_i - \lambda)^2}. \tag{7.3.6}$$

If $\rho < 0$, $f'(\lambda)$ is negative for all values of λ which are not poles. Therefore each continuous piece of f is strictly decreasing. As λ approaches the pole $\tilde{d}_i$, the function is dominated by the i^{th} term $\frac{z_i^2}{(\tilde{d}_i - \lambda)}$. Since each of the terms $\frac{z_i^2}{(\tilde{d}_i - \lambda)}$ tends to zero as $\lambda \to \pm\infty$, so

$$\lim_{\lambda \to \pm\infty} f(\lambda) = 1,$$

since also

$$\lim_{\lambda \to \tilde{d}_i -} f(\lambda) = -\infty$$

and

$$\lim_{\lambda \to \tilde{d}_i +} f(\lambda) = +\infty \ ,$$

it follows that if $\rho < 0$ then the secular equation $f(\lambda) = 0$ has exactly one solution between each pair of poles and one additional solution to the left. Let the zeros of $f(\lambda)$ (i.e. the eigenvalues of $\tilde{D} + \rho z z^T$) be denoted by $\lambda_1 < \lambda_2 < \ \ldots \ < \lambda_n$ [40]. This allows us to conclude that if $\rho < 0$ then f has exactly n roots, one in each of the intervals

$$(-\infty, \tilde{d}_1), (\tilde{d}_1, \tilde{d}_2), \ldots \ , (\tilde{d}_{n-1}, \tilde{d}_n).$$

A similar argument shows that if $\rho > 0$ then f has precisely n roots, one in each of the intervals

$$(\tilde{d}_1, \tilde{d}_2), \ldots \ , (\tilde{d}_{n-1}, \tilde{d}_n), (\tilde{d}_n, \infty).$$

For $\rho < 0$ the graph of $f(\lambda)$ takes the form:

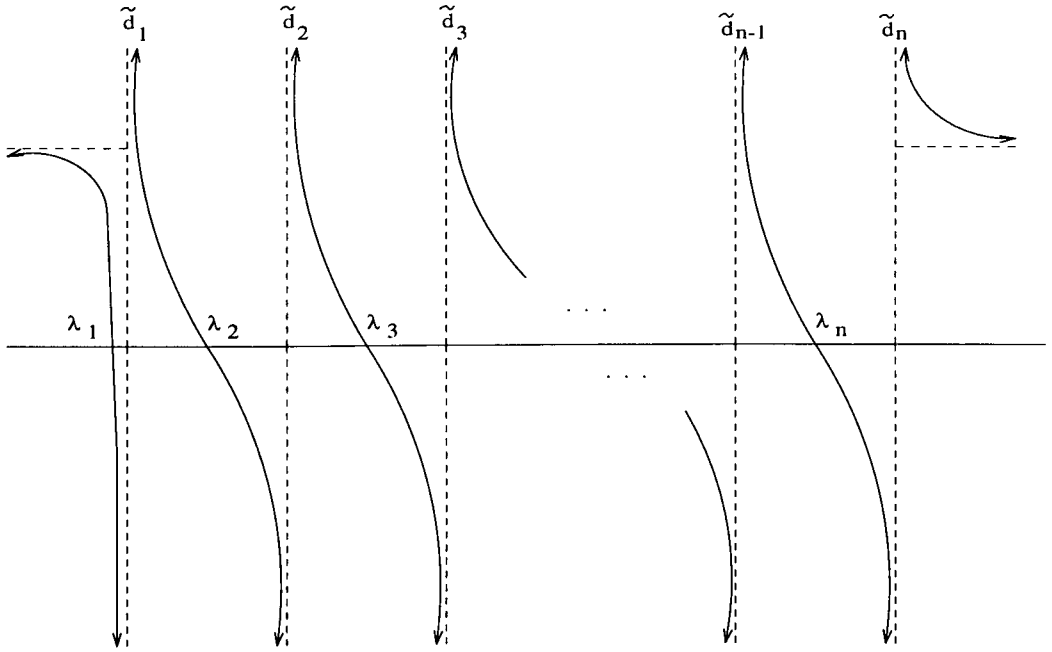


Fig. 7.1 $f(\lambda) = 1 + \rho \sum_{i=1}^{n} \frac{z_i^2}{\tilde{d}_i - \lambda}$ in the case $\rho < 0$.

for $\rho > 0$ the graph is reflected in the y axis.

After calculating the eigenvalues of $\tilde{D} + \rho z z^T$, it is not difficult to compute the eigenvectors. Using equation (7.3.3) for any eigenvalue λ we can take the corresponding eigenvector as

$$q = c(\tilde{D} - \lambda I)^{-1} z$$

where $c = \pm\sqrt{\frac{\rho}{f'(\lambda)}}$ is a nonzero scalar chosen to normalise the vector. Thus for each eigenvalue λ, the corresponding eigenvector has components given by

$$q_i = \frac{c z_i}{\tilde{d}_i - \lambda}, \qquad 1 \leq i \leq n. \tag{7.3.8}$$

The above shows that we can find an orthogonal $\hat{Q}$ and a diagonal D such that $\tilde{D} + \rho zz^T = \hat{Q}D\hat{Q}^T$. It follows that the original matrix T using (7.2.4) can be expressed in the form $T = QDQ^T$, where

$$Q = \tilde{Q}\hat{Q}, \tag{7.3.9}$$

that is we have solved the original problem.

7.4 Exceptional Cases

In section 7.3, eigenvalues and eigenvectors are calculated when the intermediate matrix $\tilde{D}$ has distinct components along the diagonal and the components of z are not zero. If some of the diagonal components of $\tilde{D}$ are equal or some components of z are zero, then we can use a deflation technique. For equal diagonal components this technique involves altering the $\tilde{Q}$ matrices as there is additional freedom which can be used to make some of the z_i zero.

As an example consider the 4×4 symmetric tridiagonal matrix

$$T = \begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix}.$$

Then for $\rho = -2$,

$$\bar{T}_1 = \begin{pmatrix} 2 & -1 \\ -1 & 3 \end{pmatrix} \quad \textit{and} \quad \bar{T}_2 = \begin{pmatrix} 3 & -1 \\ -1 & 2 \end{pmatrix}.$$

These two submatrices have the same eigenvalues 1.382 and 3.618, so the intermediate matrix $\bar{D}$ is as follows:

$$\bar{D} = \begin{pmatrix} \bar{D}_1 & 0 \\ 0 & \bar{D}_2 \end{pmatrix} \equiv \begin{pmatrix} 1.382 & 0 & 0 & 0 \\ 0 & 3.618 & 0 & 0 \\ 0 & 0 & 1.382 & 0 \\ 0 & 0 & 0 & 3.618 \end{pmatrix},$$

and $\bar{d}_1 = \bar{d}_3$, $\bar{d}_2 = \bar{d}_4$. Let us choose a permutation matrix denoted by P as

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

We first permute the elements of $\bar{D}$ and $\bar{Q}$ in such a way that new diagonal elements are in order

$$\tilde{D} = P\bar{D}P = \begin{pmatrix} \tilde{d}_1 & 0 & 0 & 0 \\ 0 & \tilde{d}_2 & 0 & 0 \\ 0 & 0 & \tilde{d}_3 & 0 \\ 0 & 0 & 0 & \tilde{d}_4 \end{pmatrix},$$

where $\tilde{d}_1 = \bar{d}_1, \tilde{d}_2 = \bar{d}_3, \tilde{d}_3 = \bar{d}_2$, and $\tilde{d}_4 = \bar{d}_4$ and to correspond the first row of $\bar{Q}_2$ and the last row of $\bar{Q}_1$ are interchanged so that

$$\bar{T} = \bar{Q}P\tilde{D}P\bar{Q}^T = \tilde{Q}\tilde{D}\tilde{Q}^T$$

where $\tilde{Q} = \bar{Q}P$. Therefore,

$$T = \tilde{Q}(\tilde{D} + \rho zz^T)\tilde{Q}^T$$

where $z = \tilde{Q}^T w$.

Note further that

$$\tilde{D} = H\tilde{D}H^T$$

where H is any orthogonal matrix of the form

$$\begin{pmatrix} \bar{H}_1 & 0 \\ 0 & \bar{H}_2 \end{pmatrix},$$

with $\bar{H}_i$, 2×2 matrices for this examples. Also $\tilde{D} + \rho z z^T$ will have the same eigenvalues as

$$H(\tilde{D} + \rho z z^T)H^T \tag{7.4.1}$$

i.e. $H\tilde{D}H^T + \rho z^* z^{*T}$ when $z^* = Hz$. Using (7.2.4) we can write the original matrix as follows

$$T = \tilde{Q}H^T(\tilde{D} + \rho z^* z^{*T})H\tilde{Q}^T.$$

The orthogonal matrix H may be chosen so that z^* has some zero components, that is, it looks like $z^* = (z_1\ 0\ z_3\ 0)^T$ in the example. Then the problem $\tilde{D} + \rho z^* z^{*T}$ is as follows:

$$\tilde{D} + \rho \begin{pmatrix} z_1^* \\ 0 \\ z_3^* \\ 0 \end{pmatrix} \begin{pmatrix} z_1^* & 0 & z_3^* & 0 \end{pmatrix} = \tilde{D} + \rho \begin{pmatrix} z_1^{*2} & 0 & z_1^* z_3^* & 0 \\ 0 & 0 & 0 & 0 \\ z_3^* z_1^* & 0 & z_3^{*2} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Let H be an orthogonal Householder matrix which transforms the first pair of the components of z with $\bar{H}_1(z_1,\ z_2)^T = (z_1^*, 0)^T$ and the second pair of

the components of z with $\bar{H}_2(z_3,\ z_4)^T = (z_3^*, 0)^T$. This implies that

$$H(z_1, z_2, z_3, z_4)^T = (z_1^*, 0, z_3^*, 0)^T.$$

If we now choose a new $\tilde{Q}$ such that new $\tilde{Q}$ = old $\tilde{Q}H^T$, then

$$T = \tilde{Q}(\tilde{D} + \rho z^* z^{*T})\tilde{Q}^T.$$

In general when we have k of the components of $\tilde{D}$ equal, that is

$$\tilde{d}_{i+1} = \tilde{d}_{i+2} = \ldots = \tilde{d}_{i+k}$$

the basis for the eigenspace of these eigenvalues is not unique and can hence be changed. Let us consider the $k \times k$ submatrix denoted by R

$$R = \begin{pmatrix} \tilde{d}_{i+1} & & & \\ & \tilde{d}_{i+2} & & \\ & & \ddots & \\ & & & \tilde{d}_{i+k} \end{pmatrix} + \rho \begin{pmatrix} z_{i+1} \\ z_{i+2} \\ \vdots \\ z_{i+k} \end{pmatrix} \begin{pmatrix} z_{i+1} & z_{i+2} \cdots & z_{i+k} \end{pmatrix}$$

Let H be an orthogonal Householder matrix which transforms $(z_{i+1}\ z_{i+2}\ \ldots\ z_{i+k})^T$ to $(z_{i+1}^*\ 0\ \ldots\ 0)^T$. Then, since the identity matrix does not change with Householder transformation, i.e. $HIH^T = I$,

$$HRH^T = \begin{pmatrix} \tilde{d}_{i+1} & & & \\ & \tilde{d}_{i+2} & & \\ & & \ddots & \\ & & & \tilde{d}_{i+k} \end{pmatrix} + \rho \begin{pmatrix} z_{i+1}^* \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{pmatrix} z_{i+1}^* & 0 \ldots & 0 \end{pmatrix}.$$

Then

$$H(\tilde{D} + \rho z z^T)H^T = \tilde{D} + \rho z^* z^{*^T}$$

where z^* has zero values in the $i+2, i+3, \ldots, i+k$ positions so that $\tilde{d}_i$ is a eigenvalue of multiplicity $k-1$ for R. Alternatively all elements except z_{i+k} could be made zero.

After permuting the columns of $\bar{Q}$, the diagonal elements of $\tilde{D}$ and introduction of the zero components of z, it follows that corresponding to the zero components of z then D and $\bar{D}$ have common eigenvalues where D are the eigenvalues of T. The original matrix T is expressed as

$$T = \tilde{Q}H^T\hat{Q}D\hat{Q}^T H\tilde{Q}^T = QDQ^T$$

where matrix $\hat{Q}$ comes from (7.3.9). The eigenvalues of the original matrix are the diagonal components of D while the eigenvectors of T are the columns of

$$Q = \tilde{Q}\hat{Q}.$$

In effect this transformation allows the $(i+2)th$ to $(i+k)th$ rows and columns to be ignored. Consequently we may assume that if $\bar{d}_i = \bar{d}_j$ for some i and j then $z_i = 0$ or $z_j = 0$ where $j = i+1$ [17]. Furthermore, equal $\bar{D}$ values along the diagonal components result in a significant reduction in the work required to find the eigenvalues and eigenvectors of $\bar{D} + \rho z z^T$. The above exceptional cases assume that k eigenvalues are exactly equal.

Suppose $z_i = 0$ for some i. Therefore, if $q = (q_1, q_2, \ldots, q_n)^T$ is an

eigenvector of $\tilde{D} + \rho z z^T$ where q is a column of $\hat{Q}$ associated with some eigenvalue $\lambda \neq \tilde{d}_i$, then $q_i = 0$, so column i of Q is equal to column i of $\tilde{Q}$. Otherwise column j of Q is $\tilde{Q}$ times column j of $\hat{Q}$.

With exact arithmetic only $k = 2$ should occur as no more than two eigenvalues of $\tilde{D}$ should be identical. Deflation may also be used when $\tilde{d}_i$ are approximately equal. In this case it is possible that $k > 2$ may occur. It should be noted that in this case the z_i are still set to exactly zero. The deflation technique for approximately equal eigenvalues has been investigated in [26].

7.5 Arithmetic Complexity

We now consider the operations count in the divide-and-conquer algorithm. To determine the arithmetic complexity of the algorithm, we multiply the statement count by the number of times that the statement is executed. Here we examine the total count of multiplications and divisions required to compute the eigenvalues and corresponding eigenvectors of a symmetric tridiagonal matrix when no deflation is used.

We first consider the join of two halves of a matrix of size n. The arithmetic complexity of this algorithm can be formed as follows:

- Cuppen [17] makes the assumption that the root finding part of the algorithm requires on average t function evaluations per zero. In one rank-one modification step of order n this contributes tn^2 operations (see section 7.3.5 of [26]).

- The calculation of z requires n operations (see section 7.2.5). The calculation of n eigenvectors of $\tilde{D}+\rho zz^T$ costs $3n^2$ operations (see section 7.3.8).

- Finally, the matrix multiplication of $\tilde{Q}$ and $\hat{Q}$ require n^3 operations (see section 7.2.9). For the matrix of order n, the total computation for tridiagonal matrix T is

$$\ell(n) = tn^2 + 3n^2 + n^3.$$

For the complete problem recursive and non-recursive versions have the same operation count. The eigenvalue problem T of size $n \times n$ is solved in terms of two independent $\frac{n}{2} \times \frac{n}{2}$ sized sub-problems, which in turn are solved in terms of four independent $\frac{n}{4} \times \frac{n}{4}$ sized sub-problems, and so on. Thus, the total work will be

$$\ell(n) + 2\ell(n/2) + 4\ell(n/4) + \ldots$$

Note that since $\ell(n) \sim n^3$ the total operation count will be

$$\ell_T(n) \sim n^3(1 + 1/4 + 1/16 + \ldots) \approx 1.33n^3.$$

As can seen in the approximate total operation, most of the work takes place at the top level.

It should be noted that a significant portion of the time in the eigenvalue problem algorithm is spent in computing the columns of the eigenvectors

using (7.3.8) and multiplication of the $\tilde{Q}$ and $\hat{Q}$ matrices (7.3.9). If only the eigenvalues are required, then we can do better by saving the n^3 operations required to compute the eigenvectors. Details of this are given in [91].

7.6 Sequential Algorithms

In this section we first present a recursive version of the algorithm that calculates all the eigenvalues and corresponding eigenvectors of a symmetric tridiagonal matrix T. Then non-recursive implementations of this algorithm are considered.

In the recursive version of this algorithm the function to find the eigenvalues and corresponding eigenvectors includes two calls of itself with different parameter values. Suppose we consider a submatrix with its first diagonal element in position index st and last index fin. We can partition this matrix into two submatrices $\bar{T}_1$ and $\bar{T}_2$ where matrix $\bar{T}_1$ has subscript $st : mid$ and $\bar{T}_2$ has subscript $(mid + 1) : fin$, where mid refers to the middle index, that is $mid = \frac{st+fin}{2}$ and $st \leq mid \leq fin$.

For clarification purposes, we can represent the recursive process as a binary tree with each node representing a rank-one tear and hence a partition into two subproblems. It consists of a node called the *root* together with two binary trees called the *left subtree* and the *right subtree* of the root. Thus, for example, in figure 7.2, a matrix of size 61×61 is on the top level and the level number is zero and submatrices of size 31×31 and 30×30 are on the next to top level with level number one. The height of a tree is

the maximum level among all nodes in the tree. The tree in figure 7.2 is of height 5 (including submatrices of size 1×1 and 2×2). Pictorially, we use a tree illustrating the partitioning of the original matrix into a number of submatrices using a number of levels. Thus, there are two symmetric tridiagonal submatrices for which we need to find eigenvalues at each node of the tree. For example consider a matrix size $n = 61$. The numbers in the diagram indicate the sizes of the matrices. Subdivisions are represented by the tree in figure 7.2.

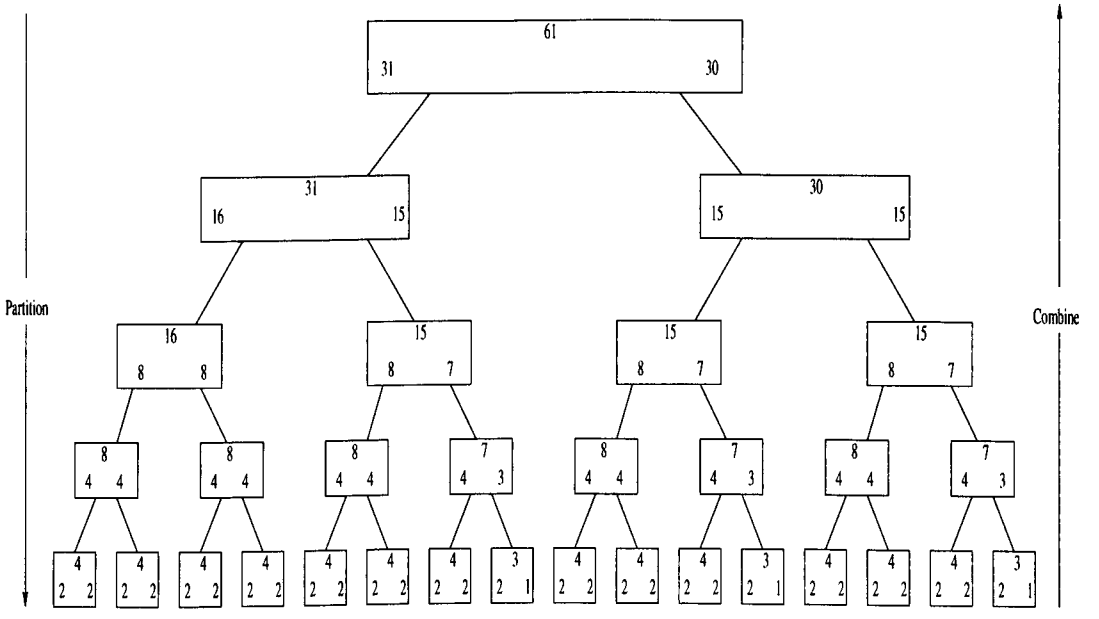


Fig. 7.2 Computation Tree.

This tree illustrates that if the matrix size is an odd number, then the partitioning scheme will not produce equal sized symmetric tridiagonal submatrices at the next level. It also shows that at the lowest level (i.e. in the fifth level) there are 32 submatrices and some submatrices are 1×1 and

some 2×2 which are both trivial eigenvalue problems. In the case of a 1×1 submatrix, the intermediate eigenvalue is the element of that submatrix and for a 2×2 submatrix the eigenvalues are the solution of a quadratic equation.

Now let us begin by considering the tree of recursive calls and determine the order in which the combinations are actually done. As mentioned before, there are two recursive calls which are required at each subtree of the tree in figure 7.2 to solve eigenvalue problems. Each of these subproblems may be solved independently without fear of data conflict. The combination of subproblems is started at the highest level (i.e. smaller submatrices). As soon as the first two sub-matrices which are represented as the leftmost two nodes in the tree are solved, they are then combined. Then the next two sub-matrices are combined, and afterward the resulting sub-matrices of these are combined and the node will become a node at the previous level. In this way the process continues building up larger sizes of sub-matrix.

We also considered two non-recursive versions. The important distinction between them is that the partitioning in the two implementations is different. The first version uses a different partitioning to the recursive version as it seemed simpler to implement. The matrix is partitioned into 2×2 submatrices from the left. If the original matrix size is even then the submatrices will all be of size 2×2. If its size is odd then the submatrices will also be of size 2×2 except for the last one which will be 1×1. The second implementation uses the same partitions as the recursive method but uses a linked list to store the submatrices. The motivation for a non-recursive

version was to compare this with the recursive version and to investigate the relationship between the recursive and non-recursive implementations of Cuppen's divide-and-conquer method.

The first non-recursive version uses the simple partitioning scheme carried out at the lowest level. This algorithm is carried out in stages corresponding to each level. The first part of the algorithm carried out partitioning and modification of the matrix followed by solution of the quadratic equations. The second part of the algorithm combines the sub-solutions for pairs of submatrices in each successive level. The algorithm is as follows:

i, j, k are subscripts
nb is final index of the matrix
α a vector - matrix diagonal
β a vector off-diagonal elements of the matrix

$j = 0;$
(this is needed for $nb \leq$ 2 case)
for $(i = 2;\ i <\ nb;\ i = i + 2)$
{
  (modify the matrix)
  $\alpha(i) = \alpha(i) - \beta(i);$
  $\alpha(i+1) = \alpha(i+1) - \beta(i);$

  (solve quadratic equation)
  $quadr(i-1, i, \alpha, \beta, ld, Q);$
  $j = i;$
}
(solve quadratic equation or $1 \times$ 1 matrix)
$quadr(j+1, nb, \alpha, \beta, ld, Q);$

(submatrices join process)
$k = 2;$ (k represents submatrix size at current level)
do
{

```
    for (i = 1; i <= nb − k; i = i + 2 ∗ k)
    {
      ma = i; mc = i + k − 1; mb = i + 2 ∗ k − 1;
      if (mb > nb) mb = nb
      ρ = 2 ∗ β(mc);
      calculate w and z = Q̃^T w using (7.2.2) and (7.2.4)
      use deflation if applicable
      find the new eigenvalues
      calculate q_i using (7.3.8)
      multiply Q = Q̃Q̂ using (7.3.9)
    }
    k = 2 ∗ k;
}
while (k <= nb + 1);
```

It is important to note that the actual sizes of the submatrices used by this "simple non-recursive" version are not the same as those used by the recursive implementations. In order to illustrate the workload (i.e. the actual size of the original matrix and submatrices) of the recursive (ℓ_{rec}) and simple non-recursive (ℓ_{nrec}) versions we need to find out the relative costs of matrix multiplication for the sizes of the matrices which are used in this work. For simplicity, at the top and next to top level the work is

$$\ell(n) = n^3 + n_1^3 + n_2^3 \tag{7.6.1}$$

where n is the original matrix size and n_1 and n_2 are the sizes of the sub-matrices. Note that for the calculation of the relative costs of matrix multiplication at the top (e.g. matrices of size $n \times n$) and next to top (e.g. matrices of size $n/2 \times n/2$ and $n \times n$) levels the formula is:

$$\frac{\ell_{nrec}(n)}{\ell_{rec}(n)} = \frac{n^3 + n_1^3 + n_2^3}{1.25n^3} \tag{7.6.2}.$$

We will particularly consider the relative cost of the matrix sizes 100×100, 200×200, 300×300, and 400×400 for the matrix multiplication.

For $n = 100, 200$, and 400, the ratio is approximately 1.047. Similarly, for $n = 300$, the ratio is approximately 1.300.

For a further example we shall consider an extreme case. Suppose that we have the original matrix of size 65×65. In the simple non-recursive version the last combination is a 64×64 with a 1×1, while in the recursive implementation this would be a combination of a size 33×33 with 32×32 at the top level. For $n = 65$ the ratio is approximately 1.563. So we can conclude from the above calculation of the operation costs that the simple non-recursive version involves much more work than the corresponding recursive version. The former version also requires an extra level for some matrix sizes.

It is clear that the relative cost here depends on the size of the matrix i.e. how evenly or nearly evenly it is partitioned in the algorithm. In general, a lower bound for the relative cost of the work is 1 for matrices with size a power of 2 and an upper bound is 1.8 for matrices of other sizes.

The second non-recursive version used the same partition as the recursive version and used a linked list structure to accomplish this. This version can be run in virtually any programming environment but the recursive implementation obviously requires one that supports recursion. Some programming languages, such as FORTRAN and BASIC, do not support recursion [59]. Furthermore, the non-recursive algorithm is also useful even when the programming language used supports recursion, as loops are usually more efficient than recursive calls.

The first step carried out in this version of the algorithm is the partitioning scheme and modification of the original matrix. The partitioning is carried out till the submatrices are of dimension 1×1 or 2×2.

Conventional *linked lists* are described, in [80]. Each node in a linked list contains both the list user's data and an explicit link to the next node using a *pointer*. We can depict a list not as a sequence of contiguous entries but as a sequence of entries in an order that is determined solely by pointers leading from one entry to another. Going through this sequence is called *traversing a list*. To traverse a list, we must start at the list head and follow the list pointers. Conventionally, the last node of the list is marked by a *Null* pointer. An empty linked list is likewise represented by a *Null* pointer. Figure 7.3 illustrates these conventions. Using a linked list structure in the algorithm allows one to avoid data movement that would otherwise be necessary to insert or delete an item from a list.

To illustrate how basic linked list primitives might be implemented in C++, we begin by precisely specifying the format of the list nodes. The following declaration illustrates the linked list in C++

```
typedef struct nodepars * PtrNode;
struct nodepars
{
    int st;
    int fin;
    int lev;
    PtrNode link;
},
```

The C++ name for a record is *struct* and the structure type allows the programmer to assemble several items of data in a single structure, which is very similar to a Pascal record. Each node in the linked list used here is a record containing the position of the first and last diagonal elements of the submatrices and level of the subdivision and a link to another node. The *st* is the position of the first element and *fin* is the last diagonal element of

the submatrix. For example, from figure 7.3, $st = 1$ and $fin = 2$ in the first node and $st = 3$ and $fin = 3$ in the second node and so on. *lev* indicates the level of the node in the binary tree, and *link* is a pointer to point to the next node. That is for the original matrix $lev = 0$ and every time a subdivision is made *lev* is incremented by one.

Figure 7.3 illustrates how our data can be represented as a linked list. The list consists of a sequence of nodes linked by pointers. Each list contains an item of data and a link to the next node. As an example in figure 7.3 the representations of the list is given for a 9×9 matrix after all subdivisions have been carried out.

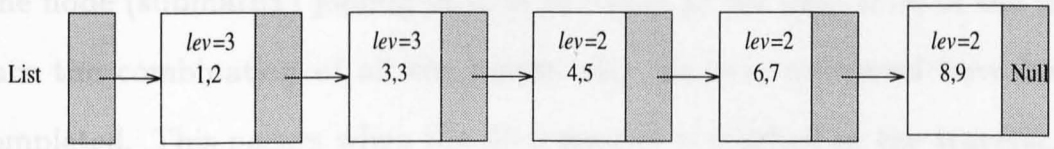


Fig. 7.3 Linked list with pointers indicating order of items.

The submatrices are stored as a list in both the partitioning and the modification stage. In the linked list implementation space is allocated dynamically. This invokes a call to the constructor for the type. The nodes are actually created only when the function *new* is called. The following line of code is used for such a construction:

$$ptrb = new \ \ nodepars(mid + 1, fin, lev + 1, ptr- > link)$$

where *lev* is level number of the partition, and $ptr- > link$ is a pointer which points to the next node. (The $(- >)$ notation is used in C++ to provide access to the members of a structure via a pointer.)

The submatrices are joined in the second part of the algorithm. In order to combine submatrices, two integers are kept for checking the level number; these are *lev* which indicates the level of the submatrix for each node in the tree and *mlv* which indicates the maximum level of the nodes in the whole list. The list is searched starting from its head for the first pair of adjacent nodes with the same level number as the maximum level number ($lev == mlv$). The matrices corresponding to the two nodes are then joined, then the second of the nodes is deleted from the list and the first of the nodes becomes a node with *lev* decremented by 1. When deleting a node from the list it is returned to free store using the *delete* operator. The node (submatrix) joining process proceeds to the next level of the tree once the combination of all the nodes with $lev == mlv$ level have been completed. This occurs when the *Null* pointer is reached in the traverse of the list, at which point the *mlv* (maximum level) number is decremented. Note that the nodes with equal maximum level number always occur in pairs, as a consequence of the way the tree is constructed.

7.7 Parallel Implementations

We consider parallel implementations of the Symmetric Tridiagonal Eigenproblem, discussed in the previous section. The algorithms have some natural parallelism which can be made use of in both the recursive and the non-recursive versions. In section 7.6, the discussions help to see and understand how to implement parallelisation of these algorithms.

7.7.1 Recursive Implementations

We consider four variant recursive parallel implementations. As mentioned in section 7.5, the function makes two calls to itself and both parts of the calculation are independent so that these calls can be done in parallel without fear of data conflicts.

In the first implementation these calls are made using two "THREADcreate"s and are terminated with two "THREADjoin"s, which act as a barrier.

The second recursive parallel implementation was designed to reduce the number of new threads created from that in the first implementation. The first call is done by *THREADcreate* and the second is a conventional call.

The third recursive implementation attempts to avoid creating more threads than processors. This can be done by keeping a count of the number of *THREADcreate*'s. The count indicates the number of threads that have been created. If the count is less than the number of processors then this count is incremented and a "THREADcreate" call is made; otherwise a conventional recursive call is made. The termination is done with one "THREADjoin" for each *THREADcreate*. The thread count is immediately decremented after the "THREADjoin". As this count is used by different processes, a *lock* is used when both incrementing and decrementing the count.

In the fourth implementation we considered another approach to avoid

creating more threads than processors. The process is carried out by using a counter to count the level of recursion and this level is multiplied by 2 after each recursion. The fourth implementation only creates new threads at the top levels, and because it only checks the level it does not need to use a *lock* so that it reduces the amount of interprocessor communication.

7.7.2 Non-recursive Implementations

We consider parallel implementations of the two non-recursive versions outlined in section 7.5. The first one uses the simple partitioning scheme carried out from right to left at the lowest level. This version is carried out in stages corresponding to the level. In the first part the algorithm carried out partitioning and modification of the original matrix sequentially. Solution of the quadratic equations is then carried out in parallel. When the first part is completed the second part of the algorithm combines pairs of submatrices, and this is carried out in parallel.

The second implementation uses the same partitions as the recursive method but a linked list is used to store information about the submatrices. In the first part the modifications to the original matrix are carried out sequentially but all the quadratic equation calculations are carried out in parallel. When the first part is completed, then the submatrix joining is carried out in parallel in the next part.

The parallelisation of the first implementation is straightforward. It simply involves parallelising a loop using "THREADcreate" and

"THREADjoin". In the first part of the parallel implementation (quadratic equations) threads are allocated to tasks in a predetermined (scattered) ordering. We define task i to be the solution of the quadratic equations formed from rows and columns i and $i+1$ of the matrix. The *thread* chooses pairs of 2×2 matrices and solves the corresponding quadratic equations until there are no more pairs of matrices to be allocated.

The submatrices are combined in the second part of the algorithm. The combination of the pairs is carried out from left to right until the end of the list. In the second part of the parallel implementation threads are also allocated to tasks in a predetermined (scattered) ordering. We also define task i at the level k where the submatrix size is k to be the combination of the sub-solutions formed from rows and columns with subscripts $(\ i,\ i+k-1\ ),(\ i+k,\ i+2k-1\ )$. When $k=2$ then the sequence of pairs will have indices $(\ i,\ i+1\ )$, $(\ i+2,\ i+3\ )$ and so on. In the parallelisation of the combination process, thread j carries out the following tasks $for\ (i = na + (2 * k * (j-1));\ i <= nb - k;\ i+ = 2 * prcs * k)$, where k is submatrix size $(k = 2, 4, 8, \ldots)$, na is the first index of the matrix, $prcs$ is number of processors, and j is the thread number.

The process is carried out with the step size (i.e. $2 * k * prcs$) increasing in value by a factor of 2 for each consecutive level. This allows the pre-allocation of tasks to threads which should reduce the amount of inter-thread communication as no lock is needed for the task allocation in this non-recursive version. This organisation is simple to implement in parallel. This

algorithm, in addition to the problem discused for the sequential version, however, has a problem with lack of balance when the upper levels of the tree are reached (if n is not a power of 2).

In order to illustrate the allocation of the task to threads (i.e. the actual size of the submatrices) of the recursive (ℓ_{recp}) and simple non-recursive (ℓ_{nrecp}) versions, we can calculate the relative costs of matrix multiplication at the top two levels using 2 processors with a modification of (7.6.1), that is

$$\frac{\ell_{nrecp}(n)}{\ell_{recp}(n)} = \frac{n^3 + n_1^3}{1.125n^3} \qquad (7.7.2.1).$$

where n_1 is the larger of the two submatrix sizes.

We need to find out the relative costs of matrix multiplication for the matrix sizes tested in this work using the formulae (7.7.2.1). For $n = 100, 200$, and 400 size of matrices the ratio is approximately 1.122. Similarly, for $n = 300$ size of matrices the ratio is approximately 1.441. The parallel relative costs is larger than the sequential one, because the two levels can be done simultaneously by 2 processors.

When we consider the matrix multiplication ratio for 2 processors for the extreme case with $n = 65$, the ratio of $\ell_{nrecp}(n)$ and $\ell_{recp}(n)$ versions is approximately 1.728.

The second version uses the same partitioning scheme as in the recursive implementations, and parallelises the sequential linked list version described in section 7.6. The parallel version of the linked list implementation carries out the partitioning and modification of the original matrix sequentially.

Solving the quadratic equations in the first part and combining the submatrices in the second part of the algorithm are done in parallel.

The first part of the parallel implementation finds the solution of the quadratic equations. The allocation of tasks (i.e. quadratic equations) to threads is done in a pre-determined way. The tasks are allocated to threads by searching from the head of the list for the first pair of nodes. This process continues until the head of the list reaches a $Null$ pointer.

The second part of the implementation carries out the combination of the sub-matrices. In addition to keeping the integers (lev and mlv) as in the sequential linked list version, a flag atv (active) is kept in the parallel implementation, its purpose will be explained later. It is important to clarify that the level (lev) and the active flag (atv) are associated with each node (submatrix) while the maximum level (mlv) is associated with the whole list, i.e. there is an atv flag in each node. The submatrices are allocated to threads by each thread searching from the head of the list for the first pair of nodes with the same level number as the maximum level number ($lev == mlv$). The submatrices are combined, as in the sequential version. Then the second of the two nodes is deleted from the list and the first of the nodes will become a node at the previous level of the list. When the current pointer points to $Null$, then mlv is decremented and the current pointer is reset to point to the head and a new level is started. Whenever a submatrix join process is carried out the lev number is decremented but mlv is only updated when all submatrices on that level have been allocated but not

necessarily completed. Updating of the level number (lev) and maximum level number (mlv) was done by only one thread at a time. We achieve this using a *lock*. The list updates must also be done by only one thread at a time, and again this is accomplished using the *lock*. The flag atv (active) is kept in this parallel version and is initially set to $FALSE$ for each node. It is set to $TRUE$ whenever a node is being processed. This is needed to avoid the possibility of a $THREAD$ attempting to start a new level from using a node which is not yet ready. If a thread finds a node with the flag $atv==TRUE$ it terminates. Because this phenomenon is only likely to occur at the lower levels (i.e. large submatrix sizes) and at those levels where there are likely to be plenty of threads it seems sensible to terminate the threads rather than to look for further work. The submatrix join process continues until the mlv value (maximum level number) reaches zero and ($head->link == Null$).

Both the quadratic solution and submatrix combination part is terminated with "THREADjoin"s, which act as barriers. Based on this discussion, we depict below the parallel non-recursive version using a linked list in the following code. This is used by each thread for the submatrix combination process. In this version $pars->$ indicates the shared variables in the code below.

```
int mlv, lev;
PtrNode  ptr, ptra;
PtrNode  head = pars-> head;
char  stop = FALSE;
ptr-> atv = FALSE;
```

```
mlv = pars- > mlv;
while ((pars- > head- > link! = NULL)&&(!stop))
{
    pars- > lck.lock();
    ptr = pars- > ptr;
    lev = ptr- > lev;
   if  (ptr! = NULL)
   {
      stop = ptr- > atv;
      if  (!stop)
      {
         if (lev == mlv)
         {
            ptra = ptr- > link;
            if (ptra ! =  NULL)
            {
               if  (ptra- >atv)
               {
                     stop=TRUE;
                     pars- >lck.unlock();
               }
               else
               {
                     ptr- >atv=TRUE;
                     pars- >ptr=ptra- >link;//for other processors
                     pars- >lck.unlock();
                     (start to update the submatrices join process)
                     join(ptr,pars);
                     pars- >lck.lock();
                     ptr- >fin=ptra- >fin;
                     ptr- >link=ptra- >link;
                     ptr- >lev=ptr- >lev-1;
                     ptr- >atv=FALSE;
                     pars- >lck.unlock();
                     delete ptra;
               }
            }
         }
         else
         {
               pars- >ptr=ptr- >link;
```

```
                    pars- >lck.unlock();
                }
            }
            else                                        // stop TRUE
              pars- >lck.unlock();
        }
        else
        {
            pars- >ptr=head;
            pars- >mlv=mlv-1;
            pars- >lck.unlock();
        }
    }
```

7.7.3 Additional Parallelisation in Matrix Multiplication Part $\tilde{Q}\hat{Q}$

In addition to the algorithms in sections 7.7.1 and 7.7.2 concerning parallelisation of the recursive and non-recursive implementations, we also parallelised the matrix multiplication part of the algorithms using (7.3.9) in all the above parallel implementations. As observed in section 7.5, a significant portion of the time in the eigenvalue problem algorithm is spent in computing the product of the $\tilde{Q}$ and $\hat{Q}$ matrices. The products are carried out by a nest of loops, the first loop (subscript) is for the row of $\tilde{Q}$ and the second loop (subscript) is for the column of $\hat{Q}$. The Procedure Matrix Multiplication is the same for both the recursive and non-recursive versions. It is also important to point out that allowance for the effect of deflation is included. The parallel matrix multiplication part of the algorithm is given below.

Procedure MATRIX MULTIPLICATION
na is first index of the submatrix
nb is last index of the submatrix

```
prcount is the thread number
prcs is number of processor (threads)

if (lev ≤ prcs)
{
  START PARALLEL SECTION
  for (i = na + prcount − 1; i <= nb; i = i + prcs)
  {
     for (j = na; j <= nb; j + +)
     {
        if (z(j) == 0)
          Q(i, j)=Q̃(i, j)
        else
        {
           for (k = na; k <= nb; k + +)
           Q(i, j) = Q(i, j) + Q̃(i, k) ∗ Q̂(k, j)
        } //endelse
     } //endj
  } //endi
  FINISH PARALLEL SECTION
} //endif

else
{
    (matrix multiplication is done sequentially)
} //endelse
```

For the matrix multiplication the allocation of the work to threads has been done in a pre-determined way. We consider here an approach similar to the fourth recursive parallel implementation in order to avoid creating more threads than processors at the lowest level. This process is carried out by using a counter for the level of the sub-matrices and this level is multiplied by 2 after the completion of each level. This enables parallelisation of matrix multiplication to be only used at the top levels. The parallel structure is controlled in the following manner: if the level number (lev) is less than the number of processors then the matrix multiplication is done in parallel, otherwise it is done sequentially.

7.8 Experimental Results

In this section we present and analyse the effect of deflation on the accuracy of the sequential recursive and non-recursive versions in sub-section 7.8.1. A comparison of the results of the recursive and non-recursive parallel implementations outlined in the previous sections is given in sub-section 7.8.2. An extra comparison is also presented for the results of the additional parallelisation in the matrix multiplication part in sub-section 7.8.3.

7.8.1 Comparison of the Effect of Deflation on Accuracy

This section presents an experimental evaluation of the effect of deflation on accuracy of the eigenvalues (i.e. the accuracy of the eigenvectors are not measured). A number of test matrices were used to test Cuppen's algorithm since the amount of deflation in this method depends on the test matrix type and this affects the timings.

Tables 7.1 and 7.2 show the raw times, deflation counts and the accuracy for several orders of matrix types $[-1, 2, -1]$ and $[-1, u, -1]$ for solving eigenvalue problems. These test matrices were introduced in chapter 2. For the matrix $[-1, u, -1]$ exact eigenvalues are unknown and hence the accuracy of this matrix is not shown in the tables. We also use two tolerances: the first tolerance (ε_r) is used in the root finding stage and the second tolerance (ε_d) is used to test for deflation. The first test matrix type $[-1, 2, -1]$ has significant deflation in the root finding step. The second test matrix type [-1,u,-1] has the value $u = i \times 10^{-6}$ in the i^{th} diagonal position, and for matrices of this type the intermediate matrix $\tilde{D}$ has distinct components along its diagonal at each stage and the components of z are ratios of the diagonal components to the off-diagonal components which are small.

For the sizes tested little deflation (using $\varepsilon_d \geq 1e-6$) or no deflation (using $\varepsilon_d \leq 1e-8$) occurs with this type of matrix. The times, the deflation counts and accuracy tests are given in the tables below. The results given in tables 7.1a, 7.1b, and 7.1c are obtained from the recursive version, non-recursive (using linked list) version and simple non-recursive version respectively. The recursive and non-recursive (using linked list) versions give the same amount of deflation and same accuracy.

The amount of deflation and hence the timing also depends on the size of the tolerance ε_d which is used to test whether two eigenvalues are nearly equal. When the difference between the two eigenvalues is less than ε_d we consider the eigenvalues to be equal and then apply deflation otherwise we will consider the eigenvalues not to be equal. The tolerance ε_d is also used in the check for the value of z_i i.e. $|z_i| < \varepsilon_d$ rather that $|z_i| = 0$. An important effect on the accuracy of the result is the size of the tolerance ε_r which is used to terminate the root finding loop. When the differences of the absolute value of the (second and first) roots is less than or equal to ε_r then the loop terminates. These tolerances have an effect on the amount of work required.

The accuracy of all the implementations is also tested. We test matrices chosen from matrix type [-1,2 ,-1] using different tolerances. This matrix type has known eigenvalues, so that the error can be directly evaluated. Let $\bar{\lambda}_i$ be the approximation of the exact eigenvalue λ_i. Tables 7.1a, 7.1b, 7.1c, 7.2.a 7.2b and 7.2c give the times as well as the error

$$max_i\{|\bar{\lambda}_i - \lambda_i|\}$$

of eigenvalues computed by the recursive, non-recursive (linked list), and simple non-recursive versions for the test problems.

From the point of view of numerical computation, in most cases the error

seems to be roughly $max\ \{\varepsilon_r,\ \varepsilon_d\}$ and no advantage seems to be gained if one epsilon is much larger than the other though choosing ε_d slightly larger than ε_r seems to be helpful. We make several observations about the experimental results. The accuracy is affected by the relationship of the different tolerances (ε_d or ε_r). Firstly, we observe that when testing the problems with $\varepsilon_d < \varepsilon_r$, the accuracy of all the implementations are very poor and hence the accuracy of the implementations were not tested further, and the results are not shown in the tables. The second observation is that when testing the problems with $\varepsilon_d > \varepsilon_r$ for all matrix sizes, the implementations achieve better accuracy than when testing the problems with $\varepsilon_d = \varepsilon_r$ for given ε_d.

Finally, when testing the problems with $\varepsilon_d \approx \varepsilon_r$, in most cases the implementations achieve good accuracy except the recursive and non-recursive (linked list) implementations which give very poor accuracy with $\varepsilon_d = \varepsilon_r = 1e-8$ for a 300×300 matrix.

The experimental results given in this section show that the accuracy of the computed eigenproblem depends on the tolerances (ε_d and ε_r) used. It appears that the simple non-recursive version achieves marginally better accuracy than recursive and non-recursive (liked list) versions with matrices of size 100(100)400. The total amount of deflation in the simple non-recursive version is much more than that in the recursive and non-recursive (linked list) versions. This is because the matrix partitioning into 2×2 submatrices is from right to left, so that there are many identical submatrices at the lowest level.

$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-9$						$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-8$				
[-1,2 ,-1]				$[-1,u,-1]$		[-1,2 ,-1]			$[-1,u,-1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
100	145	33.00	5.87e-09	-	43.95	127	32.50	1.86e-07	-	51.31
200	353	213.82	5.47e-09	-	332.61	330	236.48	2.62e-07	-	328.72
300	449	763.38	3.11e-08	-	1180.48	298	1125.21	1.49e-02	-	1153.24
400	869	1755.78	2.17e-08	-	2730.43	828	1950.35	8.87e-07	-	2765.73

$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1,u,-1]$		[-1,2 ,-1]			$[-1,u,-1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
100	145	30.16	1.86e-07	92	49.15	128	30.70	9.45e-06	93	40.49
200	353	211.49	2.75e-07	197	330.84	325	230.59	1.83e-05	197	332.48
300	449	722.19	3.08e-07	230	1168.82	379	905.28	5.22e-05	228	1175.63
400	869	1776.43	8.87e-07	405	2694.18	825	1957.46	6.88e-05	405	2656.91

$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1,u,-1]$		[-1,2 ,-1]			$[-1,u,-1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
100	145	27.71	1.86e-07	145	27.39	145	26.87	9.45e-06	145	31.49
200	355	202.65	5.24e-05	355	206.31	355	202.15	5.60e-05	355	202.84
300	449	746.49	7.53e-04	449	793.95	449	745.52	7.51e-04	449	739.54
400	877	1720.35	6.12e-04	876	1707.76	875	1721.71	6.33e-04	876	1717.21

Table 7.1a: A Comparison of the Deflation Matrices Size of 100(100)400 Times for Recursive Version.

$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-9$						$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-8$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
100	145	29.60	5.87e-09	-	46.03	127	35.81	1.86e-07	-	50.56
200	353	213.20	5.47e-09	-	358.13	330	233.83	2.62e-07	-	352.48
300	449	801.27	3.11e-08	-	1261.33	298	1185.57	1.49e-02	-	1216.48
400	869	1718.11	2.17e-08	-	2901.08	828	1928.01	8.87e-07	-	2822.26
$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
100	145	29.80	1.86e-07	92	44.48	128	38.57	9.45e-06	93	47.17
200	353	207.75	2.75e-07	197	347.07	325	253.17	1.83e-05	197	337.16
300	449	750.41	3.08e-07	230	1216.55	379	984.65	5.22e-05	228	1189.46
400	869	1799.64	8.87e-07	405	2855.54	825	1909.82	6.88e-05	405	2800.92
$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
100	145	29.73	1.86e-07	145	29.46	145	27.79	9.45e-06	145	28.15
200	355	208.80	5.24e-05	355	226.78	355	213.78	5.60e-05	355	214.04
300	449	753.29	7.53e-04	449	794.19	449	746.21	7.51e-04	449	758.41
400	877	1752.58	6.12e-04	876	1762.21	875	1743.79	6.33e-04	876	1699.27

Table 7.1b: A Comparison of the Deflation Matrices Size of 100(100)400 Times for Non-Recursive Version (Linked list).

$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-9$						$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-8$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
100	162	46.14	3.87e-09	-	51.06	162	46.18	8.08e-08	-	51.27
200	420	351.03	6.62e-09	-	359.08	420	341.86	3.07e-07	-	378.79
300	714	1567.29	6.30e-08	-	1587.63	714	1564.17	2.51e-07	-	1534.43
400	1036	2946.54	2.13e-08	-	2938.49	1036	2854.02	1.69e-07	-	2937.87
$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
100	162	44.59	8.08e-08	130	42.98	162	49.35	3.99e-06	130	40.83
200	420	340.18	3.07e-07	277	325.96	420	350.20	1.27e-05	277	320.18
300	714	1519.49	2.51e-07	426	1491.53	714	1512.00	6.92e-05	426	1452.25
400	1036	2853.41	1.69e-07	572	2949.06	1036	2866.08	2.77e-05	572	2705.67
$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
100	162	44.49	8.08e-08	162	44.36	162	45.59	3.99e-06	162	44.73
200	421	348.14	5.04e-04	421	320.10	421	335.49	4.01e-04	421	346.28
300	719	1512.33	4.21e-04	719	1493.55	719	1538.49	4.21e-04	719	1468.31
400	1044	2821.02	3.47e-04	1046	2835.49	1044	2809.16	3.48e-04	1046	2870.54

Table 7.1c: A Comparison of the Deflation Matrices Size of 100(100)400 Times for Simple Non-Recursive Version.

The test results also illustrate the times measured for matrix types [-1,2,-1] and [-1,u,-1] with different amounts of deflation. For all problem sizes tested when using $\varepsilon_d = 1e-8$ and $\varepsilon_d = \varepsilon_r$ execution times are larger than those using $\varepsilon_d > 1e-8$ and $\varepsilon_d > \varepsilon_r$ for the matrix type [-1,2,-1] as there is less deflation and so more arithmetic is needed. Note that even though the non-recursive (linked list) version gives identical deflations to the recursive version, in most cases the raw times are not quite the same. In most cases, the raw times for the recursive version are relatively better than the non-recursive (linked list) version. Moreover, even though the simple non-recursive version gives more deflation than the recursive version and the non-recursive (linked list) version (i.e. all cases except using $\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-9$ with the matrix type [-1,u,-1]), the times are still significantly worse.

The theory given by Dongarra and Sorensen in [26] suggests that it would be appropriate to relate the two tolerances by

$$\rho\varepsilon_r \cong \varepsilon_d$$

where ρ is the scalar 2 times the off-diagonal element of the original matrix. This is consistent with the results obtained here.

The results for the simple non-recursive version shows substantial improvements when the implementations are tested for power of 2 size of matrices, when all the implementations have the same submatrices and the same deflation. Comparing the raw times for all versions indicates that the simple non-recursive version executes most efficiently. Overall the simple non-recursive version is relatively better than the recursive and the non-recursive (linked list) versions in terms of raw time,using all values of ε_d and ε_r for both test matrix types [-1,2,-1] and [-1,u,-1].

$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-9$						$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-8$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
64	100	7.94	4.92e-09	-	12.54	100	7.82	1.20e-07	-	14.88
128	260	56.11	5.38e-09	-	96.63	260	55.15	1.91e-07	-	92.95
256	644	455.70	1.46e-08	-	753.06	644	451.46	5.30e-07	-	741.35
$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
64	100	7.78	1.20e-07	68	12.30	100	7.27	5.36e-06	68	10.82
128	260	54.83	1.91e-07	164	85.39	260	53.36	7.53e-06	164	87.50
256	644	457.17	5.30e-07	357	721.62	644	472.33	2.75e-05	356	719.49
$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
64	100	8.73	1.20e-07	100	7.71	100	8.68	5.36e-06	100	7.39
128	260	55.05	1.91e-07	260	55.47	260	56.02	7.53e-06	260	53.18
256	646	510.86	7.36e-04	646	457.59	646	465.00	7.37e-04	646	442.41

Table 7.2a: A Comparison of the Deflation Matrices Size of 2^n Times for Recursive Version.

$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-9$						$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-8$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
64	100	8.32	4.92e-09	-	13.08	100	8.40	1.20e-07	-	14.67
128	260	56.80	5.38e-09	-	93.11	260	61.90	1.91e-07	-	91.15
256	644	456.99	1.46e-08	-	742.76	644	452.11	5.30e-07	-	741.10
$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
64	100	8.15	1.20e-07	68	12.15	100	7.71	5.36e-06	68	11.36
128	260	56.83	1.91e-07	164	89.30	260	54.23	7.53e-06	164	85.17
256	644	449.91	5.30e-07	357	724.71	644	445.87	2.75e-05	356	713.92
$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
64	100	9.70	1.20e-07	100	8.04	100	7.68	5.36e-06	100	7.70
128	260	58.87	1.91e-07	260	56.47	260	54.03	7.53e-06	260	54.14
256	646	447.93	7.36e-04	646	453.02	646	464.66	7.37e-04	646	445.36

Table 7.2b: A Comparison of the Deflation Matrices Size of 2^n Times for Non-Recursive Version (Linked list).

$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-9$						$\varepsilon_d = 1e-8$ and $\varepsilon_r = 1e-8$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
64	100	9.36	4.92e-09	-	12.69	100	7.71	1.20e-07	-	12.18
128	260	57.30	5.38e-09	-	97.09	260	54.93	1.91e-07	-	88.45
256	644	449.82	1.46e-08	-	741.03	644	448.26	5.30e-07	-	742.85
$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
64	100	7.76	1.20e-07	68	13.86	100	7.21	5.36e-06	68	10.71
128	260	54.36	1.91e-07	164	89.17	260	52.72	7.53e-06	164	82.22
256	644	453.75	5.30e-07	357	726.11	644	437.26	2.75e-05	356	696.63
$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-8$						$\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-6$				
[-1,2 ,-1]				$[-1, u, -1]$		[-1,2 ,-1]			$[-1, u, -1]$	
n	count	secs	accur	count	secs	count	secs	accur	count	secs
64	100	8.73	1.20e-07	100	7.69	100	7.21	5.36e-06	100	7.20
128	260	55.72	1.91e-07	260	54.66	260	52.04	7.53e-06	260	52.00
256	646	450.78	7.36e-04	646	459.55	646	462.62	7.37e-04	646	442.48

Table 7.2c: A Comparison of the Deflation Matrices Size of 2^n Times for Simple Non-Recursive Version.

Comparing power of 2 size of matrices in tables 7.2, for recursive and non-recursive (linked list) versions using $\varepsilon_d > \varepsilon_r \geq 1e{-}9$ for test matrix type [-1,2,-1], we observe that the recursive version is marginally better than the non-recursive (linked list) version in terms of raw time for small matrix sizes. For larger matrix sizes the reverse is true. The accuracies are, however, the same in both cases. On the other hand, for test matrix type [-1,u,-1] and using $\varepsilon_d > \varepsilon_r \geq 1e-9$, the non-recursive (linked list) version is marginally better than the recursive version for larger matrix sizes except when using ($\varepsilon_d = 1e-6$ and $\varepsilon_r = 1e-8$) and ($\varepsilon_d = 1e-4$ and $\varepsilon_r = 1e-6$), but for smaller matrix sizes the reverse is true. The other observation is that for all matrix sizes the recursive version is relatively better than the non-recursive (linked list) version when using $\varepsilon_d = \varepsilon_r$ for the matrix type [-1,2,-1] except for $n = 256$. However, in most cases the opposite is true for the matrix type [-1,u,-1]. These raw times are shown in tables 7.2a and 7.2b.

7.8.2 Comparison of the Parallel Implementations

Results were obtained from the four recursive versions and two non-recursive versions of the Cuppen's method outlined in section 7.6.

The results use the notation Tr for recursive implementations of parallel recursive algorithms. The numerical results were obtained for the four versions of the algorithm which are outlined in section 7.6.1. The first method which simply implements the function calls using two "THREADcreates" and terminates with two "THREADjoins" is indicated by p. The second implementation using the same method but with the first call done by *THREADcreate* and the second using a conventional call, is indicated by $p0$. The third implementation which uses the processors count, is indicated by $p1$. The fourth implementation which tests the level of recursion, is indicated by $p2$. Results were obtained for the two non-recursive

versions outlined in section 7.6.2. The first implementation using the simple non-recursive version is indicated by *Tnrsp* and the second implementation using linked lists is indicated by *Tnrlp*.

We tested the algorithms using from one up to six processors with matrices of sizes 100(100)400 and power of two size (i.e. 2^n, $6 \leq n \leq 8$). The methods are not efficient when applied to smaller matrices because of the parallel overheads.

For simplicity the results for the comparison of the parallel implementations used the same tolerance in the root finding stage and in the deflation stage (i.e. $\varepsilon = \varepsilon_r = \varepsilon_d$). In this section the comparison has no matrix multiplication parallelism included.

The sequential times were obtained from the non-recursive linked list algorithm (*Tnrlp*). This was slightly better than the other versions. The parallel and sequential times were, of course, obtained for the same type and size of matrix.

To show the performance of the four variant parallel recursive and two different parallel non-recursive versions, we plot in figures 7.4 and 7.5 ($\varepsilon = 1e-6$), and 6 and 7 ($\varepsilon = 1e-8$) mean efficiencies against number of processors for the matrix type [-1,2,-1], and in figures 7.8 and 7.9 ($\varepsilon = 1e{-}6$), and 7.10 and 7.11 ($\varepsilon = 1e{-}8$) mean efficiencies against number of processors for the matrix type $[-1, u, -1]$. Figures 7.12 ($\varepsilon = 1e{-}6$) and 7.13 ($\varepsilon = 1e{-}8$) show actual efficiencies using 2 processors for the matrix type [-1,2,-1], and figures 7.14 ($\varepsilon = 1e-6$) and 7.15 ($\varepsilon = 1e-8$) show similar plots for the 6 processors. Figures 7.16 ($\varepsilon = 1e-6$) and 7.17 ($\varepsilon = 1e-8$) show actual efficiencies using 2 processors for the matrix type [-1,u,-1], and figures 7.18 ($\varepsilon = 1e-6$) and 7.19 ($\varepsilon = 1e-8$) show similar plots and same matrix type for 6 processors.

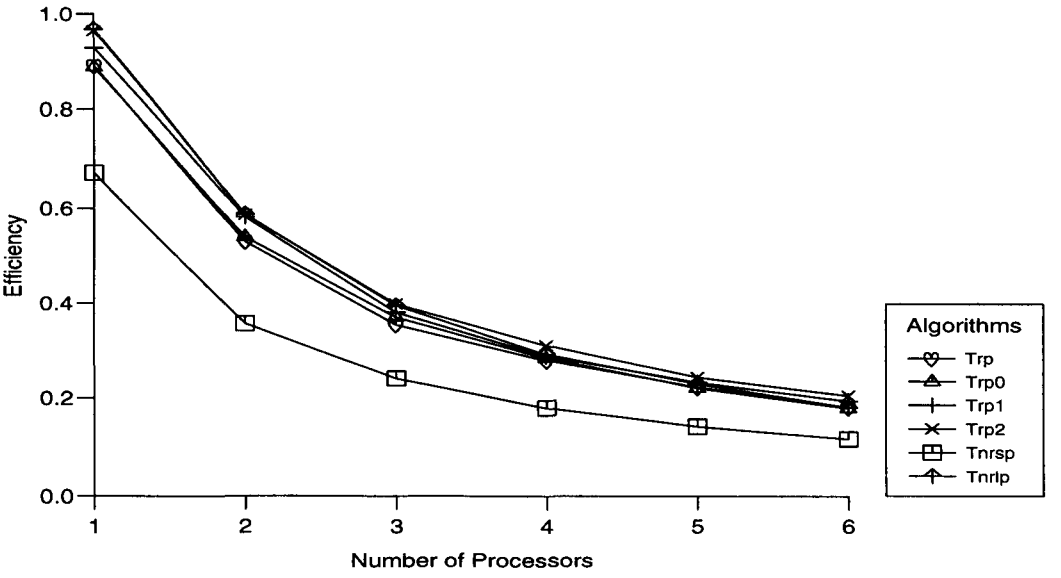


Fig. 7.4 Mean Efficiencies of the $\epsilon = 1e-6$,(-1,2,-1)

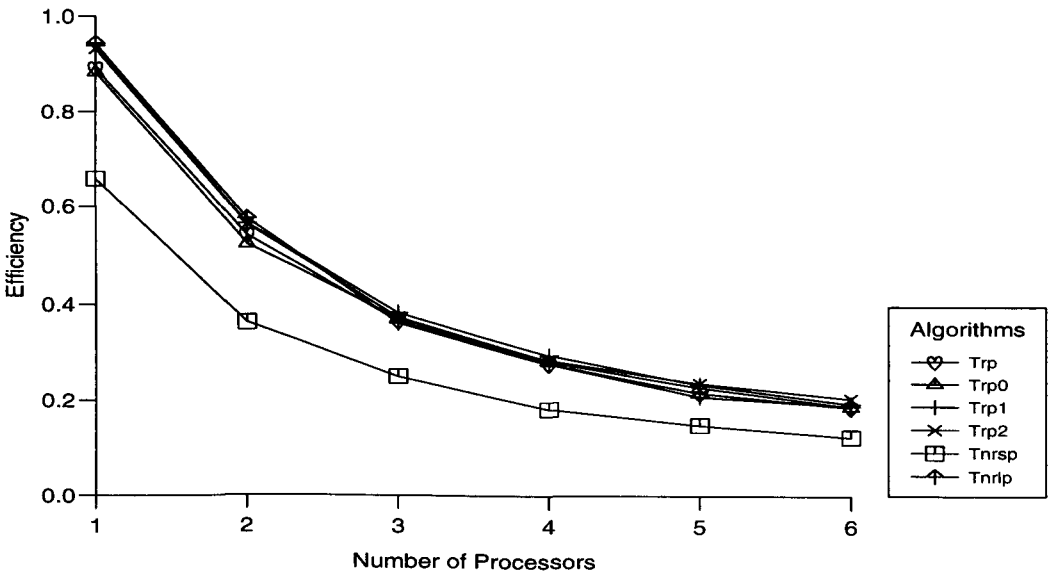


Fig. 7.5 Mean Efficiencies of the $\epsilon = 1e-8$,(-1,2,-1)

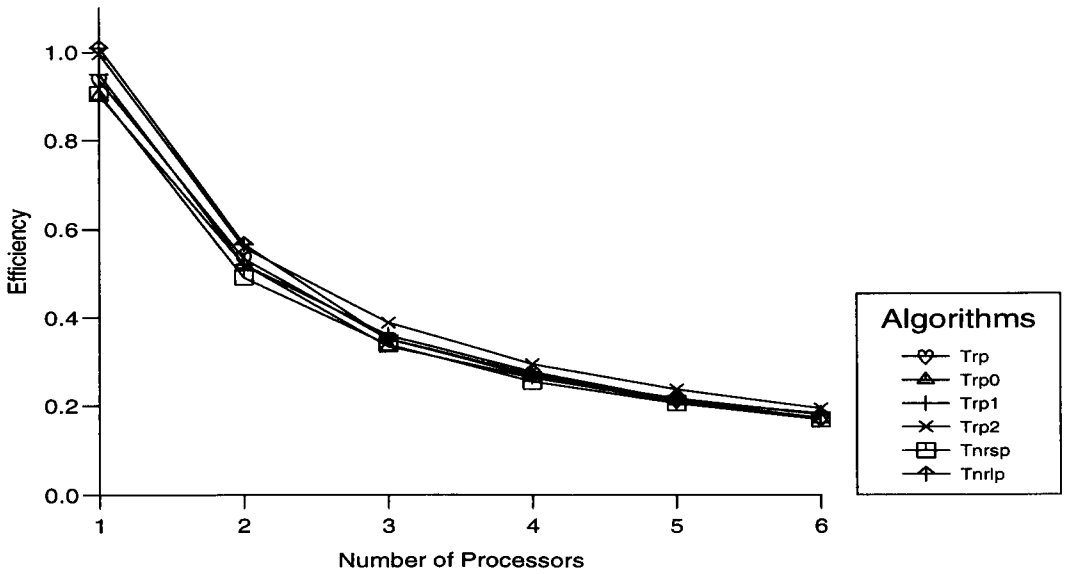


Fig. 7.6 Mean Efficiencies of the $\varepsilon = 1e-6$,(-1,u,-1)

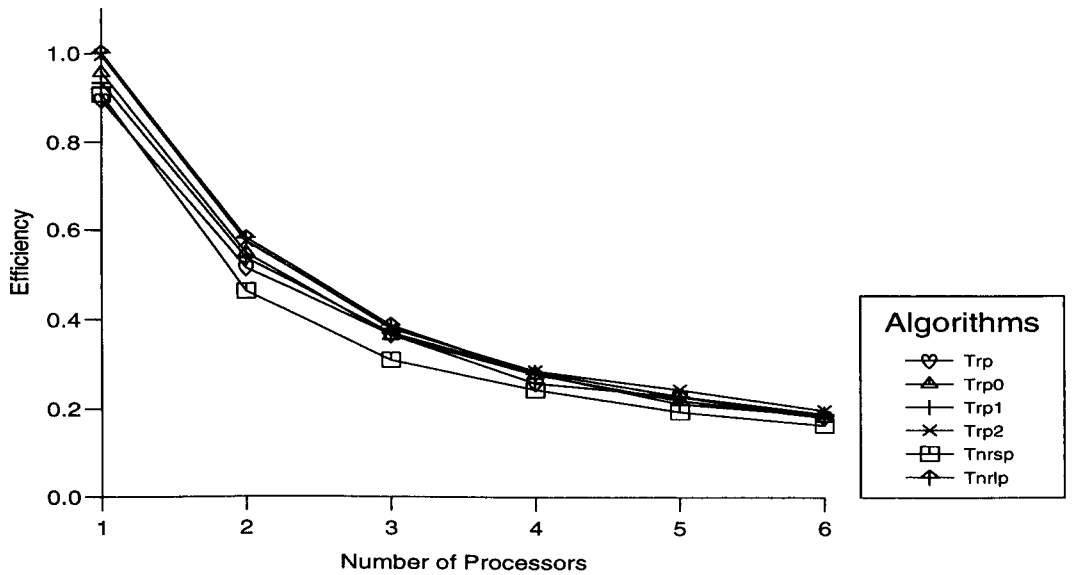


Fig. 7.7 Mean Efficiencies of the $\varepsilon = 1e-8$,(-1,u,-1)

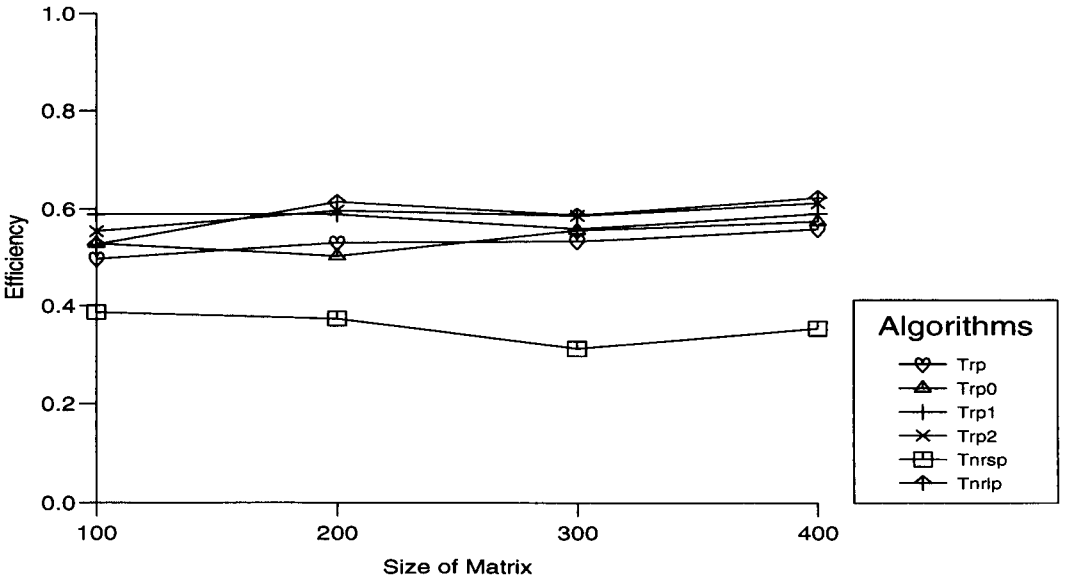


Fig. 7.8 Prcs 2 Efficiencies of the $\varepsilon = 1e-6$,(-1,2,-1)

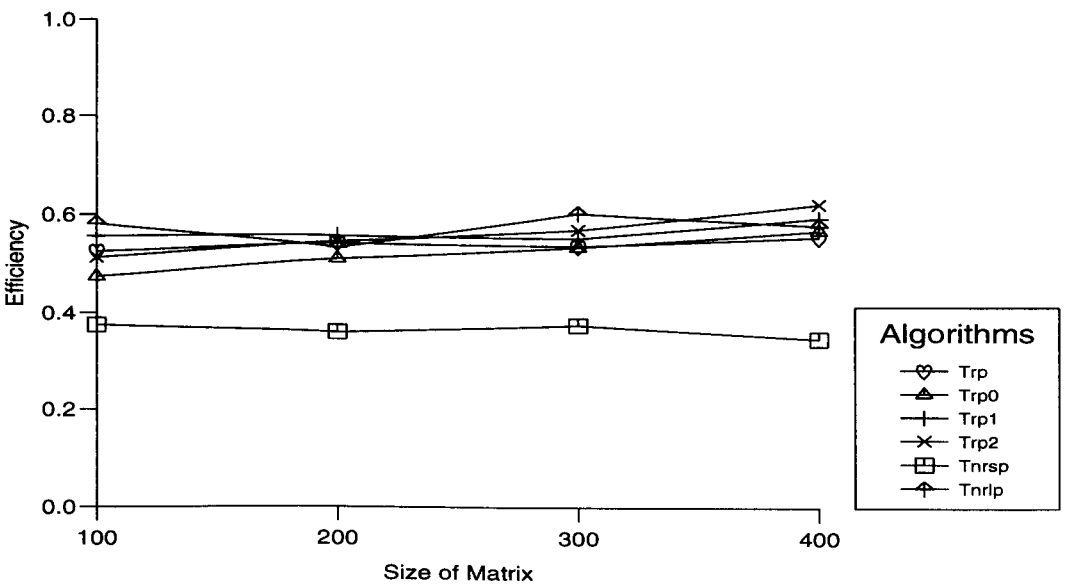


Fig. 7.9 Prcs 2 Efficiencies of the $\varepsilon = 1e-8$,(-1,2,-1)

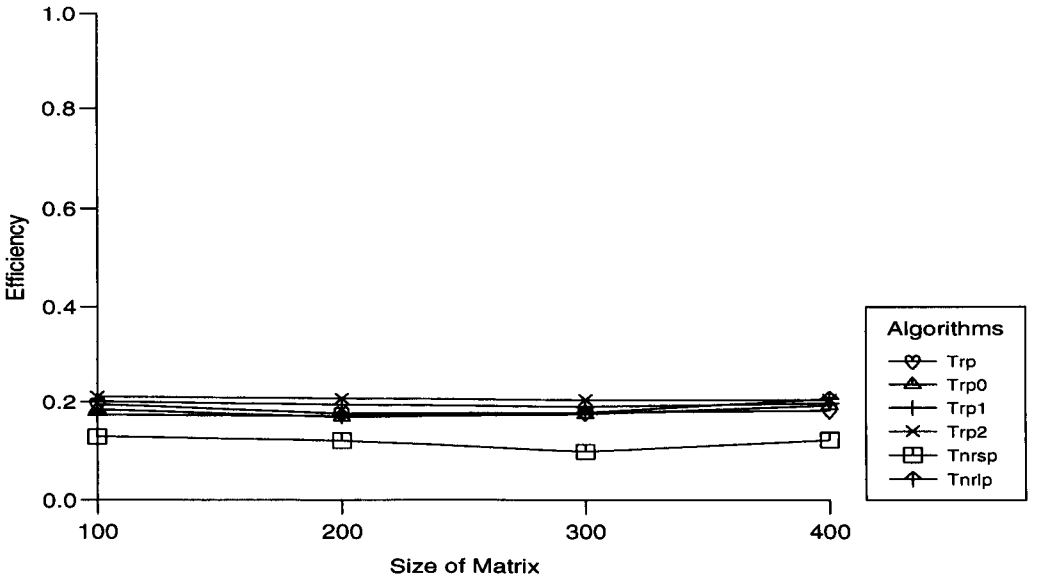


Fig. 7.10 Prcs 6 Efficiencies of the $\varepsilon = 1e-6$,(-1,2,-1)

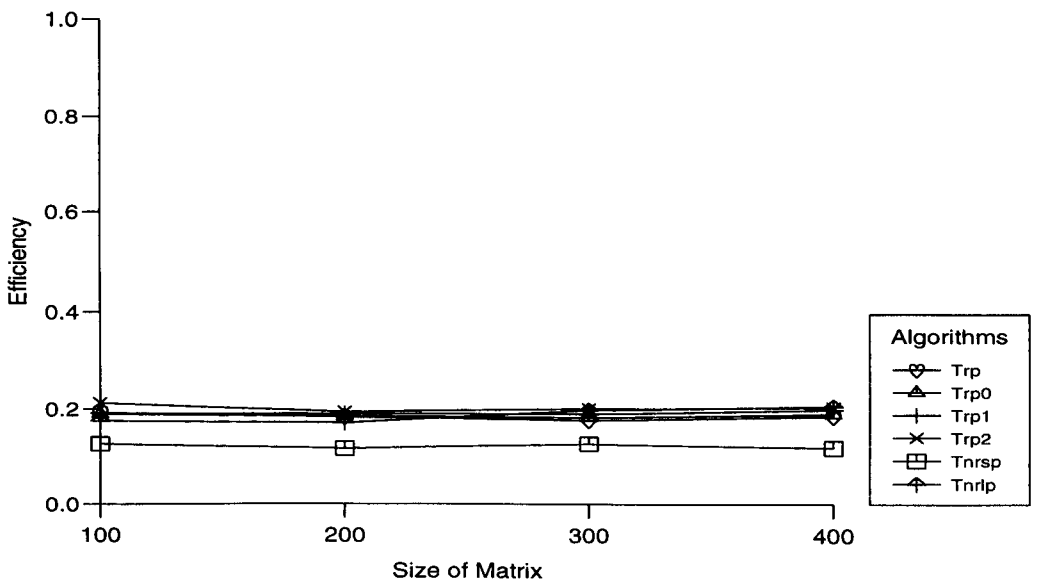


Fig. 7.11 Prcs 6 Efficiencies of the $\varepsilon = 1e-8$,(-1,2,-1)

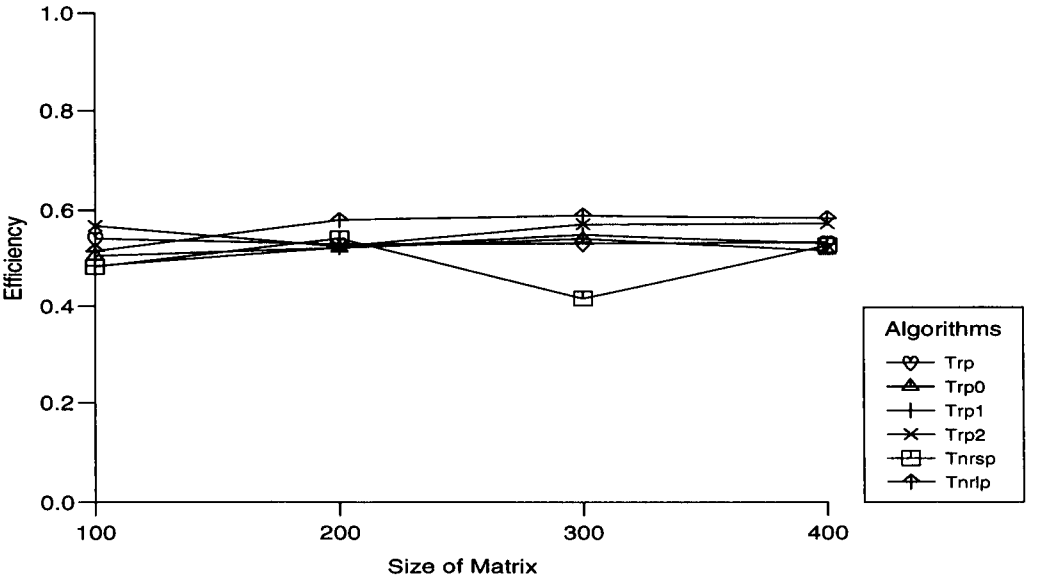


Fig. 7.12 Prcs 2 Efficiencies of the $\varepsilon = 1e-6$,(-1,u,-1)

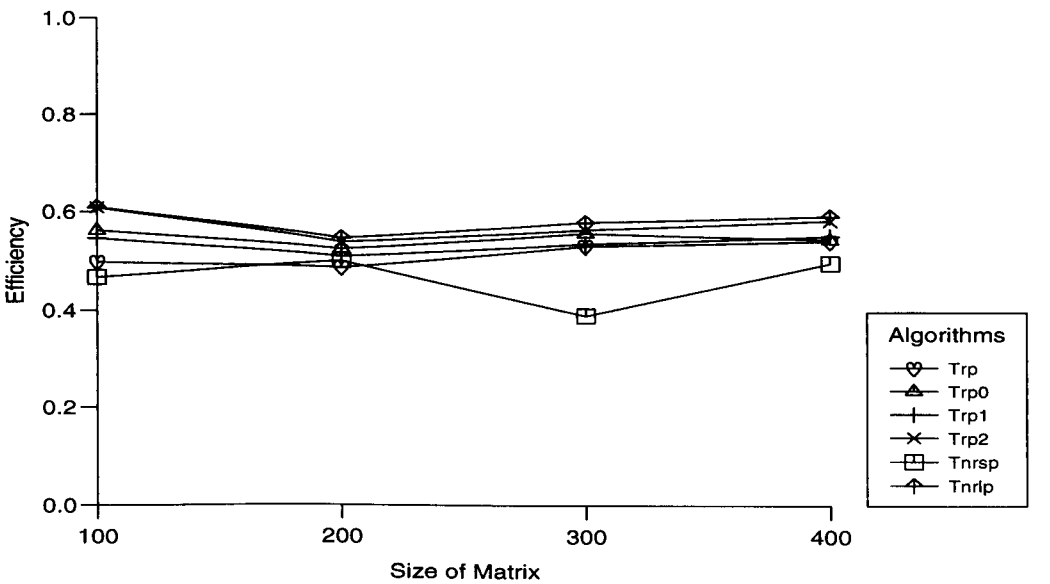


Fig. 7.13 Prcs 2 Efficiencies of the $\varepsilon = 1e-8$,(-1,u,-1)

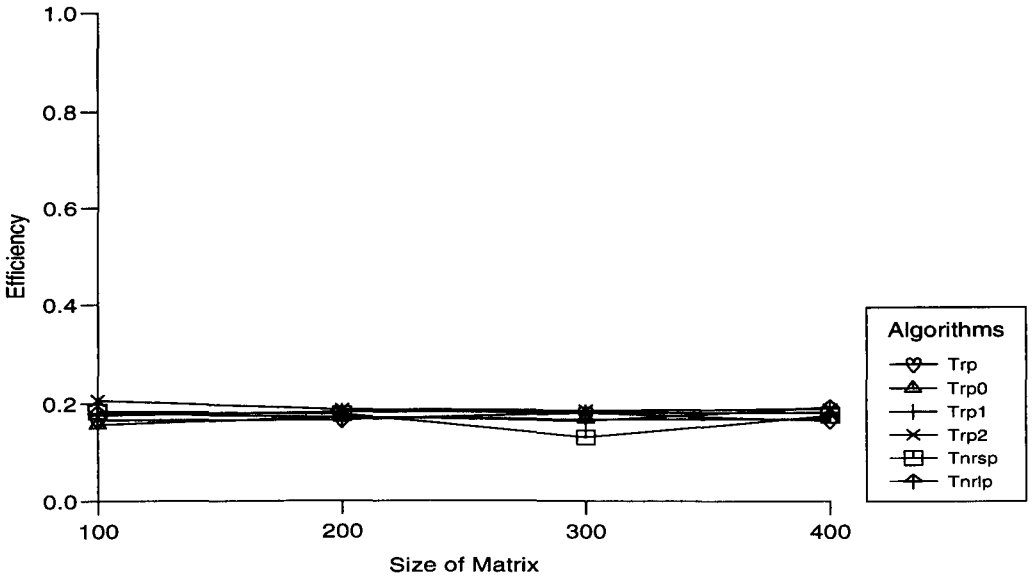


Fig. 7.14 Prcs 6 Efficiencies of the $\varepsilon = 1e-6$,(-1,u,-1)

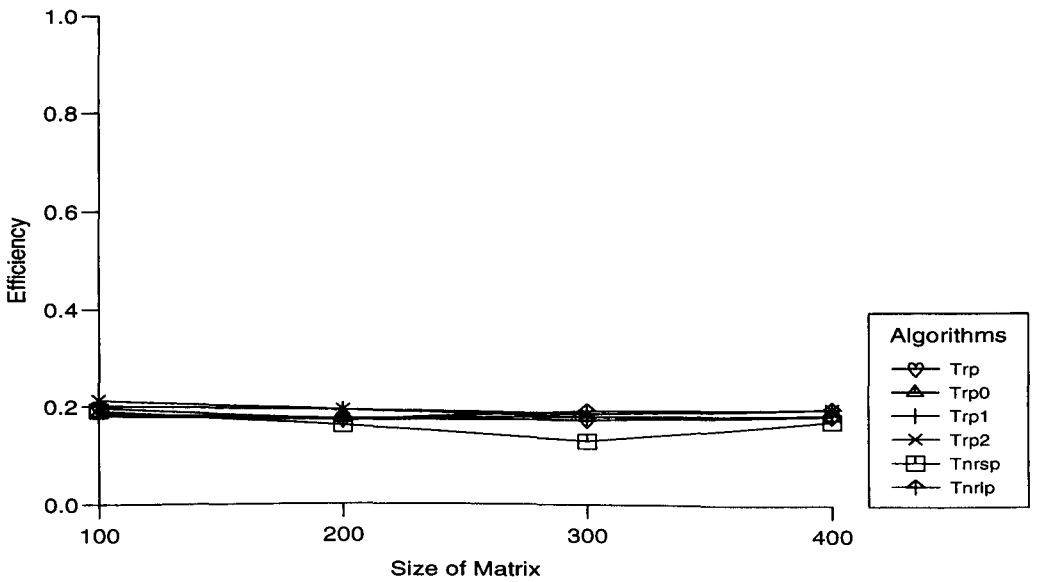


Fig. 7.15 Prcs 6 Efficiencies of the $\varepsilon = 1e-8$,(-1,u,-1)

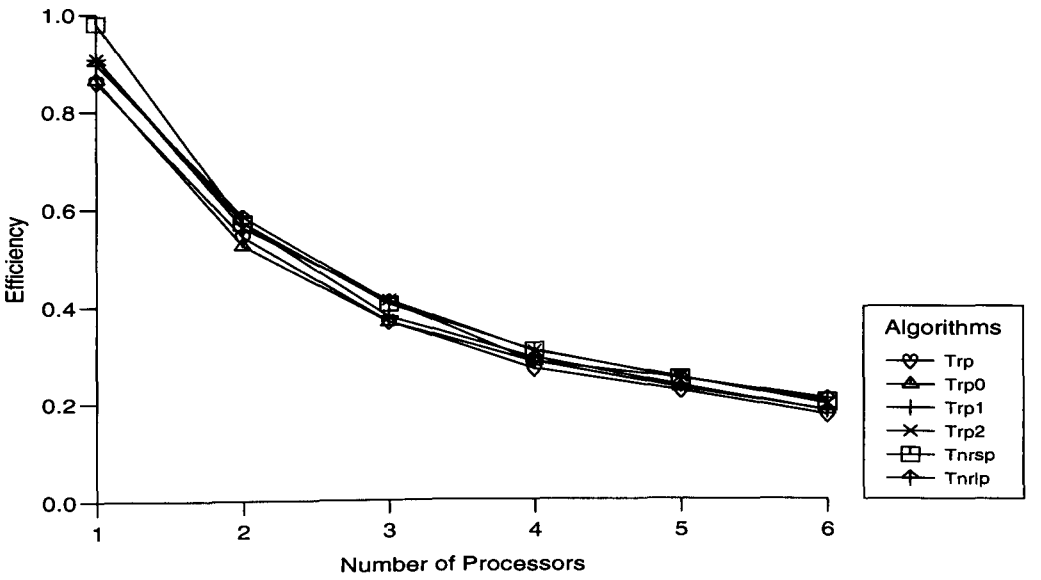


Fig. 7.16 Power of 2, Mean Efficiencies of the $\varepsilon = 1e-6$,(-1,2,-1)

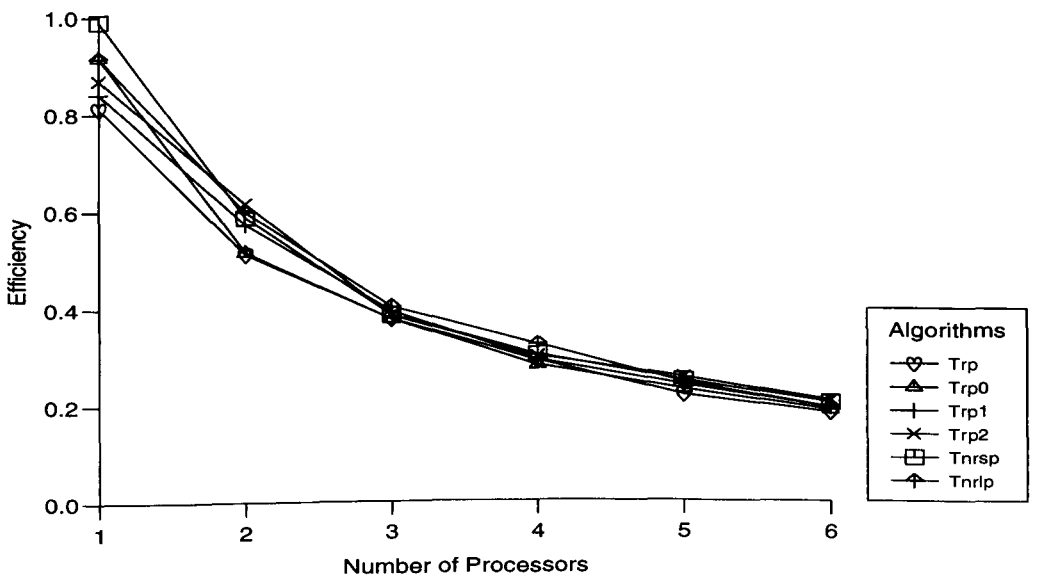


Fig. 7.17 Power of 2, Mean Efficiencies of the $\varepsilon = 1e-8$,(-1,2,-1)

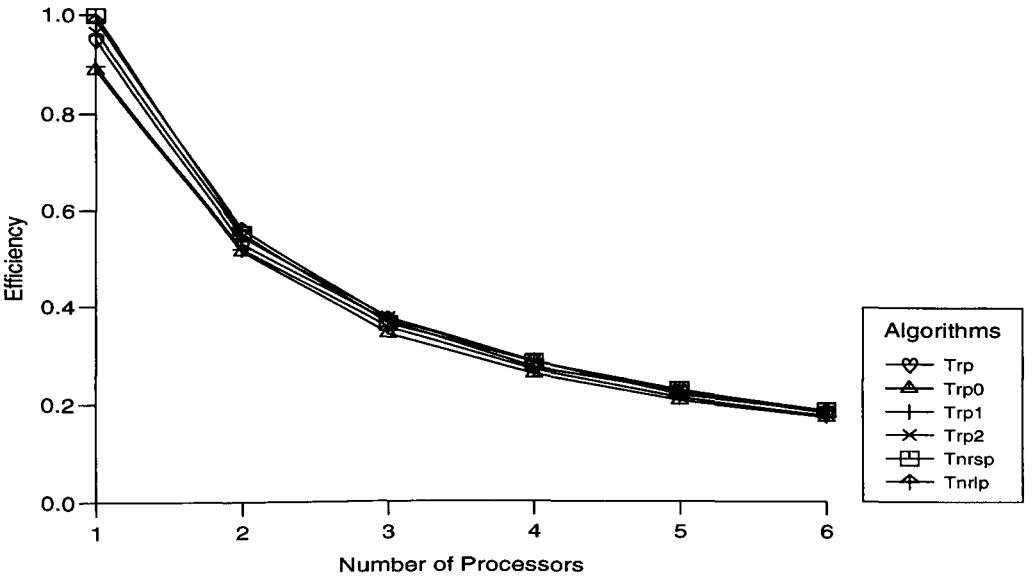


Fig. 7.18 Power of 2, Mean Efficiencies of the $\varepsilon = 1e-6$,(-1,u,-1)

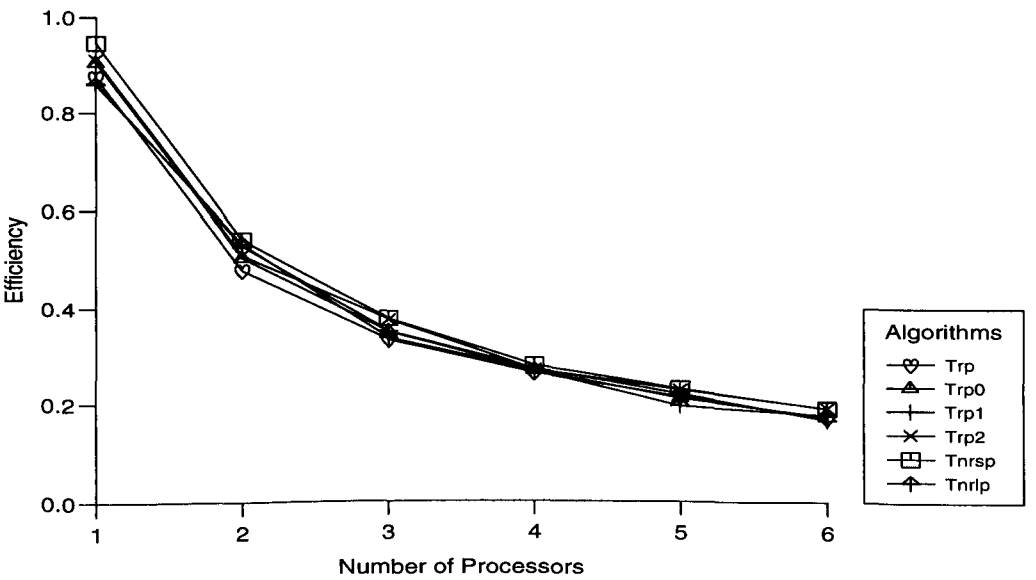


Fig. 7.19 Power of 2, Mean Efficiencies of the $\varepsilon = 1e-8$,(-1,u,-1)

The differences between the parallel recursive versions are relatively small, but overall the parallel recursive version (*Trp2*) seems to be marginally better than the other versions. This may be because in this version (*Trp2*) new threads are only created at the top levels, and threads are not created for the lower levels where the amount of work is small. The results do show that having a lot more threads than processors does not cause a severe loss of efficiency due to the extra overhead.

In the non-recursive (linked list) version tasks are dynamically allocated to the threads while the combination process is similar to the recursive version. Another important point is that even though the sizes of submatrices are the same the version *Tnrlp* is relatively better than the recursive versions when using 1 or 2 processors. This is shown in figures 7.8, 7.9, 7.12, and 7.13. When using 4 or more processors, the version *Tnrlp* gave quite similar efficiency to the recursive versions.

As expected, the simple non-recursive parallel implementation *Tnrsp* is slower than all the other versions. The main reasons for this are firstly the extra work involved in this method and secondly, the uneven allocation of submatrix sizes to the available threads. Taking into account the analysis in sections 7.6 and 7.7.2 where we have considered the relative costs of different versions, we now compare here the ratio of the sequential and the parallel times (e.g. for 2 processors) in order to see whether the ratio of the times agree with theory. The first comparison is carried out testing the problems with the matrix type [-1,2,-1] using $\varepsilon = 1e-6$ as follows:

	Matrix [-1,2 ,-1]		Matrix $[-1, u, -1]$	
n	$Sequential_{ratio}$	$Parallel_{ratio}$	$Sequential_{ratio}$	$Parallel_{ratio}$
100	1.409	1.430	1.155	1.123
200	1.505	1.481	1.134	0.977
300	1.613	1.765	1.289	1.280
400	1.581	1.524	1.051	1.013

Table 7.3: The Comparison for the Ratio of the Times for ℓ_{nrec} and ℓ_{rec} versions

Note that the ratio for 300×300 matrices is larger than the others, as expected though it is not very close to the ratio predicted by the simplified model. Possible reasons why the measured ratios are not close to the predicted ones may be:

- The assumption that the time for root finding is equal for roots at different levels, even though the calculation on different levels is not identical,
- differences at higher levels are ignored,
- the effect of deflation on the timings.

As we pointed out in the previous section, even though the simple non-recursive version gives more deflation (i.e. for 2^n problem size) than the recursive version and the non-recursive (linked list) versions (i.e. in all cases except using $\varepsilon = 1e - 8$ with the matrix type [-1,u,-1]), the times are still significantly worse. One possible additional reason for this may be that tasks are allocated to threads in a predetermined (scattered) ordering which gives less flexibility and so affects the performance of the implementation *Tnrsp*. Thus, although one thread may encounter significant savings when

deflation occurs, the gain may not be shared by the calculation as a whole unless the effects of deflation are evenly distributed to threads while solving the deflated problem and multiplying matrices.

The parallel implementation *Tnrsp* executes most efficiently when its workload (submatrix sizes) is evenly allocated to the available threads. This is illustrated by matrix sizes a power of 2 in tables 7.2a, 7.2b, and 7.2c as well as in figures 7.16-19. In some cases the implementation *Tnrsp* is relatively better than the other implementations for one processor using the two different test matrix types and epsilon values for these matrix sizes. When the number of processors is increased the version *Tnrsp* is still competitive with other versions for matrix type [-1,2,-1] and as well as for matrix type [-1,u,-1]. This is shown in figures 7.16-19.

7.8.3 Comparison of the Additional Parallelisation for Matrix Multiplication

In this section we consider the results of the versions using parallel matrix multiplication in all the implementations presented in this chapter. The results in sub-section 7.8.2 show that without additional parallelisation of the matrix multiplication the efficiencies of implementations are very poor. Results in this section show that if we have computed the matrix multiplication in parallel then the implementations can achieve moderate efficiency.

The sequential times used for the comparison of these parallel implementations were obtained from the non-recursive linked list algorithm (*Tnrlp*) as before for matrices of sizes 100(100)400. But in contrast, for matrices of power of two size (i.e. 2^n, $6 \leq n \leq 8$) sequential times were obtained from the simple non-recursive algorithm (*Tnrsp*) which were slightly better than the other versions, whereas, in the previous section the

sequential times for the *Tnrlp* version were used for these power of 2 matrix size.

The figures below show the performance of the four variant parallel recursive and two different parallel non-recursive versions using parallel matrix multiplication. We plot in figures 7.4a and 7.5a $\varepsilon = 1e-6$), and 7.6a and 7.7a ($\varepsilon = 1e-8$) mean efficiencies against number of processors for the matrix type [-1,2,-1], and in figures 7.8a and 7.9a ($\varepsilon = 1e-6$), and 7.10a and 7.11a ($\varepsilon = 1e-8$) mean efficiencies against number of processors for the matrix type $[-1, u, -1]$. Figures 7.12a ($\varepsilon = 1e-6$) and 7.13a ($\varepsilon = 1e-8$) show actual efficiencies using 2 processors for the matrix type [-1,2,-1], and figures 7.14a ($\varepsilon = 1e-6$) and 7.15a ($\varepsilon = 1e-8$) show similar plots for the 6 processors. Figures 7.16a ($\varepsilon = 1e-6$) and 7.17a ($\varepsilon = 1e-8$) show actual efficiencies using 2 processors for the matrix type [-1,u,-1], and figures 7.18a ($\varepsilon = 1e-6$) and 7.19a ($\varepsilon = 1e-8$) show similar plots and same matrix for 6 processors.

To show the performance for matrices with size a power of 2, we plot mean efficiencies against number of processors for the matrix type [-1,2,-1] in figures 7.16a ($\varepsilon = 1e-6$) and 7.17a ($\varepsilon = 1e-8$), and those for the matrix type $[-1, u, -1]$ in figures 7.18 and 7.18a ($\varepsilon = 1e-6$), and 7.19 and 7.19a ($\varepsilon = 1e-8$).

The results for this extra parallelisation in the matrix multiplication part show substantial improvements, and in fact when using 1 processor some parallel recursive versions gave better times than the best sequential implementation used for comparison. It is not obvious to us why this abnormality occurred. When the number of processors increases, in some cases the efficiency curves slightly decrease and then again increase using matrix type $[-1, 2, -1]$ as well as $[-1, u, -1]$ for matrices with sizes 100(100)400.

Again using additional parallelisation of the matrix multiplication, the differences between the parallel recursive versions are relatively small. Note that the parallel recursive version *Trp2* does not seem to be the best recursive version. It looks as though the parallel recursive version *Trp* is marginally better than the other recursive versions.

The non-recursive linked list version (*Tnrlp*) is overall more efficient than the other implementations using both matrix types $[-1, 2, -1]$ and $[-1, u, -1]$ when using more than one processor. The simple non-recursive version (*Tnrsp*) displayed very poor performance when we tested the version *Tnrsp* using matrix type $[-1, 2, -1]$ but when we tested this algorithm with matrix type $[-1, u, -1]$, the efficiency curve is improved and in some cases is close to the efficiency curves of the recursive versions.

Comparing power of 2 size matrices efficiency curves in figures 7.16a-19a we observe that in the recursive version there is a very rapid decrease in the efficiency followed by a small increase, as the number of processors increases when using $\varepsilon = 1e-6$ for test matrix type $[-1, 2, -1]$. It is also worth noting that the efficiency for 2 processors is especially low. This is shown in figure 7.16a. But this decrease and increase in the efficiency curves are small when using $\varepsilon = 1e-8$ value and matrix type $[-1, u, -1]$. The non-recursive linked list (*Tnrlp*) version is still relatively better than other versions as in the previous section. In some cases the simple non-recursive version is relatively better than the other versions for 1 processor. When the number of processors are greater than one, the efficiency curves is quite close to the version *Tnrlp*.

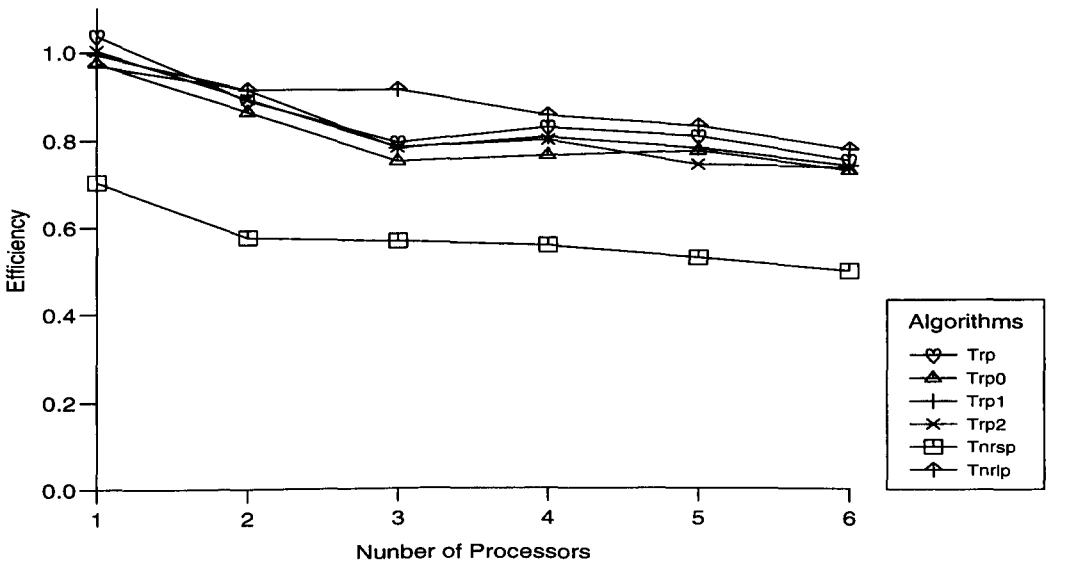


Fig. 7.4a Mean Efficiencies of the $\varepsilon = 1e-6$, (-1,2,-1)

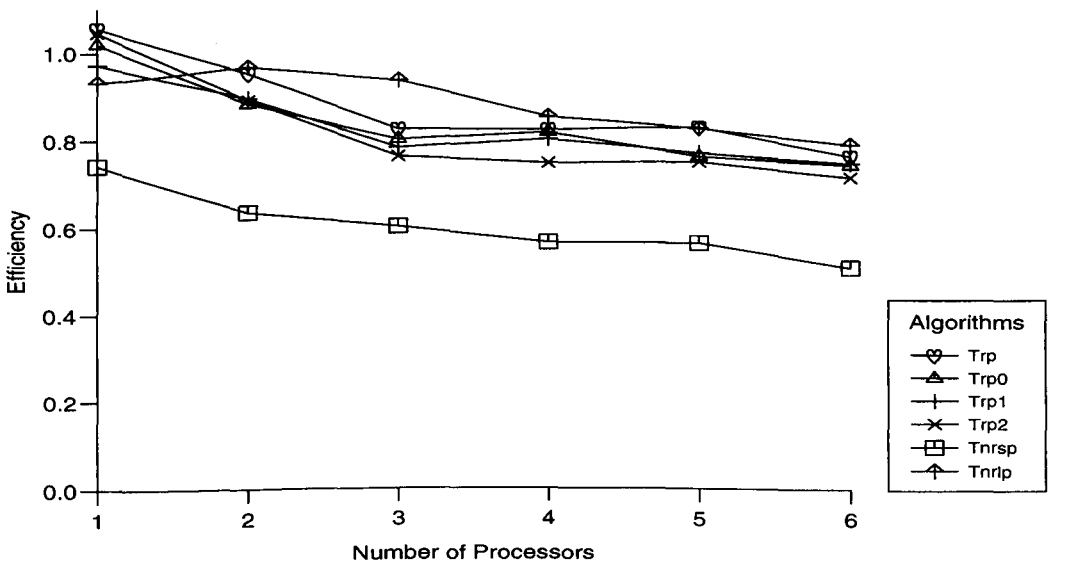


Fig. 7.5a Mean Efficiencies of the $\varepsilon = 1e-8$, (-1,2,-1)

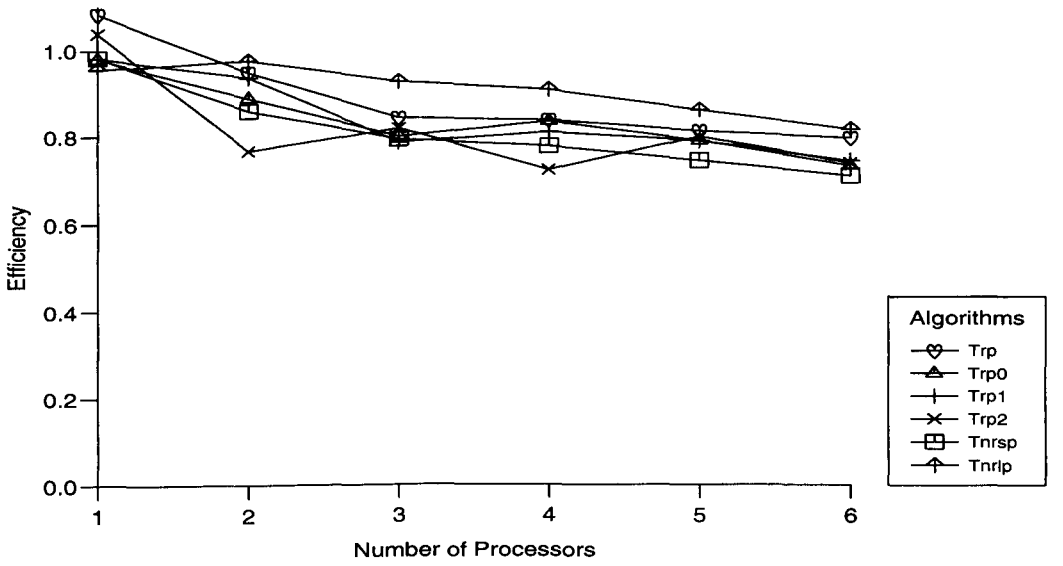


Fig. 7.6a Mean Efficiencies of the $\varepsilon = 1e-6$,(-1, u,-1)

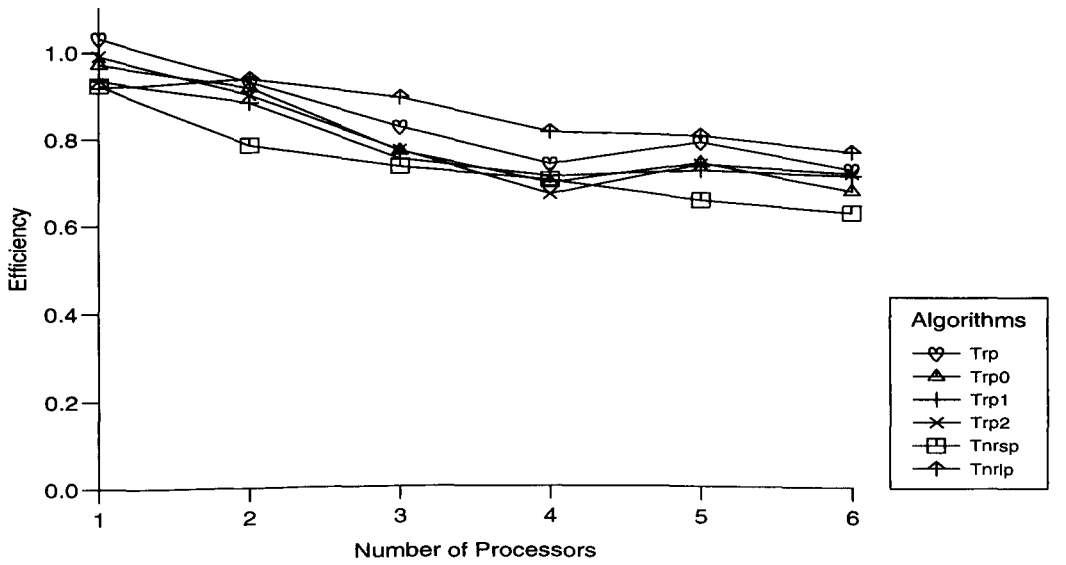


Fig. 7.7a Mean Efficiencies of the $\varepsilon = 1e-8$,(-1, u,-1)

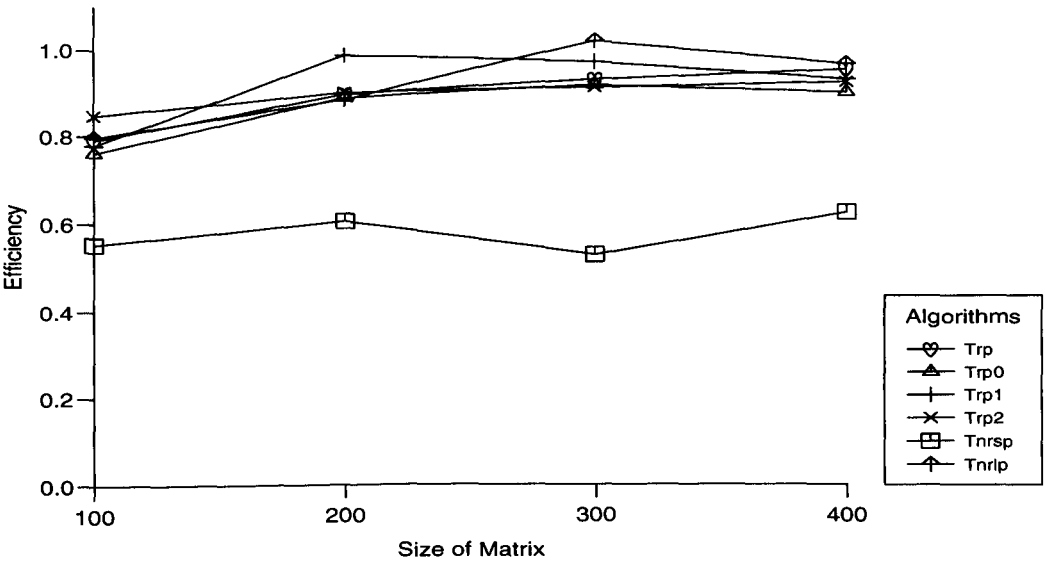


Fig. 7.8a Prcs 2 Efficiencies of the $\varepsilon = 1e-6$, (-1,2,-1)

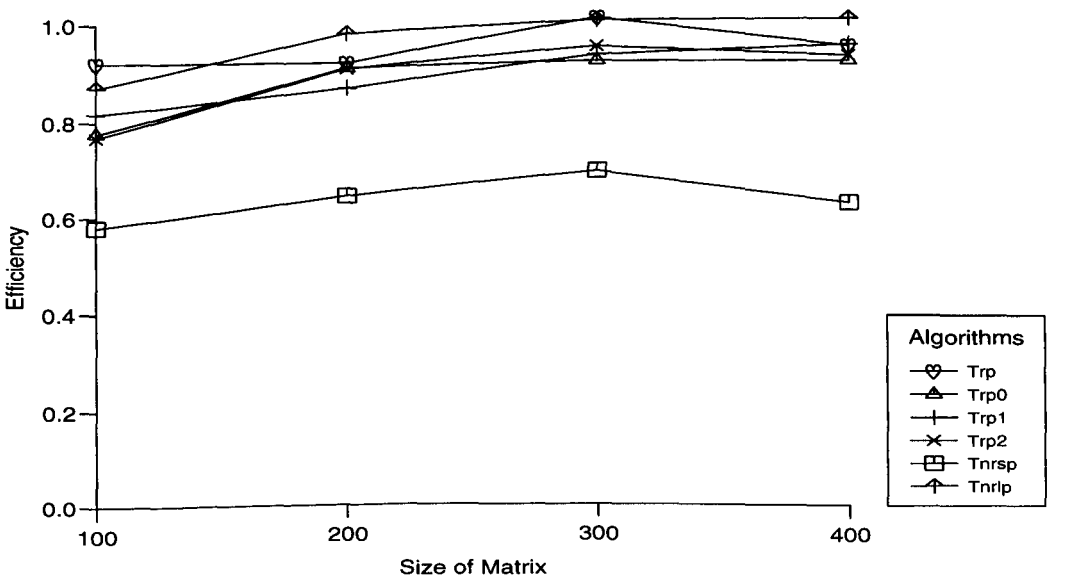


Fig. 7.9a Prcs 2 Efficiencies of the $\varepsilon = 1e-8$, (-1,2,-1)

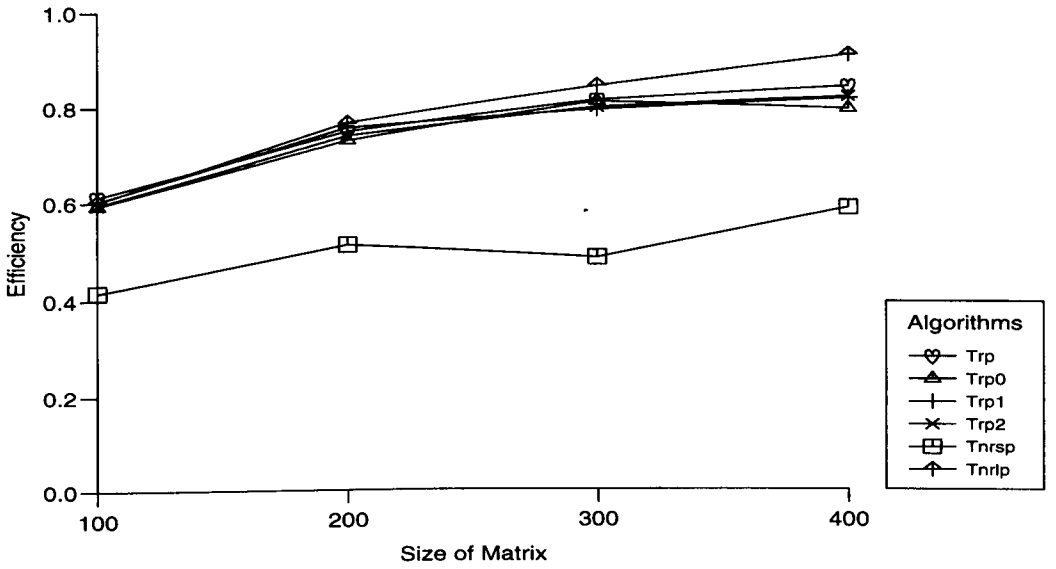


Fig. 7.10a Prcs 6 Efficiencies of the $\epsilon = 1e-6$, (-1,2,-1)

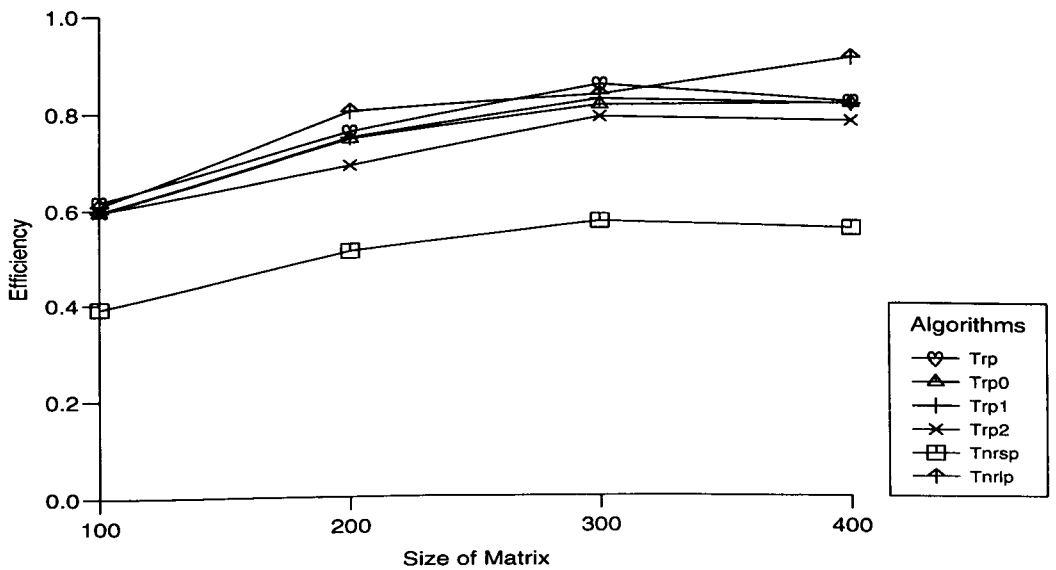


Fig. 7.11a Prcs 6 Efficiencies of the $\epsilon = 1e-8$, (-1,2,-1)

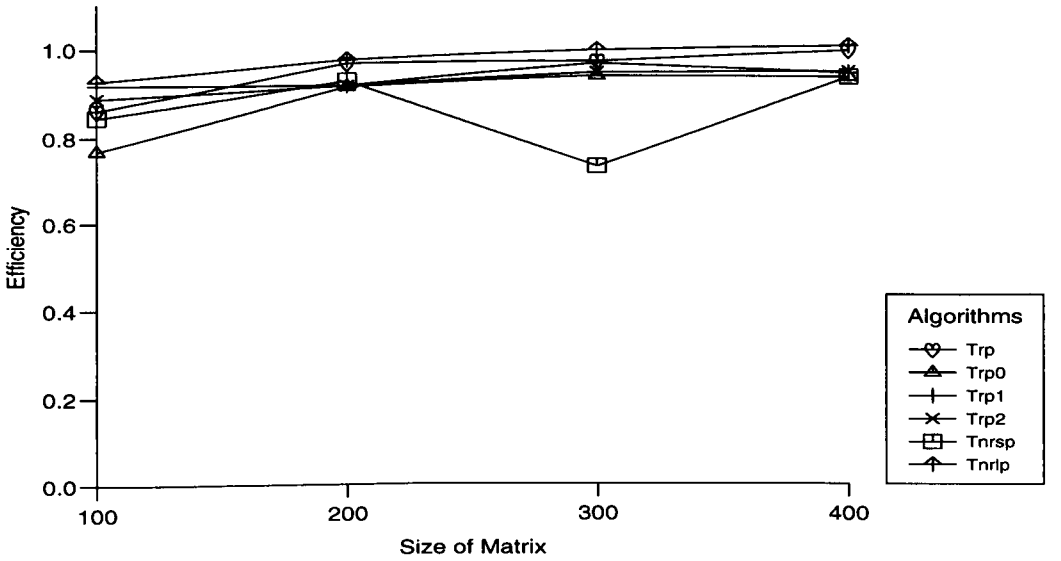


Fig. 7.12a Prcs 2 Efficiencies of the $\varepsilon = 1e-6$,(-1,u,-1)

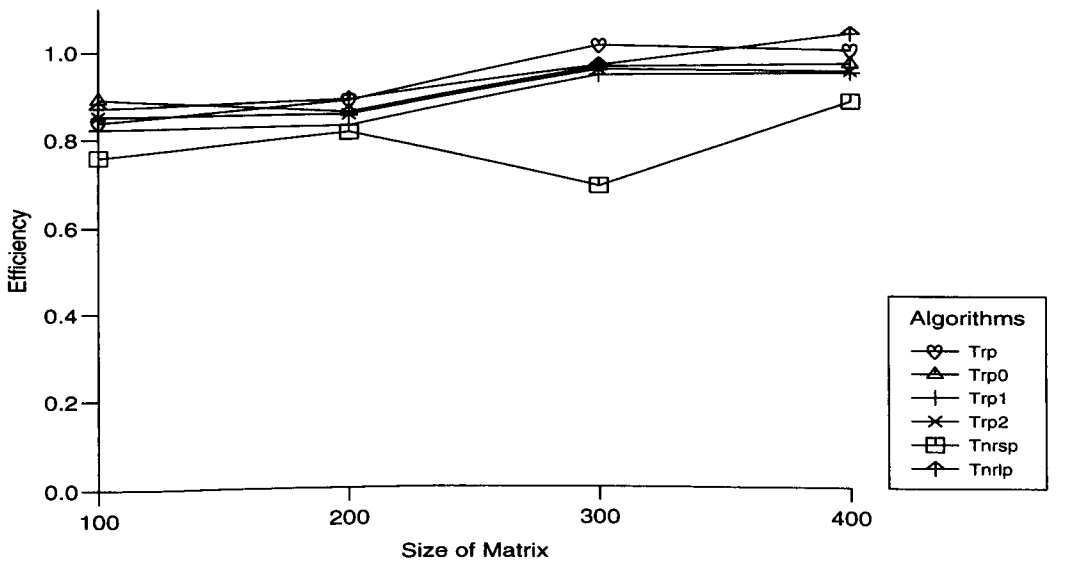


Fig. 7.13a Prcs 2 Efficiencies of the $\varepsilon = 1e-8$,(-1,u,-1)

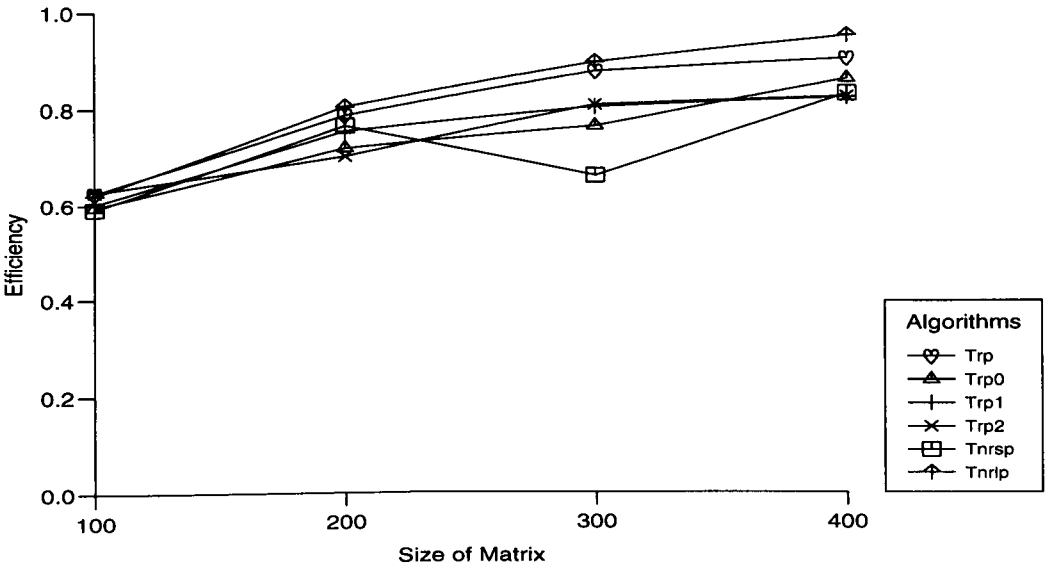


Fig. 7.14a Prcs 6 Efficiencies of the $\varepsilon = 1e-6$,(-1,u,-1)

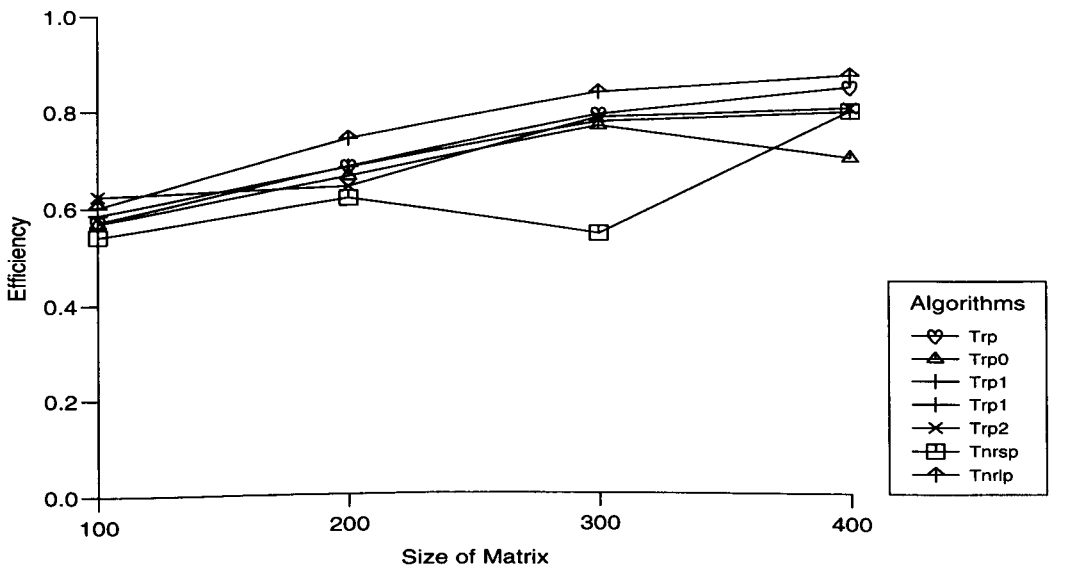


Fig. 7.15a Prcs 6 Efficiencies of the $\varepsilon = 1e-8$,(-1,u,-1)

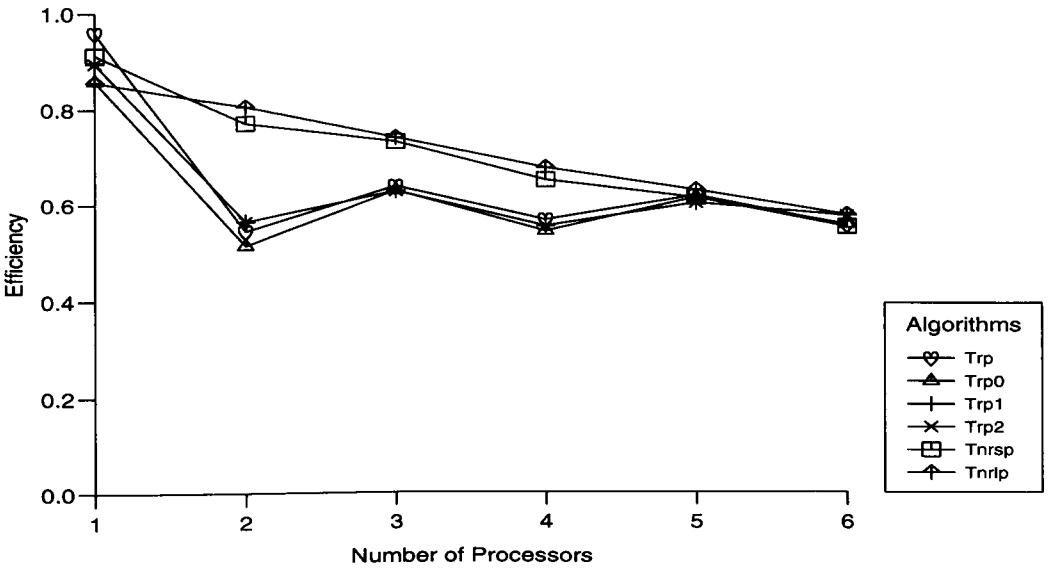


Fig. 7.16a Power of 2, Mean Efficiencies of the $\varepsilon = 1e-6$,(-1,2,-1)

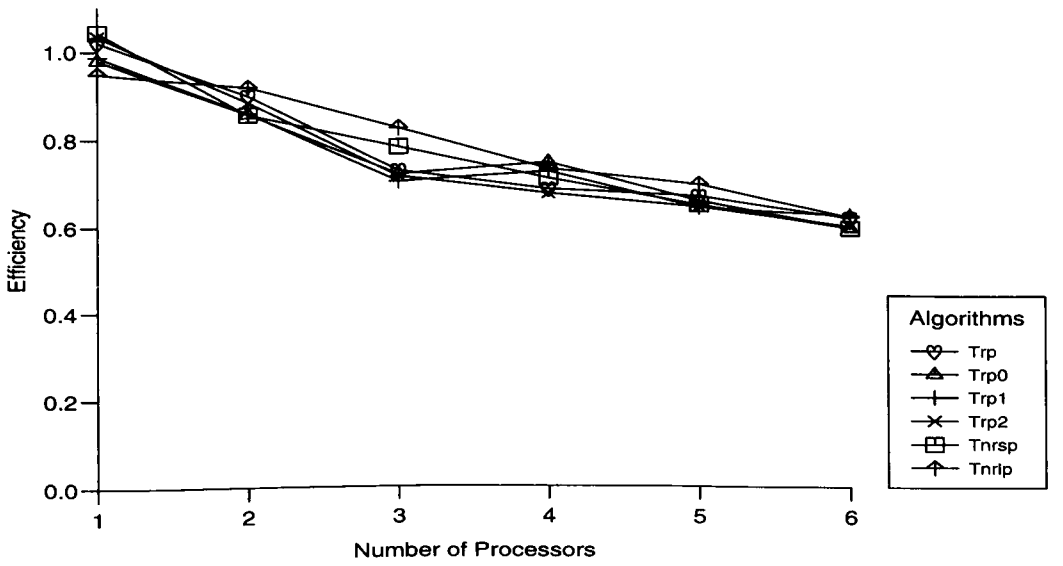


Fig. 7.17a Power of 2, Mean Efficiencies of the $\varepsilon = 1e-8$,(-1,-2,-1)

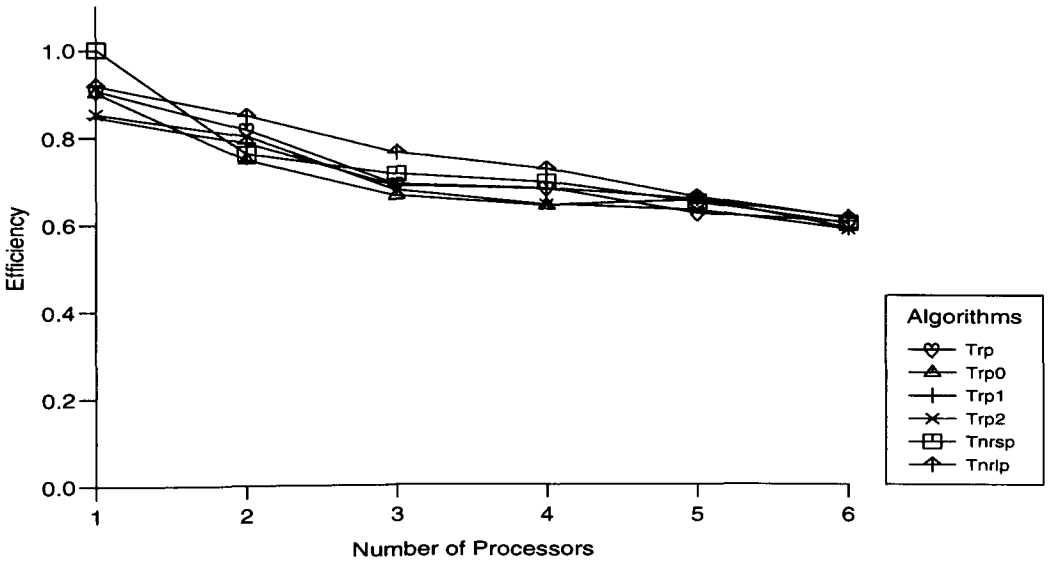


Fig. 7.18a Power of 2, Mean Efficiencies of the $\varepsilon = 1e-6$,(-1,u,-1)

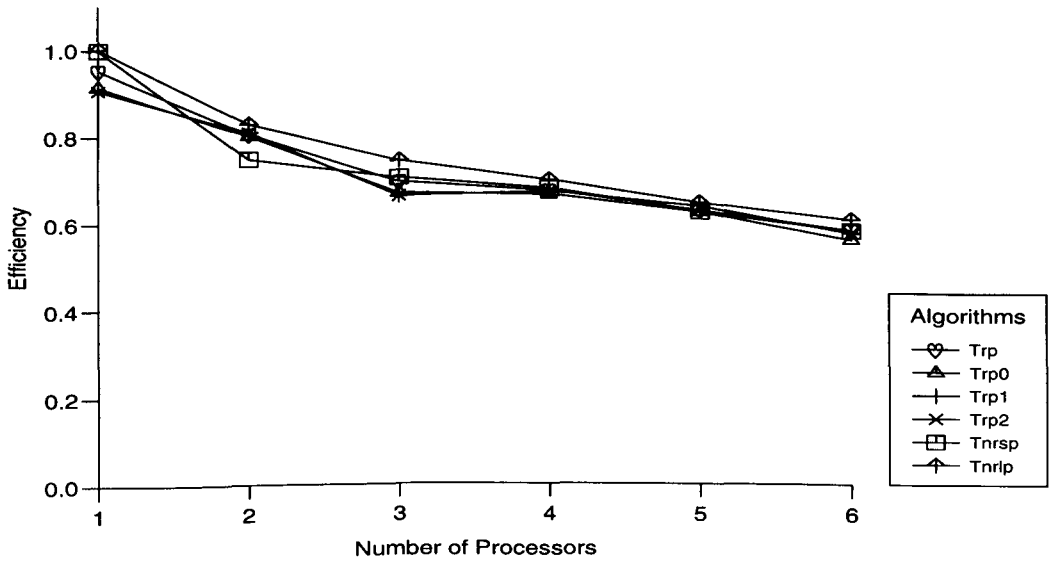


Fig. 7.19a Power of 2, Mean Efficiencies of the $\varepsilon = 1e-8$,(-1,u,-1)

Comparing power of 2 size matrices efficiency curves in figures 7.16a-19a we observe that in the recursive version there is a very rapid decrease in the efficiency followed by a small increase, as the number of processors increases when using $\varepsilon = 1e-6$ for test matrix type $[-1, 2, -1]$. It is also worth noting that the efficiency for 2 processors is especially low. This is shown in figure 7.16a. But this decrease and increase in the efficiency curves are small when using $\varepsilon = 1e-8$ value and matrix type $[-1, u, -1]$. The non-recursive linked list (*Tnrlp*) version is still relatively better than other versions as in the previous section. In some cases the simple non-recursive version is relatively better than the other versions for 1 processor. When the number of processors are greater than one, the efficiency curves is quite close to the version *Tnrlp*.

The efficiency of the implementation *Tnrsp*, however, suffered badly from the load imbalance for matrix size not a power of 2. As we pointed out in section 7.7.2 for the 300×300 matrix, at the next to top level the matrix multiplication operation costs are almost four times larger than these for recursive and non-recursive linked list (*Tnrlp*) versions. The efficiency depends very much on matrix sizes. Moreover, as we mentioned before, when the tasks are evenly distributed to threads the version *Tnrsp* shows significantly improved performance and competes with the other versions in the case using the matrix sizes a power of 2.

7.9 Conclusions

In this chapter we have investigated some recursive and non-recursive parallel divide-and-conquer methods on the shared memory machine. In particular, recursive and non-recursive (*Tnrsp*) sequential versions are compared in terms of deflation using different types of matrices. All the implementations are much faster for matrix type [-1,2,-1] than for

matrix type [-1,u,-1] which indicates that the saving from the deflation is considerable.

The implementation *Tnrlp* avoids the difficulty in allocating tasks evenly across the available threads in the parallel implementation (*Tnrsp*). There is a clear conclusion that the non-recursive parallel implementation using linked lists *Tnrlp* is a good alternative algorithm to the recursive algorithm. In most cases *Tnrlp* is marginally better than the recursive versions.

As expected, a significant decrease in the execution times occurs when the matrix multiplications procedure is parallelised. The result was an increase of efficiency of 40-55 % in most cases. This observation shows that it is very important to include parallelisation of the matrix multiplication in all these algorithms.

CHAPTER 8

Conclusions and Future Research

We have presented several algorithms in Numerical Linear Algebra on a shared memory multiprocessor system. In this chapter, we summarise our major contributions in this thesis. Extensions to the current work will be suggested, plans for future work will be given, and closing remarks will be stressed.

8.1 Summary

As pointed out earlier, the main aim of this thesis was to examine the improvement of efficiency by implementing in parallel some algorithms for numerical linear algebra which are widely used computational tools in science and engineering research. This research is becoming increasingly dependent upon the development and implementation of efficient parallel algorithms. Numerical linear algebra algorithms are indispensable in science and engineering research and this thesis has attempted to collect and describe a selection of some of the more important algorithms and their

parallel implementations. The algorithms, including the solution of the linear system of equations $Ax = b$ using QR and LU decompositions, reduction of a general matrix to Hessenberg form, reduction of a real symmetric matrix to tridiagonal form, and Cuppen's divide-and-conquer method for finding all of the eigenvalues and corresponding eigenvectors of a real symmetric tridiagonal matrix were implemented on the system available to us namely the Encore Multimax. Our evaluation focus was not to rewrite an existing program but to restructure the algorithms in order to produce efficient serial and parallel methods.

We have concentrated in particular on algorithms written in the C++ programming language. C++ is an object-oriented programming language, which can provide various types of matrix classes. The use of C++ implies some storage organisation for array elements. The primitive arrays provided in C++ use storage by rows. As far as we are aware there is little previous work on parallelising these algorithms using C++. Consequently in this thesis we have considered implementation of the serial and parallel methods using the object-oriented programming language C++. The algorithms have been designed for the C++ programming language using the Encore Parallel Threads [31] package. The package provides mechanisms for synchronisation. The synchronisation strategy to use in any situation depends on the program's structure and the computer system's flexibility. Each strategy has its benefits and costs.

The comparisons were carried out by measuring the elapsed time for

each implementation. These comparisons were carried out using both row and column representation of the matrix for most of the implementations described in chapters 4, 5, and 6. This was made easy by the use of some C++ matrix classes which were altered internally. Comparisons were also carried out both with and without array bound checking. These extra comparisons were included to compare the algorithms under different conditions, and make spurious conclusions about the relative merits of the algorithms less likely. Chapter 7 presents an experimental evaluation of the effect of deflation on accuracy. A number of test matrices were used to test Cuppen's algorithm since the amount of deflation in this method depends on the test matrix. The results of the recursive and non-recursive parallel implementations of this algorithm were compared. An extra comparison was also presented for the results of the additional parallelisation in the matrix multiplication part.

There are some general observations from the experiments carried out using the shared memory architecture in this thesis and the following overall conclusions can be drawn:

- Firstly, the experimental results presented show that dynamic task allocation can sometimes be very effective on this machine, and that very high efficiency is often obtainable with careful construction of the parallel algorithms even for relatively small matrices.

- Secondly, as we pointed out in chapter 4, in the solution of algebraic equations (or QR and LU decompositions) all column updates do not

need to be performed before the next pivotal column is treated. Various alternative ways of allocating column updates were investigated where pivotal and normal columns were treated simultaneously. The most effective of these was an algorithm which carried out as many column updates to a column as possible by the same processor, and when no further updates were possible due to the relevant pivotal column not having been treated, then starting on a different column. The results show that the column based multiple column approach for LU decomposition does have significant advantages on a shared memory multiprocessor.

- Thirdly, restructuring the algorithms to avoid splitting the algorithms into stages gives significant improvement particularly for reduction to Hessenberg form (i.e. the fifth implementations *Hella* and *Helle*). The idea here is that the columns and rows are allocated dynamically to the threads, and counts for the columns and the *rowb* rows are used. This was based on the observation made clear in the dataflow diagram that the pivotal, column and *rowb* row updates are not dependent on the *rowa* row updates being completed. When any thread finds no column or *rowb* row available for allocation, then any *rowa* rows which are ready for updating are allocated instead.

Comparisons were carried out between the performance of the five implementations of algorithms for a general matrix to be reduced to upper Hessenberg form. The fifth implementations, *Hella* and *Helle*

are clearly better than the other implementations. As the number of processors increases the graphs for this algorithm are very nearly horizontal indicating little loss and remarkably high efficiencies, with both row and column representations of the matrix.

Comparisons were also carried out between the performance of the three different synchronisation mechanisms in the second implementation. In most cases the "Locks" synchronisation mechanism was more efficient than those using "Monitors" or "Semaphores" at least for this particular problem. The synchronisation mechanism to use in any situation depends on the program's structure and the computer system's flexibility. Each mechanism has its benefits and costs. In general locks are only suitable for short waits, semaphores and monitors are more appropriate for longer waits. Here we organised the program so that the waits are usually short.

- Fourthly, the algorithm described for reduction to Hessenberg form can also be used for the reduction of a symmetric matrix to tridiagonal form, and there are some rather obvious savings from the symmetry. However, the alternative algorithm described in chapter 6 involve less arithmetic but has much greater data dependencies so limiting the possibilities for parallelisation. It is concluded from the experimental results of parallelising this algorithm that some minor changes did give some improvements, but all the efficiencies obtained were much poorer than those for the algorithms described in chapter 5.

- Finally, we have investigated some recursive and some non-recursive parallel implementations of a divide-and-conquer method for finding all of the eigenvalues and corresponding eigenvectors of a real symmetric tridiagonal matrix in chapter 7. As we stated in that chapter, the motivation for a non-recursive version was to compare this with the recursive version and to investigate the relationship between the recursive and non-recursive implementations of Cuppen's divide-and-conquer method.

  An important point that arises in the implementation of Cuppen's algorithm concerns the choices of tolerance ε_r in the computation of the root finding stage and tolerance ε_d in the computation of the deflation stage. From the point of view of numerical computation, in most cases there is no advantage to be gained if one epsilon is much larger than the other, although choosing ε_d slightly larger than ε_r seems to be helpful.

  The experimental results illustrated that the differences between the parallel recursive versions are relatively small. The experiments show that having more threads than processors does not cause a severe loss of efficiency due to the extra overhead.

  There is a clear conclusion that the non-recursive linked lists serial and parallel implementation is a good alternative algorithm to the recursive algorithm. In most cases this algorithm is marginally better than the recursive versions.

As expected, a significant decrease in the execution times occurs when the matrix multiplications procedure is parallelised. The result was an increase of efficiency of 40-55 % in most cases. This observation shows that it is very important to include parallelisation of the matrix multiplication in all these algorithms. The experimental results illustrate that the non-recursive linked list version is still relatively better than the recursive versions.

8.2 Future Research

To extend the work reported in this thesis, we identify the following areas that deserve further study.

All the above algorithms were implemented on a shared-memory Encore Multimax using the C++ programming language. These algorithms can be compared using the other popular parallel architectures such as distributed memory machines and also shared memory machines with different relative speeds for arithmetic and storage. This extra comparison should include the comparison of the algorithms under different environments and to investigate how generally applicable they are using different architectures.

The results described in this thesis indicate that for QR and LU decomposition and Hessenberg reduction very high parallel efficiencies are obtainable on the shared memory machine used, as long as care is taken in the implementation of the algorithms. For reduction of a symmetric matrix to tridiagonal form the results are not so good, but here a combination with

finding the eigenvalues as suggested by Dongarra et al [26] might compensate for this problem. However, detailed study and experiments should be further explored.

The divide-and-conquer method might also be applied to the related problem of computing the singular value decomposition (SVD) of a real bidiagonal matrix.

Comparison of the Cuppen method with the QL method, Sturm sequence (or bisection with inverse iteration), and QR method for the symmetric tridiagonal eigenvalue problem deserve to be considered.

8.3 Closing Remarks

In general, we encountered memory problems during our implementation. For example, in the algorithms for QR decomposition the memory fills up with large matrices when using the *newmat* matrix package because when using this package for the manipulation of matrices we need much more space than using a simple matrix class. The second example is when the objects are declared in the parallel part of the programs. Then, in most cases, a memory problem would occur with large matrix sizes. This requires allocation of storage space on the heap. The memory problem is disappointing and limits the size of the problem that we can use for testing our algorithm. Although such problems could be somewhat avoided by declaring the object before the parallel section and passing it as a reference parameter and also addressed by improving the management of dynamic

structures, they would not be completely solved.

The objective of this research as pointed out from the outset was to develop serial and parallel algorithms. From the research that we have carried out and which is reported in this thesis we can conclude that effective exploitation of parallelism with numerical linear algebraic problems is dependent on several factors some of which include the nature of the problem to be solved and the type of architecture on which we intend to implement the problem.

Bibliography

1. Akl, S.G., The Design and Analysis of Parallel Algorithms, Prentice-Hall, 1989.

2. Almasi, G. S. and Gottlieb, A., Highly Parallel Computing, The Benjamin/Cummings Publishing Company, 1989.

3. Bischof, C., QR Factorization Algorithms for Coarse-Grained Distributed Systems, Tech. Rep. TR 88-939, Dep. of Computer Science, Cornell University, Ithaca, NY, 1988.

4. Bischof, C. and Van Loan, C., The WY Representation for Products of Householder Matrices, SIAM J. Statist. Comput., 8 (1987), pp. s2-s13.

5. Booch, G., Object-Oriented Analysis and Design with Applications, The Benjamin/Cummings Publishing Company, 1994.

6. Bunch, J.R., Neilsen, C.P., Sorensen, D.C., Rank one Modification of the Symmetric Eigenproblem, Numer. Math. 31 (1978) 31-48.

7. Burden,R.L. and Faires, J.D., Numerical Analysis, Fifth Edition, PWS Publishing Company, 1993.

8. Bus, J.C. and Dekker, T.J., Two Efficient Algorithms with Guaranteed Convergence for Finding a Zero of a Function, TOMS 1, (1975) pp 330-345.

9. Businger, P.A., Reduction a Matrix to Hessenberg Form, Math. Comput. 23(1969) 819-821.

10. Bowdler, H., Martin, R., and Wilkinson, J., The QR and QL Algorithms for Symmetric Matrices, Numer. Math., 11 (1968), pp. 227-240.

11. Calahan, D., Block-Oriented Local-Memory-Based Linear Equation Solution on the Cray-2: Uniprocessor Algorithms, in Proc. Intl. Conf. Par. Processing, IEEE Computer Society Press, New York, 1986, pp. 375-378.

12. Catanzaro, B., Multiprocessor System Architectures, Sun Microsystems, 1994.

13. Chang, H.Y., Utku, S., Salama, M., and Rapp, D., A Parallel Householder Tridiagonalisation Stratagem Using Scattered Square Decomposition, Parallel Comput. 6 (1988) 297-311.

14. Chu, E. and George, A., Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor. Parallel Computing 5, (1987) pp. 65-74.

15. Coplien, J.O., Advanced C++ Programming Styles and Idioms, Addison- Wesley, 1992.

16. Cosnard, M., Muller, J., and Robert, Y., Parallel QR Decomposition of a Rectangular Matrix, Numer. Math., 48 (1986), pp. 239-249.

17. Cuppen, J.J.M., A Divide and Conquer Method for the Symmetric

Tridiagonal Eigenproblem, Numer. Math., 36 (1981), pp. 177-195.

18. Davies, R.B., Newmat05: An Experimental Matrix Package in C++, DSIR, New Zealand, 1992.

19. Dijkstra, E. W., The Structure of the 'THE' Multiprogramming System. Communications of the ACM 11, (1968) pp. 341-346.

20. Doeppner, T. W. Jr., Threads, A system for the Support of Concurrent Programming. Brown University Department of Computer Science Technical Report CS-87-11, 1987.

21. Dongarra, J.J. and Duff, I.S., Advanced Architecture Computers, Argonne National Laboratory Report, ANL/MCS-TM-57, 1989.

22. Dongarra, J.J, Duff, I.S., Sorensen, D.C., and A. van der Vorst, H., Solving Linear Systems on Vector and Shared Memory Computers, SIAM, Philadelphia, Second Printing 1993.

23. Dongarra, J.J, Gustavson, F.G., and Karp, A., Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine, SIAM Review, 26 (1984), pp. 91-112.

24. Dongarra, J., Sameh, A., and Sorensen, D., Implementation of Some Concurrent Algorithms for Matrix Factorization, Parallel Comput., 3 (1986), pp. 25-34.

25. Dongarra, J.J. and Sidani, M., A Parallel Algorithm for the Nonsymmetric Eigenvalue Problem, SIAM J. Sci. Stat. Comput.

14(1993) 542-569.

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

27. Dongarra, J.J., Sorensen, D.C., and Hammarling, S.J., Block reduction of matrices to condensed forms for eigenvalue computations, J. Computat. Applied Maths 27(1989) 215-227.

28. Dongarra, J.J. and Van de Geijn, R.A., Reduction to Condensed form for the Eigenvalue Problem on Distributed Memory Architectures, Parallel Comput. 18(1992) 973-982.

29. Eckel, B., C++ Inside and Out, McGraw-Hill, 1993.

30. Elden, L., A Parallel QR Decomposition Algorithm, Tech. Rep. Lith-MAT-R-1 988-02, Linkoping University, Linkoping, Sweden, 1987.

31. Encore Computer Corporation, Encore Parallel Threads Manual. No. 724-06210 Rev. A. 1988.

32. Flynn, M. J., 1966. Very High-Speed Computing Systems. Proceedings of the IEEE 54, 12, pp. 1901-1909.

33. Gallivan, K., Jalby, W., Meier, U., and Sameh, A., Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design, Intl. J. Supercomputer Appl., 2(1988), pp. 12-48. Presented at the Level 3 BLAS Workshop, Argonne National Laboratory, January 1987.

34. Gallivan, K.A., Plemmons, R.J., and Sameh, A.H., Parallel Algorithms for Dense Linear Algebra Computations. SIAM Review 32, (1990) pp. 54-135.

35. Geist, G.A. and Davis, G.J., Finding Eigenvalues and Eigenvectors of Unsymmetric Matrices Using a Distributed-Memory Multiprocessor, Parallel Comput. 13(1990) 199-209.

36. Geist, G.A. and Romine, C.H., LU Factorisation Algorithms on Distributed-Memory Multiprocessors Architectures. SIAM J. Sci. Statist. Comput. 9, (1988) pp. 639-649.

37. Goles, E. and Kiwi, M., A Lower Bound on the Computational Complexity of the QR Decomposition on a Shared Memory SIMD Computer. Parallel Computing 18, (1992) pp. 345-354.

38. Golub, G. H., Some Modified Matrix Eigenproblem, SIAM Rev., 15 (1973), pp. 318-334.

39. Golub, G. and Ortega, J.M., Scientific Computing an Introduction with Parallel Computing, Academic Press, 1993.

40. Golub, G.H., and Van Loan, C.F., Matrix Computations, 2nd Ed. Johns Hopkins University Press, Baltimore and London, 1989.

41. Graham, N., Learning C++, McGraw-Hill, 1991.

42. Gu, M., and Eisenstat, S.C., A Stable and Efficient Algorithm for the Rank-one Modification of the Symmetric Eigenvalue Problem, Research

Report YALEU/DCS/RR-916 30 September 1992.

43. Hager, W.W., Applied Numerical Linear Algebra, Prentice-Hall International Editions, 1988.

44. Higham, N.J., A collection of Test Matrices in MATLAB. Numerical Analysis Technical Report, No.172, Dept. of Maths., University of Manchester (1989).

45. Hoare, C. A. R., Monitors: an Operating System Structuring Concept. Communications of the ACM 17, (1974) pp. 549-557.

46. Huang, H., Parallel Algorithms for Symmetric Tridiagonal Eigenvalue Problems, Tech. Rep. CAC Doc. 109, Center for Advanced Computation, University of Illinois, Urbana, IL, 1974.

47. Hwang, K., Advanced Computer Architecture, McGraw-Hill, 1993.

48. Hwang, K. and Briggs, F. A., Computer Architecture and Parallel Processing, McGraw-Hill, 1984.

49. Ipsen, I., and Jessup, E., Solving the Symmetric Tridiagonal Eigenvalue on the Hypercube, SIAM J.Sci. Stat. Comput. 11 (1990) pp. 203-229.

50. Jalby, W. and Philippe, B., Loss of Orthogonality in a Gram-Schmidt Process, Tech. Rep., IRISA, Rennes, France, 1987.

51. Jensen, R., Chaos in Atomic Physics, In Proceedings of the Xth International Conference on Atomic Physics ICAP-X, 1987.

52. Jensen, R. and Shankar, R., Statical Behavior in Deterministic Quantum Systems with Few Degrees of Freedom, Phys. Rew. Lett., 54 (1985), pp. 1879-1882.

53. Jessup, E.R. and Ipsen, I., Improving the Accuracy of Inverse Iteration, SIAM J. Sci. Stat. Comput. 13 (1992) pp. 550-572.

54. Kahan, W., Rank-1 perturbed diagonal's Eigensystem. Unpublished Man., Dept. Computer Science, Stanford University, July 1989.

55. Kalamboukis, T.Z., A Parallel Algorithm for the Dense Symmetric Eigenvalue Problem on a Transputer Array, Parallel Comput. 18(1992) 207-212.

56. Kaya, D. and Wright, K., Parallel Algorithms for LU Decomposition on Shared Memory Multiprocessor, Technical Report Series No. 450, University of Newcastle upon Tyne, Computing Science, November, 1993.

57. Kaya, D. and Wright, K., Parallel Algorithms for Reduction of a General Matrix to upper Hessenberg form on Shared Memory Multiprocessor, Technical Report Series No. 490, University of Newcastle upon Tyne, Computing Science, October, 1994.

58. Krishnakumar, A. and Morf, M., Eigenvalues of a Symmetric Tridiagonal Matrix: A Divide and Conquer Approach, Numer. Math., 48 (1986), pp. 348-368.

59. Kruse, R.L., Data structures and Program Design, Prentice-Hall, 1987.

60. Kuck, D. and Sameh, A., Parallel Computation of Eigenvalues of Real Matrices, in Proc. IFIP Congress 1971, North-Holland, Amsterdam, 1972, pp. 1266-1272.

61. Kuttler, J. and Sigillito, V., Eigenvalues of the Laplacian in two Dimension, SIAM Review, 26 (1984), pp. 163-193.

62. Lewis, T.G. and El-Rewini, H., Introduction to Parallel Computing, Prentice-Hall International, 1992.

63. Li, R. -C., Solving Secular Equations Stably and Efficiently, UC Berkeley Math. Dept. Report, California 94720, 1993.

64. Liu, J.W.H., Computational Models and Task Scheduling for Parallel Sparse Cholesky Factorisation. Parallel Computing 3, (1986) pp. 327-342.

65. Lo, S.S, Philippe, B. and Sameh, A., A Multiprocessor Algorithm for Symmetric Tridiagonal Eigenvalue Problem, SIAM J. Sci. Stat. Comput. 8, (1987) s155-s165.

66. Modi, J.J., Parallel Algorithms and Matrix Computation. Oxford University Press, 1989.

67. Modi, J. and Clarke, M., An Alternative Givens Ordering, Numer. Math., 43 (1984), pp. 83-90.

68. O'Neil, P.V., Advanced Engineering Mathematics, Third Edition, Wadsworth Publishing Company, 1991.

69. Ortega, J. and Voigt, R., Solution of Partial Differential Equations on Vector and Parallel Computers. SIAM Review 27, (1985) pp.149-240.

70. Parlett, B., The Symmetric Eigenvalue Problem, Prentice-Hall, Englewood Cliffs, NJ, 1980.

71. Pohl, I., Turbo C++, The Benjamin/Cummings Publishing Company, 1991.

72. Pohl, I., Object-Oriented Programming Using C++, The Benjamin/ Cummings Publishing Company, 1993.

73. Pothen, A. and Raghavan, P., Orthogonal Factorization on a Distributed Memory Multiprocessor, Tech. Rep. CS-87-24, Pennsylvania State University, Computer Science Dept., University Park, PA, 1987.

74. Pothen, A., Somesh, J., and Vemulapati, U., Orthogonal Factorization on a Distributed Memory Multiprocessor, in Hypercube Multiprocessors 1987, M. T. Heath, ed., SIAM, Philadelphia, 1987, pp. 587-596.

75. Quinn, M.J., Designing Efficient Algorithms for Parallel Computers, McGraw-Hill, 1987.

76. Rice, J.R., Experiments on Gram-Schimidt Orthogonalization, Math. Comp. 20, (1966) 325-328.

77. Sameh, A., Numerical Algorithms on the Cedar System, Presented at Second SIAM Conference on Parallel Processing, 1985.

78. Sameh, A. and Kuck, D., On Stable Parallel Linear Systems Solvers, JACM, 25 (1978), pp. 81-91.

79. Schreiber, R. and Van Loan, C., A Storage-Efficient WY Representation for Products of Householder Transformations, SIAM J. Statist. Comput., 10 (1989), pp. 53-57.

80. Sedgewick, R., Algorithms in C++, Addison-Wesley Publishing Company, 1992.

81. Sevcik, K.C., Application Scheduling and Processors Allocation in Multiprogrammed Parallel Processing Systems, Technical Report CSRI-282, Computer Systems Research Institute University of Toronto, March 1993.

82. Sorensen, D.C., Analysis of pairwise pivoting in Gaussian Elimination. IEEE Trans. Computers C34, (1984) pp. 275-278.

83. Sorensen, D.C., and Tang, P.T.P., On the Orthogonality of Eigenvectors Computed by Divide-and-Conquer Techniques, SIAM J. Numer. Anal. 28 (1991) pp. 1752-1775.

84. Stark, S. and Beris, A.N., LU Decomposition Optimized for a Parallel Computer with a Hierarchical Distributed-Memory. Parallel Computing 18, (1992) pp. 959-971.

85. Stewart, G. W., Introduction to Matrix Computation, Academic Press, New York, 1973.

86. Stoker, M. A., The Exploitation of Parallelism on Shared Memory Multiprocessors, PhD dissertation, University of Newcastle upon Tyne, 1990.

87. Stroustrup, B., Possible Directions for C++. Proceedings of the USENIX C++ Workshop. Santa Fe, NM: USENIX Association November 1987.

88. Stroustrup, B., What is Object-Oriented Programming? IEEE Software val. 5 (1988), pp. 10-20.

89. Stroustrup, B., The C++ Programming Language, Second Edition, Addison-Wesley, 1991.

90. Sun, X. and Bischof, C., A Basis-Kernel Representation of Orthogonal Matrices, Tech. Rep. MCS-P431-0594, Argonne National Laboratory, Mathematics and Computer Science Division, 1994.

91. Tadmor, E. and Gill, D., 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. Statist. Comput., 11(1990), pp. s161-s173.

92. Van de Vorst, J.G.G., The Formal Development of a Parallel Program Performing LU-Decomposition. Acta Informatica 26, (1988) pp. 1-17.

93. Watkins, D.S., Fundamentals of Matrix Computations, John Willey and Sons, Inc., 1991.

94. Weston, J.S. and Clint, M., Two Algorithms for the Parallel Computation of Eigenvalues and Eigenvectors of Large Symmetric Matrices Using the ICL DAP, Parallel Comput. 13(1990) 281-288.

95. Wilkinson, J., The Algebraic Eigenvalue Problem, Claredon Press, Oxford, 1965.

96. Wilkinson, J. and Reinsch, C., Handbook for Automic Computation: Linear Algebra, Vol. 2, Springer-Verlag, Berlin, 1971.

97. Wright, K., Parallel Algorithms for QR Decomposition on a Shared Memory Multiprocessor. Parallel Computing 17, (1991) pp. 779-790.

98. Wright, K. and Kaya, D., Parallel Algorithms for Linear Algebra on a Shared Memory Multiprocessor, 3th Int. Coll. on Numerical Analysis, pp. 209-218, VSP 1995.

99. Zhang, X. and Castaneda, R., Spin-Lock Synchronisation on the Butterfly and KSR1. IEEE. Parallel & Distributed Tech. Vol. 2, Spring Issue, (1994) 51-63.

100. Zhang, H. Moss, W.F., Using Parallel Banded Linear System Solvers in Generalized Eigenvalue Problems. ICASE Report No. 93-71, September 1993.

101. Zhou, B.B. and Brent, R.P., Parallel Implementation of QRD

Algorithms on the Fujitsu AP1000. Report TR-CS-93-12, Computer Sciences Laboratory, ANU, November 1993.